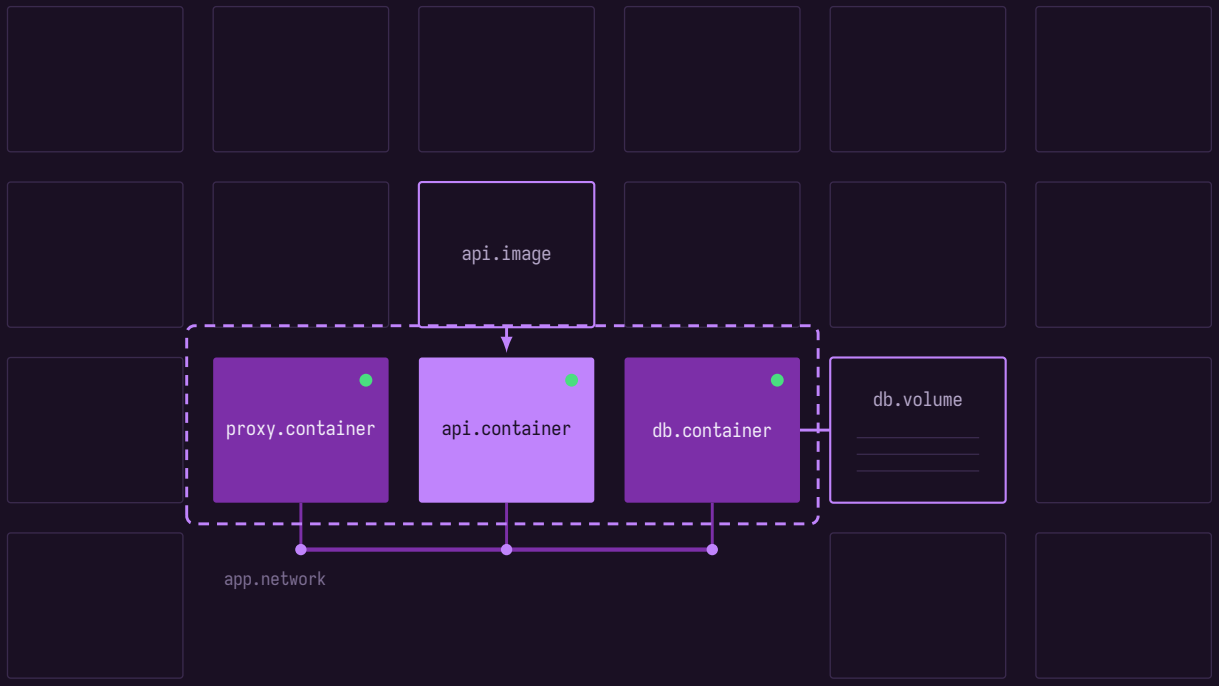

Podman in Production

Containers, Orchestration, and Observability with Podman

Wolfgang Kerschbaumer

sysinit

About This Sample

This sample contains the complete table of contents, the preface, and one full chapter — *Health Checks and Self-Healing* — from **Podman in Production** (332 pages, PDF). The full book is available at podman-book.com.

0.1 The Complete Table of Contents

Every example in the book was verified against real Podman 6 systems; purchases include the examples archive and updates for the Podman 6.x line.

Preface

This is a book about running containers in production with Podman, and about the quietly radical idea behind it: a container is a Linux process, nothing more, and the operating system you already run, with the init system you already trust, is a perfectly good place to orchestrate it.

The book targets Podman 6.x specifically. Podman 6 is a sharper tool than its predecessors: cgroups v1 is gone, the legacy network stacks (CNI, slirp4netns) are gone, BoltDB is gone, iptables is gone from the firewall path. What remains is a smaller, more coherent system (crun, netavark with nftables, pasta, SQLite, systemd), and this book describes only that system. Where you are migrating from older deployments, short labeled migration notes point the way; nothing deprecated is taught as current practice.

Who this book is for. Practicing DevOps and platform engineers who already operate Linux systems and know Docker or another container runtime. We do not explain what a shell is. We do explain, precisely, what a user namespace is, how pasta gets packets out of a rootless container, why an OCI-format image silently loses its `HEALTHCHECK`, and what actually happens between `systemctl start` and a running container.

What makes this book different. Three commitments:

1. *Source-grounded.* The content was researched against the Podman, Buildah, Skopeo, and Netavark source trees, not against blog posts or folk memory. Where the documentation and the code disagree, the code wins, and we say so.
2. *Verified.* Key examples ran against real Podman 6 systems (Fedora with podman-next builds, and podman machine on macOS). Output shown as verified is real output.
3. *Honest about alternatives.* Part VI deploys the same application on Podman with Quadlet, Docker Swarm, HashiCorp Nomad, and Kubernetes, and tells you plainly where Podman's single-node story ends and a scheduler earns its complexity.

The through-line. Parts I–IV build the foundations: how containers actually work, the image lifecycle with Buildah and Skopeo, networking, and security. Part V is the heart of the book: orchestrating real workloads with systemd and Quadlet. It covers health checks, self-healing, automatic updates, and observability with OpenTelemetry, and it ends with a complete production deployment. Part VI places Podman in the container-native world it inhabits, Kubernetes included.

A small reference application, `snip`, accompanies the orchestration chapters: an HTTP service with a database, a reverse proxy, health endpoints, and OpenTelemetry instrumentation. It is deliberately boring. Production infrastructure should be.

Conventions. Rootless examples use a `$` prompt; the rare root-required examples use `#` and say why. The book's version baseline is the Podman 6.0 matched set; the verification lab additionally ran 6.1 development builds, so captured transcripts show either version string. Behavior was checked against both unless noted. Image references are fully qualified. Code lines wrap at 88 characters. Chapters end with a short list of takeaways. Every example in the book ships in runnable form in the examples archive that accompanies it.

Wolfgang Kerschbaumer, July 2026

1 Health Checks and Self-Healing

Restart policies and systemd's `Restart=` (covered in *Running Containers* and *systemd and Quadlet*) handle the failure mode where the process dies. Production incidents are rarely that polite: the process keeps running while it deadlocks, exhausts a connection pool, or serves errors. A health check gives Podman an executable definition of "working", and Podman 6 can act on the answer: killing, restarting, or stopping the container, gating systemd dependencies on healthiness, and emitting events your monitoring can alert on. This chapter covers how checks are defined and scheduled in 6.x, startup probes for slow starters, on-failure actions, and the patterns that replace `sleep`-loop "wait-for" scripts with real readiness gating.

1.1 Anatomy of a Health Check

A health check is a command that Podman executes *inside* the container on a schedule. The contract is the exit code: `0` means healthy, anything else is a failure. Podman accepts the command in two forms, mirroring Docker: a plain string is wrapped as `CMD-SHELL` and executed via `/bin/sh -c` inside the container (so the image must contain a shell), while a JSON array becomes `CMD` and is exec'd directly with no shell involved. A third form, `NONE`, disables the check entirely.

Since the health-check refactor that landed in the 5.5 development cycle, each check runs as a lightweight exec session that is never written to the database. Results go to a small JSON state file (`healthcheck.log`) next to the container's run directory, written atomically with mode `0600`. Two practical consequences: health checks leave no exec sessions behind in the database, and a busy fleet of frequent checks causes no database churn. (Health checks stopped emitting `exec` events much earlier, in the 4.2 cycle.)

A container's health state is one of four values:

- `starting` — the container started, but no check has succeeded yet (or a startup probe is still running; see below).
- `healthy` — the most recent check succeeded.
- `unhealthy` — checks failed `--health-retries` times in a row.
- `stopped` — the container stopped while a check was in flight.

The transition logic is a failing streak. Exit code `0` sets the status to `healthy` and resets the streak to zero. A non-zero exit increments the streak; when the streak reaches `--health-retries` (default 3), the status flips to `unhealthy`. Failures during `--health-start-period` (default `0s`) are forgiven: they do not increment the streak, and the status stays `starting`. A single success during the start period, though, flips the container to `healthy` immediately. A check that runs longer than `--health-timeout` (default 30s) is killed with SIGKILL and recorded as a failure with exit code `-1` and a timeout error. Restarting a container wipes the health log and resets the status; health state never survives a restart.

The defaults, for reference: interval 30s, retries 3, timeout 30s, start period 0s.

1.1.1 Designing the probe command

The probe runs inside the container image, so it can only use what the image ships. Production images built `FROM` slim or distroless bases frequently lack `curl`, and a health check that exits `127` (command not found) looks exactly like an unhealthy application from Podman's perspective. The `snip` reference

application (see the Part V introduction) is built on `docker.io/library/python:3.14-slim`, which has no `curl` or `wget`; its probe uses the interpreter that is guaranteed to be there:

```
python -c 'import urllib.request as u;u.urlopen("http://localhost:8000/healthz")'
```

`urlopen` raises on any non-2xx status, so the exit code carries the result without explicit checks. Beyond existence, two rules serve well:

- **Probe the serving path, not the process.** `pgrep uvicorn` proves nothing about whether requests succeed. An HTTP request to a real endpoint does.
- **Separate liveness from readiness.** `snip` exposes `/healthz` (process is up, never touches the database) and `/readyz` (database answers a query). Wiring the *readiness* endpoint into a check that triggers `--health-on-failure=kill` means a database outage gets your stateless API killed and restarted in a loop, which accomplishes nothing. Use liveness semantics for checks that trigger destructive actions; use readiness semantics for startup gating (covered under startup probes and `sdnotify=healthy` below).

1.2 Defining Checks: Image, CLI, Quadlet

There are three places a health check can be defined, and in Podman 6 they are not equally trustworthy.

1.2.1 The Containerfile HEALTHCHECK and the OCI format trap

Containerfiles support the `HEALTHCHECK` instruction with the flags `--interval`, `--timeout`, `--start-period`, `--start-interval`, and `--retries`:

```
FROM docker.io/library/nginx:1.29-alpine

HEALTHCHECK --interval=10s --timeout=3s --retries=3 \
  CMD wget -qO /dev/null http://localhost/ || exit 1
```

The trap: **`HEALTHCHECK` is not part of the OCI image specification**, and `podman build` produces OCI-format images by default. The instruction is parsed, the build emits a single log-line warning (`HEALTHCHECK is not supported for OCI image format and will be ignored`), the build succeeds, and the check is dropped when the image config is written: a default build of the Containerfile above produces an image whose `.Config.Healthcheck` is `null`, while the same build with `--format docker` preserves the check, visible in `podman image inspect`. A warning scrolling past in CI output goes unnoticed; the exit code stays zero either way.

```
$ podman build -t localhost/hc-trap:oci .   # OCI, the default format
...
[warning] HEALTHCHECK is not supported for OCI image format and will be
ignored. Must use `docker` format
$ podman build --format docker -t localhost/hc-trap:docker .   # kept
...
$ podman image inspect --format '{{json .Config.Healthcheck}}' \
    localhost/hc-trap:oci
```

```
null
$ podman image inspect --format '{{json .Config.Healthcheck}}' \
    localhost/hc-trap:docker
{"Test":["CMD-SHELL","wget -qO /dev/null http://localhost/ || exit 1"],...}
```

The runnable pair lives in `examples/ch16-healthchecks/format-trap/`. The same applies to `buildah config --healthcheck` and friends, which set identical fields: Docker format keeps them, OCI format drops them; buildah's own documentation states the setting "is discarded when writing images using OCIv1 formats". Setting `BUILDAH_FORMAT=docker` changes the default format process-wide.

When a container is created from a Docker-format image that carries a health check, Podman honors it automatically: the container gets a check and a timer with no flags required. Since 5.5, individual `--health-*` options override the corresponding image values without requiring you to respecify `--health-cmd`.

> **Note:** Podman parses the `HEALTHCHECK --start-interval` build flag but does not consume it at runtime, and there is no `--health-start-interval` flag on `podman run`. Podman's startup probes (below) are the equivalent mechanism, and a more capable one.

The book's recommendation follows from the trap: **define health checks in the run configuration (Quadlet keys or `--health-*` flags), not in the image.** A check baked into an image only survives if every build in the pipeline remembers `--format docker`, vanishes with nothing but a build warning the first time someone forgets, and cannot be tuned per deployment anyway. The `snip` Containerfile keeps its `HEALTHCHECK` instruction for consumers that build Docker-format images, but every deployment example in this book defines the check where it is versioned with the deployment: the unit file.

1.2.2 Runtime definition with –health-*

Everything the instruction can express, plus everything it cannot, is available at `podman run`/`podman create` time:

```
# snip creates its schema at startup, so it needs a reachable database even
# though the liveness probe below never touches it.
podman network create snip-net
podman run -d --name snip-db --network snip-net \
  -e POSTGRES_USER=snip -e POSTGRES_PASSWORD=demo-only -e POSTGRES_DB=snip \
  docker.io/library/postgres:18

probe='import urllib.request as u;u.urlopen("http://localhost:8000/healthz")'

podman run -d --name snip-api --network snip-net -p 8000:8000 \
  -e DATABASE_URL=postgresql://snip:demo-only@snip-db:5432/snip \
  --health-cmd "python -c '$probe'" \
  --health-interval 10s \
  --health-retries 3 \
  --health-timeout 5s \
  localhost/snip:dev
```

The liveness probe hits `/healthz`, which never touches the database. `snip` still creates its schema at startup, though, so it needs the database to boot; the full script is `examples/ch16-healthchecks/standalone/run-snip-health.sh`. `--health-cmd none` or `--no-healthcheck` disables an image-defined check; `--health-interval disable` keeps the check defined but creates no timer, leaving only manual runs.

1.2.3 Quadlet keys

In `.container` units (the deployment vehicle this book advocates), all fourteen health options map one-to-one to the CLI flags. The complete set, from `podman-container.unit(5)`:

Quadlet key	CLI flag
`HealthCmd=`	`--health-cmd`
`HealthInterval=`	`--health-interval`
`HealthRetries=`	`--health-retries`
`HealthTimeout=`	`--health-timeout`
`HealthStartPeriod=`	`--health-start-period`
`HealthOnFailure=`	`--health-on-failure`
`HealthLogDestination=`	`--health-log-destination`
`HealthMaxLogCount=`	`--health-max-log-count`
`HealthMaxLogSize=`	`--health-max-log-size`
`HealthStartupCmd=`	`--health-startup-cmd`
`HealthStartupInterval=`	`--health-startup-interval`
`HealthStartupRetries=`	`--health-startup-retries`
`HealthStartupSuccess=`	`--health-startup-success`
`HealthStartupTimeout=`	`--health-startup-timeout`

> **Version:** Podman 6.0 fixed `HealthCmd=` values containing double quotes producing broken health checks (#28409). Probe commands with nested quoting (common for `python -c` one-liners) now pass through the generator intact.

> **Note:** Quadlet's unit parser supports \ line continuations, but unlike systemd it concatenates the joined lines *without* inserting a space. Long `HealthCmd=` values can be wrapped, provided the split point tolerates direct concatenation.

1.3 How Checks Run: Transient systemd Timers

Podman is daemonless; there is no background process to wake up every thirty seconds. The scheduler is systemd. When a container with a health check starts, Podman calls `systemd-run` to create a transient timer/service pair: in the system manager for root containers, in the *user* manager for rootless ones. The unit name is the full container ID plus a random hex suffix (a `-startup` infix marks startup-probe timers); the random suffix prevents unit-name collisions when a container restarts.

The transient units are created with deliberate properties worth knowing when you debug them:

- `--on-unit-inactive=<interval>` — the next check fires *interval* after the previous check's unit went inactive, not on a fixed grid. Long-running probes therefore stretch the effective period.
- `AccuracySec=1s` — systemd's default timer coalescing window is one minute; without this a "10s" interval would drift badly.
- `LogLevelMax=notice` — keeps per-check start/stop chatter out of the journal.
- `StartLimitIntervalSec=0` — disables start-rate limiting for the transient unit.

  Version: Two 6.0 fixes matter operationally. First, `StartLimitIntervalSec=0`: before 6.0, systemd's start-rate limiting could silently stop frequent health checks with "Start request repeated too quickly". Second, the scheduled command is now `podman healthcheck run --ignore-result <id>`: the new `--ignore-result` flag makes the transient service exit 0 regardless of the check result, so failing checks no longer strand failed transient units in the manager. If you are coming from 5.x: periodic `systemctl --user reset-failed` cleanup rituals for health-check units are obsolete and must not be carried forward. 6.0 also forwards global storage arguments (`--root`, `--transient-store`, …) into the timer's command line, fixing silent check failures for containers on non-default storage.

The timer follows the container's lifecycle: created at container init, started when the container enters `Running` (which also sets the status to `starting`), removed on pause, re-created on unpause, and removed at stop and cleanup. You can watch the machinery on any rootless host:

```
ctr_id="$(podman inspect --format '{{.Id}}' snip-api)"
systemctl --user list-timers --all | grep "${ctr_id:0:12}"
```

Two boundary cases: `--health-interval disable` (interval `0`) skips all systemd unit handling while keeping the check defined for manual invocation, and on hosts without systemd (or FreeBSD) no timers exist at all; health checks there run only when you call `podman healthcheck run` yourself.

1.4 Observing Health

Health state surfaces in five places; knowing all five saves time under pressure.

`podman ps` appends the state to the STATUS column (`Up 3 minutes (healthy)`) and filters on it, which makes fleet triage a one-liner:

```
podman ps --filter health=unhealthy
```

podman inspect exposes the full state under `.State.Health`: current status, the failing streak, and the recent check log with timestamps, exit codes, and captured output:

```
podman inspect --format \
  '{{.State.Health.Status}} streak={{.State.Health.FailingStreak}}' snip-api
podman inspect --format '{{json .State.Health}}' snip-api | jq
```

The check *configuration* lives under `.Config.Healthcheck`, with startup probes in `.Config.StartupHealthCheck` and the on-failure action in `.Config.HealthcheckOnFailureAction`. The `.State.Healthcheck` key still exists as a backward-compatibility alias; write new tooling against `.State.Health`.

podman healthcheck run <ctr> triggers a check on demand and reports through its exit code: `0` healthy, `1` failed, `125` error (no such container, no check defined, container not running). `podman --log-level debug healthcheck run` shows why a check fails; it is the fastest way to debug a probe command.

podman wait blocks on health state, which turns "poll until up" scripts into one line:

```
podman wait --condition=healthy snip-db && ./run-migrations.sh
```

`--condition=unhealthy` works symmetrically, and you can give several conditions; the command returns as soon as any of them matches.

podman events emits a `health_status` event on every executed check, carrying the resulting status:

```
podman events --filter event=health_status \
  --format '{{.Time}} {{.Name}} {{.HealthStatus}}'
```

With the journald events backend the status is also a structured journal field, `PODMAN_HEALTH_STATUS`, so `journalctl` filtering and log shippers get it for free. Event emission is controlled by `healthcheck_events` (default `true`) in the `[engine]` section of `containers.conf`; on hosts with hundreds of containers checking every few seconds, setting it to `false` trades audit trail for journal volume. Alerting on these events is covered in *Observability and OpenTelemetry*.

1.4.1 The health log and fleet tuning

The state file behind all of this defaults to `--health-log-destination local`: a `healthcheck.log` JSON file beside the container's run directory, which typically lives on tmpfs and does not survive a reboot. Point it at a directory (`--health-log-destination /var/log/podman-hc` creates `<ctr-id>-healthcheck.log` there) for persistent post-mortem data, or at `events_logger` to fold check output into the events stream. The log is bounded by `--health-max-log-count` (default 5 entries) and `--health-max-log-size` (default 500 characters per entry); `0` means unlimited, which is rarely what a long-lived host wants.

1.5 Startup Probes

`--health-start-period` forgives failures for a fixed duration and is fine when startup time is known and short. It has two weaknesses: the regular check still runs at its regular (usually relaxed) interval during startup, so a container that becomes ready in 2 seconds may not be reported healthy for 30; and one number must cover the worst case: a database that usually starts in 5 seconds but takes 10 minutes after a crash recovery forces you to forgive failures for 10 minutes, always.

Startup probes solve both. `--health-startup-cmd` defines a *separate* check with its own cadence that runs first; the regular check's timer is not created until the startup probe passes. The knobs:

- `--health-startup-cmd` — the probe command; requires a regular health check to be defined, and can only be set via CLI, Quadlet, or Kubernetes YAML, never from an image.
- `--health-startup-interval` (default 30s) and `--health-startup-timeout` (default 30s) — cadence and per-run limit, independent of the regular check.
- `--health-startup-success` (default 0 = a single success) — consecutive successes required before startup is considered passed.
- `--health-startup-retries` (default 0 = never) — consecutive failures after which Podman restarts the container outright.

While the startup probe is active the container reports `starting`, and `podman healthcheck run` executes the startup command rather than the regular one. On success, the startup timer is torn down and the regular-interval timer replaces it; the handover is visible in `systemctl --user list-timers` as the `-startup-` unit disappearing. On exhausted retries the container is restarted by the startup-probe mechanism itself; the man page suggests the restart is subject to `--health-on-failure`, but the libpod code path restarts directly without consulting the on-failure action, so do not rely on `--health-on-failure=none` to suppress it.

The canonical use case is a database. PostgreSQL is probed every 2 seconds during startup and every 30 seconds thereafter:

```
podman run -d --name hc-db \
  -e POSTGRES_USER=snip -e POSTGRES_PASSWORD=demo-only -e POSTGRES_DB=snip \
  --health-cmd 'pg_isready -h 127.0.0.1 -U snip -d snip' \
  --health-interval 30s \
  --health-retries 3 \
  --health-timeout 5s \
  --health-startup-cmd 'pg_isready -h 127.0.0.1 -U snip -d snip' \
  --health-startup-interval 2s \
  --health-startup-retries 60 \
  docker.io/library/postgres:18
```

> **Note:** The probe deliberately uses TCP (`-h 127.0.0.1`) rather than the default Unix socket. The official postgres image runs a *temporary* server during first-time initialization that listens only on the socket; a socket probe can report ready while the real server has not yet started. A TCP probe cannot hit the temporary server.

The runnable version is `examples/ch16-healthchecks/startup-probe/postgres-startup.sh`.

For completeness: `podman kube play` maps Kubernetes `livenessProbe` to the regular health check and `startupProbe` to a startup probe (synthesizing a trivial regular check if the YAML defines none), converts

HTTP and TCP handlers to probe commands, and applies Kubernetes-flavored defaults (10s interval, 3 retries) plus Kubernetes semantics for `initialDelaySeconds`. Details belong to *Kubernetes Workflows*.

1.6 Self-Healing with –health-on-failure

Detection alone is monitoring; self-healing needs a reaction. `--health-on-failure` defines what Podman does when the status transitions to `unhealthy` (it never fires for startup probes, and never while the streak is still below the retry threshold):

- `none` (default) — record the state, take no action.
- `kill` — SIGKILL the container. The kill marks the container as explicitly stopped, so Podman's own `--restart` policy will *not* revive it; `kill` is meant for the systemd case, where the container's non-zero exit trips the unit's `Restart=`.
- `restart` — stop the container and let Podman's cleanup process restart it. This is deliberately not an in-place restart: the restart must not run in the context of the transient timer unit that triggered the check. The cleanup-driven restart happens independently of the `--restart` policy.
- `stop` — stop the container and leave it stopped.

Podman rejects any action other than `none` if no health check is defined.

Which action to pick depends on who owns the restart:

Standalone containers (no systemd unit): use the `restart` action on its own. The check detects, `restart` stops the container, and Podman's cleanup process brings it back. No `--restart` policy is involved, and combining the two is explicitly discouraged:

```
podman run -d --name hc-demo \
  --mount type=tmpfs,dst=/probe \
  --health-cmd 'test ! -f /probe/fail' \
  --health-interval 5s --health-retries 2 --health-timeout 3s \
  --health-on-failure restart \
  docker.io/library/nginx:1.29-alpine
```

This demo (full script: `examples/ch16-healthchecks/standalone/self-heal.sh`) uses a synthetic probe, a flag file on a tmpfs, so that failure injection is deterministic and the restart genuinely heals: `podman exec hc-demo touch /probe/fail` drives the container unhealthy, Podman restarts it, and the fresh tmpfs makes it healthy again. The whole cycle is observable without polling:

```
podman wait --condition=unhealthy hc-demo   # streak exhausted
podman wait --condition=healthy hc-demo     # restarted, recovered
podman events --since 2m --stream=false \
  --filter event=health_status --filter event=died --filter event=restart
```

Do **not** substitute `--health-on-failure=kill` here in the hope that a `--restart` policy will heal the container: the kill flags the container as explicitly stopped, which suppresses `--restart` entirely, so a killed standalone container stays down. `kill` pays off only under systemd (next), where the *unit's* `Restart=`, not Podman's restart policy, owns recovery.

Containers under systemd (Quadlet): use `kill` or `stop` and let systemd own the restart. systemd is the better supervisor: it has backoff (`RestartSec=`), rate limiting (`StartLimitIntervalSec=`, `StartLimitBurst=`), dependency propagation, and a journal trail. The recommended pattern in a `.container` unit:

```
[Container]
HealthCmd=pg_isready -h 127.0.0.1 -U snip -d snip
HealthInterval=30s
HealthOnFailure=kill

[Service]
Restart=on-failure
```

The kill takes conmon and the service's main process down with a non-zero status; `Restart=on-failure` starts a fresh container. Quadlet `.container` units set no `Restart=` by default; self-healing is opt-in, one line.

> **Warning:** Do not use `--restart` policies for containers managed by systemd units; the two restart managers will race each other. Inside units, `Restart=` is the only restart mechanism you should configure. This is also the upstream recommendation in `podman-run(1)`.

1.7 Readiness Gating with sdnotify=healthy

Self-healing handles containers that break; the other half of orchestration is not starting things *before their dependencies work*. Podman's bridge between health checks and systemd's dependency engine is `--sdnotify=healthy`.

`--sdnotify` controls how Podman deals with systemd's `NOTIFY_SOCKET` under `Type=notify` services. Four modes exist: `container` (the CLI default: the socket is proxied into the container so the application can announce its own readiness), `conmon` (READY is sent when the container has started), `ignore`, and `healthy`. With `healthy`, Podman sets the service's main PID to conmon and sends `READY=1` only once the container's health status reaches `healthy`. The notify socket is never exposed to the container, so the application needs no sd_notify support. Podman polls the health state every 250ms after start, so readiness propagates promptly regardless of the check interval. A health check is mandatory; container creation fails without one.

In Quadlet the switch is the `Notify=` key. Its default is `false`, which maps to `--sdnotify=conmon`; note this differs from the CLI default. `Notify=true` maps to `--sdnotify=container`, and `Notify=healthy` to `--sdnotify=healthy`. Quadlet generates `Type=notify` services either way, so changing the value changes only *when* READY fires, and therefore when the systemd start job completes:

- `Notify=healthy` makes `systemctl start my-app.service` block until the container is healthy.
- Every unit ordered `After=my-app.service` now starts after the application *works*, not after its process forked.
- If the container never turns healthy, the start job times out after `TimeoutStartSec=` (systemd's default is 90s), the unit fails, and `Restart=` policy applies. Budget the timeout for the worst-case startup: a database replaying WAL can legitimately need minutes.

This one key turns systemd ordering from "start order" into readiness orchestration, and the dependency patterns that follow build on it.

1.8 Dependencies Without Sleep Loops

Every operator has seen the anti-patterns: `sleep 15` in a unit's `ExecStartPre=`, entrypoint wrappers that poll a port in a shell loop, or "restart until the database answers" as an implicit startup strategy. They all encode a guess about timing instead of a fact about state. Podman 6 plus systemd can express the fact directly.

1.8.1 Pattern 1: a readiness-gated Quadlet chain

The `snip` API cannot start before PostgreSQL accepts connections: its startup code creates the schema, so a premature start crashes. The complete, runnable pair of units (`examples/ch16-healthchecks/quadlet/`) wires the dependency through health checks. The database unit:

```
# ~/.config/containers/systemd/snip-db.container
[Unit]
Description=snip PostgreSQL

[Container]
Image=docker.io/library/postgres:18
ContainerName=snip-db
Network=snip.network
Volume=snip-db:/var/lib/postgresql
Environment=POSTGRES_USER=snip
Environment=POSTGRES_DB=snip
Secret=snip-db-password,type=env,target=POSTGRES_PASSWORD

HealthCmd=pg_isready -h 127.0.0.1 -U snip -d snip
HealthInterval=30s
HealthRetries=3
HealthTimeout=5s
HealthStartupCmd=pg_isready -h 127.0.0.1 -U snip -d snip
HealthStartupInterval=2s

HealthOnFailure=kill
Notify=healthy

[Service]
Restart=on-failure
TimeoutStartSec=300

[Install]
WantedBy=default.target
```

And the API unit that depends on it:

```
# ~/.config/containers/systemd/snip-api.container
[Unit]
Description=snip API
Requires=snip-db.service
After=snip-db.service
```

```
[Container]
Image=localhost/snip:dev
ContainerName=snip-api
Network=snip.network
PublishPort=8000:8000
Environment=BASE_URL=http://localhost:8000
Secret=snip-db-url,type=env,target=DATABASE_URL

HealthCmd=python -c 'import urllib.request as u;\
  u.urlopen("http://localhost:8000/healthz")'
HealthInterval=10s
HealthRetries=3
HealthTimeout=5s

HealthOnFailure=kill
Notify=healthy

[Service]
Restart=on-failure

[Install]
WantedBy=default.target
```

The moving parts, and why each is there:

- `Notify=healthy` on the database turns `snip-db.service`'s activation into "PostgreSQL answers `pg_isready`". `After=snip-db.service` on the API therefore means "after the database is *healthy*". The sleep loops and retry wrappers have nothing left to do.
- The startup probe polls every 2 seconds, so the API starts within seconds of the database becoming ready instead of waiting out a 30s regular interval; `TimeoutStartSec=300` bounds the wait for a slow first `initdb`.
- The API's own `Notify=healthy` propagates the same guarantee upward: a reverse proxy unit ordered after `snip-api.service` starts against a working API. The capstone chapter, *A Production Deployment with Quadlet*, extends exactly this chain.
- The API probes `/healthz` (liveness), not `/readyz`: with `HealthOnFailure=kill`, a database outage should not get the healthy-but-degraded API killed in a loop while systemd's `Restart=` burns through its start limit.
- Credentials come from the secrets subsystem (see *The Podman Security Model*); the network unit `snip.network` provides container-name DNS (see *Networking with Netavark*).

```
./quadlet/create-secrets.sh
cp quadlet/*.{network,container} ~/.config/containers/systemd/
systemctl --user daemon-reload
systemctl --user start snip-api.service   # blocks: db healthy, then api healthy
```

Be precise about what this guarantees: **ordering is enforced at start time only.** If PostgreSQL dies at 3 a.m., systemd restarts `snip-db.service` (its own `Restart=on-failure`), but `Requires=` does not restart the API for a dependency's crash-restart; the API keeps running and must tolerate reconnecting. `snip` does, by opening short-lived connections per request; an application holding a pool needs reconnect

logic regardless of the orchestrator. Health-gated ordering removes startup races; it does not remove the need for runtime resilience.

1.8.2 Pattern 2: gating one-shot work

Migrations, seed jobs, and backup verification want the same gate without a service. `podman wait --condition=healthy` is the tool, either in scripts or as a `Type=oneshot` unit sandwiched between the database and the application:

```
[Unit]
Description=snip schema migrations
Requires=snip-db.service
After=snip-db.service

[Service]
Type=oneshot
ExecStart=/usr/local/bin/snip-migrate.sh
```

With `Notify=healthy` on the database unit, the `After=` ordering already guarantees a healthy database by the time `ExecStart=` runs; the oneshot needs no waiting logic at all. Outside systemd, the same gate is a single line: `podman wait --condition=healthy snip-db && ./run-migrations.sh`.

For the development loop, Compose expresses the same idea as `depends_on: {condition: service_healthy}` (`snip`'s `compose.yaml` uses it), but that is dev tooling; see *Compose and the Docker API*.

1.9 Reconfiguring Health Checks Live

Every health setting can be changed on an existing (even running) container with `podman update`, which accepts the full flag set: `--health-cmd`, `--health-interval`, `--health-retries`, `--health-timeout`, `--health-start-period`, the five `--health-startup-*` flags, `--health-on-failure`, the log options, and `--no-healthcheck` (which conflicts with all other health flags). Changing an interval on a running container tears down and recreates the transient timer transparently:

```
podman update --health-interval 5s snip-api      # tighten during an incident
podman update --health-max-log-count 20 snip-api # keep more forensic history
podman update --no-healthcheck snip-api          # silence a flapping check
```

`podman update --restart` adjusts the restart policy of standalone containers the same way.

> **Warning:** Avoid `podman update --health-startup-*` on a container whose startup probe has not yet passed. In Podman 6.x (confirmed on both 6.0 and 6.1) the update recreates the startup timer with an unset interval (`OnUnitInactiveSec=0`, so the timer's next elapse is `infinity`): the probe never fires again and the container is stranded in `starting`. Change startup-probe settings by recreating the container, not with a live update.

For Quadlet-managed containers, treat `podman update` as an incident tool, not a configuration channel: the container's config now disagrees with its unit file, and the change evaporates on the next `systemctl restart`. Make the change permanent in the `.container` file, reload, and restart the service.

1.10 Takeaways

- Health checks run inside the container via lightweight exec sessions; state lives in a JSON file, statuses are `starting`/`healthy`/`unhealthy`/`stopped`, and `unhealthy` fires after `--health-retries` consecutive failures.
- Default (OCI-format) `podman build` drops `HEALTHCHECK`, leaving only a build-log warning; `--format docker` preserves it. Define checks in run config (Quadlet `Health*=` keys or `--health-*` flags), not in the image.
- Scheduling is per-container transient systemd timers (user manager when rootless); 6.0 added `--ignore-result` and disabled start-rate limiting, ending the failed-unit litter and silently stopped checks of 5.x.
- Startup probes (`--health-startup-*`) probe fast during initialization and hand over to the regular check; use them instead of oversized `--health-start-period` values.
- For self-healing under systemd: `HealthOnFailure=kill` + `Restart=on-failure`. Never combine the `restart` on-failure action with a `--restart` policy, and never use `--restart` inside systemd units.
- `Notify=healthy` makes a unit's activation mean "container is healthy", turning `After=`/`Requires=` into readiness gates, the replacement for sleep-loop wait scripts. `podman wait --condition=healthy` covers the non-service cases.
- Ordering is a start-time guarantee only; applications still need reconnect logic for dependencies that restart at runtime.
- Watch health via `podman ps --filter health=`, `.State.Health` in inspect, `health_status` events (journald field `PODMAN_HEALTH_STATUS`), and `podman healthcheck run` for on-demand probes with `--log-level debug` for diagnosis.