Podman Zero-to-Expert Course

2026-02-24

Contents

1 Front Matter

1.1 README.md

2 Podman Zero-to-Expert Course (Draft)

This is a course-in-a-repo for taking a learner from zero container knowledge to running rootless Podman services with systemd (Quadlet), with strong security and troubleshooting fundamentals.

2.1 How To Use This Repo

- Read the modules in `modules/` in order.
- Do the labs as you go; each module includes a small checklist.
- Keep everything rootless unless the module explicitly says otherwise.

2.2 Structure

- `COURSE_OUTLINE.md`: the full syllabus and learning goals
- `MODULES.md`: reading order
- `modules/`: lesson content (Markdown)
- `cheatsheets/`: quick references
- `examples/`: example YAML and unit files
- `ASSESSMENTS.md`: practical exams and rubrics
- `FAQ.md`: common gotchas and fast fixes

Suggested path:

- Start with `modules/00-setup.md`
- Continue in numeric order

2.3 Safety

- Prefer rootless Podman.
- Never put secret material in images, unit files, or logs.
- If you are on a shared system, treat this repo's lab values as examples only.

2.4 Conventions

- Secrets are delivered as files mounted at runtime (not environment variables).
- Production baseline uses rootless Podman + systemd user services (Quadlet-first).

2.5 Suggested Pacing

See `COURSE_OUTLINE.md` for rough time estimates per module.

2.6 COURSE_OUTLINE.md

3 Course Outline

Goal: take an absolute beginner to an operator who can build, run, secure, and troubleshoot rootless Podman workloads, including systemd (Quadlet) production patterns.

Assumptions:

- Learners have basic command line familiarity by the end of Module 2.
- Labs target Fedora/RHEL-like systems with systemd. Where commands differ across distros, modules call it out.

3.1 Modules

0. Setup (install, rootless prerequisites, verification)
1. Containers 101 (images vs containers, OCI, rootless)
2. Everyday Podman commands (`run`, `exec`, `logs`, lifecycle)
3. Images and registries (tags vs digests, inspect, provenance basics)
4. Secrets (local-first with Podman secrets; rotation patterns)
5. Storage basics (volumes, bind mounts, permissions, SELinux notes)
6. Networking (ports, DNS, user-defined networks)
7. Pods and sidecars (Podman pods, shared network namespace)
8. Building images (Containerfiles, multi-stage, non-root)
9. Multi-service workflows (scripted, compose-ish patterns, `play kube` preview)
10. `podman play kube` (Kubernetes YAML locally, parity concepts; YAML secrets caveats)
11. Production baseline: systemd + Quadlet (rootless services, restart, upgrades; secrets add-on) 11a. Quadlet secrets add-on (runtime file secrets + rotation)
12. Security deep dive (capabilities, seccomp/SELinux, read-only FS)
13. Troubleshooting and ops (events, logs, journald, failure drills)
14. Maintenance and auto-updates (policy, `podman auto-update`, safe rollouts)
15. Capstone (Quadlet stack with secrets, backups, upgrades, rollback)
16. External secrets survey (thorough intro; optional implementation paths)

3.2 Suggested Pacing (Rough Estimates)

These are rough time boxes for a first pass (reading + doing the labs).

- Module 0: 30-60 min
- Module 1: 30-45 min
- Module 2: 60-90 min
- Module 3: 60-90 min
- Module 4: 60-90 min
- Module 5: 60-90 min
- Module 6: 2-3 hours
- Module 7: 45-60 min
- Module 8: 2-3 hours
- Module 9: 60-90 min
- Module 10: 45-75 min
- Module 11: 2-3 hours
- Module 11a: 45-75 min
- Module 12: 60-120 min
- Module 13: 60-120 min
- Module 14: 45-75 min
- Module 80 (Capstone): 3-6 hours
- Module 90 (Survey): 45-90 min

3.3 MODULES.md

4 Modules (Reading Order)

- modules/00-setup.md
- modules/01-containers-101.md
- modules/02-everyday-commands.md
- modules/03-images-registries.md
- modules/04-secrets.md
- modules/05-storage.md
- modules/06-networking.md
- modules/07-pods.md
- modules/08-building-images.md
- modules/09-multiservice-workflows.md
- modules/10-play-kube.md
- modules/11-quadlet.md
- modules/11-quadlet-secrets.md
- modules/12-security.md
- modules/13-troubleshooting.md
- modules/14-autoupdate.md
- modules/80-capstone.md
- modules/90-external-secrets-survey.md

5 Modules

6 Module 0: Setup (Fedora/RHEL + systemd)

This course targets Fedora/RHEL-like systems with systemd, using rootless Podman.

6.1 Learning Goals

- Install Podman and verify basic functionality.
- Confirm your system supports rootless containers (subuid/subgid).
- Know where logs and state live for rootless Podman.
- Confirm cgroups v2 is enabled (required for Quadlet).
- Set up a lab workspace and a few safety defaults.

6.2 Install

Fedora:

```
sudo dnf install -y podman  # install Podman
```

RHEL (package availability depends on subscription/repos):

```
sudo dnf install -y podman  # install Podman
```

Verify:

```
podman --version  # show Podman version
podman info  # show Podman host configuration
```

Record your versions (useful for debugging later):

```
podman --version  # show Podman version
rpm -q podman 2>/dev/null || true  # show installed package version
uname -r  # show kernel/system info
```

6.3 cgroups v2 Check (Required)

Quadlet requires cgroups v2.

Check:

```
podman info --format '{{.Host.CgroupsVersion}}'  # show Podman host configuration
```

Expected: `v2`

6.4 Rootless Prereqs

Rootless Podman uses user namespaces. Your user typically needs subuid/subgid ranges.

Check:

```
grep "^$USER:" /etc/subuid /etc/subgid  # filter output
```

If those are missing, create them (example range; coordinate with your admin policy):

```
sudo usermod --add-subuids 100000-165535 --add-subgids 100000-165535 "$USER"  # grant subuid/subgid ran
```

Log out and back in after updating subuids/subgids.

6.5 SELinux Quick Check (Fedora/RHEL)

SELinux is usually enabled on these systems. You do not need to understand the full policy model, but you should recognize when it affects bind mounts.

Check status:

```
getenforce  # show SELinux mode
```

If SELinux is Enforcing, prefer:

- named volumes for data
- bind mounts with `:Z` (private) or `:z` (shared)

6.6 First Container

```
podman run --rm docker.io/library/alpine:latest uname -a  # run a container
```

If this works, you have:

- network access to pull images
- a working storage backend
- a working OCI runtime

6.7 Where Things Live (Rootless)

Common locations:

- Container storage: `~/.local/share/containers/`
- Runtime files: `/run/user/<uid>/containers/`
- Quadlet units: `~/.config/containers/systemd/`

Logs:

- `podman logs <name>` for container logs
- `journalctl --user -u <service>` for Quadlet/systemd logs

6.8 Create a Course Workspace

Pick a working directory for labs:

```
mkdir -p ~/course_podman-labs  # create directory
cd ~/course_podman-labs  # change directory
```

6.9 Recommended Shell Safety

These are optional but reduce accidents:

```
set -o noclobber  # prevent overwriting files with redirects
```

Do not run this in shells where it would surprise you; it changes redirect behavior.

6.10 Checkpoint

- `podman info` runs without errors.
- `podman run --rm ...` works rootless.
- cgroups version reports `v2`.

6.11 Further Reading

- Podman install docs: https://podman.io/docs/installation
- Rootless Podman tutorial: https://github.com/containers/podman/blob/main/docs/tutorials/rootless_tutorial.md
- cgroups v2 (kernel docs): https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html
- `loginctl` (linger for user services): https://www.freedesktop.org/software/systemd/man/latest/loginctl.html
- SELinux with containers (RHEL docs): https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/9/html/u selinux-with-containers_using-selinux

7 Module 1: Containers 101

7.1 Learning Goals

- Explain images vs containers in one sentence.
- Understand what an OCI image is (layers + config).
- Understand why rootless containers matter.
- Understand the boundary: containers are isolation, not a VM.

7.2 The Three Things To Get Right

1) Image: the immutable template you pull/build.

2) Container: a running (or stopped) instance of an image with:

- a process
- a writable layer
- mounts
- network settings

3) Runtime boundaries:

- Namespaces: process IDs, networking, mounts
- cgroups: CPU/memory limits

Important: the kernel is shared.

- A container is not a VM.
- Container security is about reducing blast radius and applying least privilege.

7.3 Rootless vs Rootful

Rootless benefits:

- A container breakout is less likely to become full-host root.
- You can run services without handing developers root.

Rootless tradeoffs:

- Some networking and low ports require extra setup.
- File permissions can be confusing until you learn UID mapping.

Rule of thumb:

- Use rootless by default.
- Use rootful only when you can explain why it is required.

7.4 Terminology You Will See

- OCI: the standards that define images and runtimes.
- runtime: software that starts the container process (Podman uses an OCI runtime under the hood).
- registry: where images live (Docker Hub, Quay, your internal registry).
- short name: a shorthand like `alpine` which resolves to a fully qualified name.

7.5 Lab: Compare Host vs Container

Run a container and compare:

```
uname -a  # show kernel/system info
podman run --rm docker.io/library/alpine:latest uname -a  # run a container
```

Look at processes inside the container:

```
podman run --rm docker.io/library/alpine:latest ps -ef  # run a container
```

You will see a small process tree. This is normal.

Inspect the container config (no need to memorize, just recognize the shape):

```
podman create --name c101 docker.io/library/alpine:latest sleep 300  # create a container without start
podman inspect c101 | less  # inspect container/image metadata
podman rm c101  # remove the stopped container
```

What to look for in `inspect`:

- the image reference
- mount points
- network mode
- environment variables
- user

7.6 Checkpoint

- You can explain: "An image is a template; a container is a running instance."
- You know rootless is the default for this course.

7.7 Quick Quiz (Answer Without Running Commands)

1) If you delete a container, do you delete its image?
2) If a container runs as UID 0 inside, does that mean it is host root?

7.8 Further Reading

- OCI image spec: https://github.com/opencontainers/image-spec
- OCI runtime spec: https://github.com/opencontainers/runtime-spec
- Linux namespaces (man7): https://man7.org/linux/man-pages/man7/namespaces.7.html
- Linux control groups (man7): https://man7.org/linux/man-pages/man7/cgroups.7.html
- Podman overview docs: https://podman.io/docs

8 Module 2: Everyday Podman Commands

8.1 Learning Goals

- Run containers interactively and in the background.
- Use `logs` and `exec` to debug.
- Clean up containers and images safely.
- Understand naming, exit codes, and restart behavior.

8.2 Core Commands

Run and remove when done:

```
podman run --rm docker.io/library/alpine:latest echo hello  # run a container
```

Notes:

- `--rm` removes the container after it exits.
- Prefer `--rm` for short experiments to avoid clutter.

Run a long-lived container:

```
podman run -d --name sleep1 docker.io/library/alpine:latest sleep 600  # run a container
podman ps  # list containers
```

Inspect exit codes:

```
podman wait sleep1  # wait for a container to exit
podman inspect sleep1 --format '{{.State.ExitCode}}'  # inspect container/image metadata
```

View logs:

```
podman logs sleep1  # show container logs
```

Execute a command in a running container:

```
podman exec -it sleep1 sh  # run a command in a running container
```

Tip:

- If an image does not have `sh`, try `bash` or use a debug container on the same network/pod.

Stop/start/remove:

```
podman stop sleep1  # stop a running container
podman start sleep1  # start an existing container
podman rm -f sleep1  # stop (if needed) and force remove the container
```

8.3 Useful Flags (Learn These Early)

- `--name <name>`: stable name for scripts
- `-e KEY=VALUE`: environment variables (avoid for secrets)
- `-v name:/path`: volumes
- `-p host:container`: publish ports
- `--network <net>`: connect to network
- `--user <uid[:gid]>`: run as a specific user

8.4 Lab: The Writable Layer Is Not Persistence

1) Start a container and create a file:

```
podman run -it --name scratch docker.io/library/alpine:latest sh  # run a container
```

Inside the container:

```
echo hi > /tmp/hello.txt  # print text
exit  # exit the shell
```

2) Remove and recreate:

```
podman rm scratch  # remove the container
podman run -it --name scratch docker.io/library/alpine:latest sh  # run a container
```

Inside:

```
ls -la /tmp/hello.txt  # list files
```

Expected: the file is gone.

8.5 Lab: Name Your Containers

1) Run the same image twice with different names:

```
podman run -d --name a1 docker.io/library/alpine:latest sleep 300  # run a container
podman run -d --name a2 docker.io/library/alpine:latest sleep 300  # run a container
podman ps  # list containers
podman rm -f a1 a2  # cleanup both containers
```

2) Try to reuse a name and observe the error.

This teaches you why stable naming matters.

8.6 Cleaning Up Without Nuking Your System

List containers and images:

```
podman ps -a  # list containers
podman images  # list images
```

Remove stopped containers:

```
podman container prune  # remove all stopped containers
```

Remove unused images (be careful; it can remove bases you need):

```
podman image prune  # remove unused images (frees disk)
```

8.7 Checkpoint

- You can use `podman logs` and `podman exec` without guessing.
- You understand why volumes exist.

8.8 Quick Quiz

1) What is the difference between `podman run` and `podman start`?

2) Where do you look first if a container exits immediately?

8.9 Further Reading

- `podman-run(1)`: https://docs.podman.io/en/latest/markdown/podman-run.1.html
- `podman-ps(1)`: https://docs.podman.io/en/latest/markdown/podman-ps.1.html
- `podman-logs(1)`: https://docs.podman.io/en/latest/markdown/podman-logs.1.html
- `podman-exec(1)`: https://docs.podman.io/en/latest/markdown/podman-exec.1.html
- `podman-inspect(1)`: https://docs.podman.io/en/latest/markdown/podman-inspect.1.html

9 Module 3: Images and Registries

9.1 Learning Goals

- Pull images by tag and by digest.
- Understand why digests are safer than tags.
- Inspect image metadata (entrypoint/cmd, exposed ports, labels).
- Understand short-name resolution and why fully qualified names matter.

9.2 Tags vs Digests

- A tag (like `:latest`) is a name that can move.
- A digest (like `@sha256:...`) is content-addressed and does not move.

For production, prefer digests.

9.3 Fully Qualified Image Names

Prefer:

- `docker.io/library/alpine:latest`

Avoid relying on ambiguous short names:

- `alpine:latest`

Reason:

- Short-name resolution rules can vary by system policy.
- Fully qualified names are more predictable in automation.

9.4 Lab: Pull by Tag, Record Digest

```
podman pull docker.io/library/alpine:latest  # pull an image
podman images | grep alpine  # list images
podman inspect docker.io/library/alpine:latest | less  # inspect container/image metadata
```

Find the digest and re-pull by digest:

```
podman images --digests | grep alpine  # list images
```

Try running by digest once you have it:

```
podman run --rm docker.io/library/alpine@sha256:<digest> echo ok  # run a container
```

9.5 Registry Authentication

Login stores credentials for your user:

```
podman login <registry>  # log into a container registry
```

Logout:

```
podman logout <registry>  # log out of a container registry
```

Do not put registry credentials in shell history.

9.6 Image Metadata: ENTRYPOINT vs CMD

Many images define an ENTRYPOINT and/or CMD.

- ENTRYPOINT: the default program
- CMD: default arguments

Inspect these fields:

```
podman image inspect docker.io/library/nginx:stable --format '{{.Config.Entrypoint}} {{.Config.Cmd}}'
```

Practical implication:

- `podman run <image> <args>` appends or replaces depending on ENTRYPOINT.

9.7 Lab (Optional): Push to a Local Registry

This lab teaches the full pull/build/tag/push flow without needing a real external registry.

1) Start a local registry:

```
podman run -d --name registry -p 5000:5000 docker.io/library/registry:2  # run a container
```

2) Tag an image into the local registry namespace and push it:

```
podman pull docker.io/library/alpine:latest  # pull an image
podman tag docker.io/library/alpine:latest localhost:5000/alpine:course  # add another tag/name
podman push localhost:5000/alpine:course  # push an image to a registry
```

If the push fails with a TLS/HTTPS error:

- Some Podman configurations require explicit insecure-registry configuration for plain HTTP registries.
- For this lab only, you can bypass verification:

```
podman push --tls-verify=false localhost:5000/alpine:course  # push an image to a registry
```

Treat `--tls-verify=false` as a learning-only flag, not a production habit.

3) Remove your local copy and pull from the local registry:

```
podman rmi docker.io/library/alpine:latest       # remove local copy (forces re-pull)
podman rmi localhost:5000/alpine:course          # remove local copy (forces re-pull)
podman pull localhost:5000/alpine:course  # pull an image
```

4) Cleanup:

```
podman rm -f registry  # stop and remove the local registry container
```

Note:

- A local registry is not "secure by default". Treat it as a learning tool.

9.8 Lab: Local Tagging

```
podman pull docker.io/library/nginx:stable  # pull an image
podman tag docker.io/library/nginx:stable localhost/nginx:course  # add another tag/name
podman images | grep nginx  # list images
```

9.9 Lab (Optional): Save and Load

This teaches portable artifacts.

```
podman save -o nginx.tar docker.io/library/nginx:stable  # export an image to a tar file
podman load -i nginx.tar  # import an image from a tar file
```

Note:

- `save/load` are not a registry. They are file-based transport.

9.10 Checkpoint

- You can explain why `:latest` is risky.
- You can find and record an image digest.

9.11 Quick Quiz

1) If you deploy by tag, what can change without you changing your config?

2) What is the advantage of a digest in incident response?

9.12 Further Reading

- OCI image spec (tags vs digests context): https://github.com/opencontainers/image-spec
- `podman-pull(1)`: https://docs.podman.io/en/latest/markdown/podman-pull.1.html
- `podman-image(1)`: https://docs.podman.io/en/latest/markdown/podman-image.1.html
- Registries config (`registries.conf`): https://github.com/containers/image/blob/main/docs/containers-registries.conf.5.md
- Docker Registry HTTP API V2: https://distribution.github.io/distribution/spec/api/

10 Module 4: Secrets (Local-First)

This module replaces the common beginner pattern of putting passwords in `.env` files and environment variables.

10.1 Learning Goals

- Explain why environment variables are not a good secret transport.
- Create and use Podman secrets in rootless mode.
- Mount secrets as files and consume them safely.
- Rotate a secret with minimal downtime.
- Build a "no-secrets-in-logs" habit.

10.2 Mental Model

- A secret is a named blob managed by Podman.
- At runtime, Podman mounts that secret into the container as a file (default: `/run/secrets/<name>`).
- Your app reads the file.

What this helps with:

- Avoids accidental leaks via `env`, process listings, or committing `.env`.
- Keeps secret material out of image layers.

What this does NOT magically solve:

- Distribution of secrets across many hosts.
- Automatic rotation.
- Encryption-at-rest unless you add it externally.

Threat model in one line:

- Podman secrets help with accidental exposure; they do not turn a laptop into a secret manager.

10.3 Common Anti-Patterns

- `export DB_PASSWORD=...` in your shell
- putting passwords in `.env` and committing it
- `podman run -e DB_PASSWORD=...` for anything beyond a throwaway lab
- logging connection strings that contain credentials

10.4 Commands

Create/list/inspect/remove:

```
podman secret create db_password -  # create a secret
podman secret ls  # list secrets
podman secret inspect db_password  # inspect a secret
podman secret rm db_password  # delete the secret by name
```

Create a secret from a file:

```
chmod 600 ./db_password.txt  # change permissions
podman secret create db_password ./db_password.txt  # create a secret
```

Use a secret at runtime:

```
podman run --rm --secret db_password docker.io/library/busybox:latest sh -lc 'test -f /run/secrets/db_pa
```

You can change the target filename inside the container:

```
podman run --rm --secret db_password,target=db.pass docker.io/library/busybox:latest sh -lc 'test -f /r
```

Notes:

- Keep the secret value out of your shell history. Prefer `read -s` or a file with tight permissions.
- Do not print secret contents in logs.

10.5 Lab A: Create and Mount a Secret

1) Create a secret from stdin (example only):

```
printf '%s' 'correct-horse-battery-staple' | podman secret create db_password -  # print text without t
```

2) Run a container that confirms the secret file exists (without printing it):

```
podman run --rm --secret db_password docker.io/library/busybox:latest sh -lc 'test -f /run/secrets/db_pa
```

3) Verify you are not relying on environment variables:

```
podman run --rm --secret db_password docker.io/library/busybox:latest sh -lc 'env | wc -l'  # run a con
```

Checkpoint:

- Secret appears as a file at `/run/secrets/db_password`.
- You never printed the secret value.

10.6 Lab A2: Prove It Is Not an Env Var

This lab builds the "prove it" habit.

1) Start a long-lived container with the secret:

```
podman run -d --name secret-demo --secret db_password docker.io/library/busybox:latest sleep 600  # run
```

2) Check environment does not contain the secret:

```
podman exec secret-demo sh -lc 'env | grep -i password || true'  # run a command in a running container
```

3) Confirm the file exists:

```
podman exec secret-demo sh -lc 'ls -la /run/secrets'  # run a command in a running container
```

4) Cleanup:

```
podman rm -f secret-demo  # stop and remove the demo container
```

10.7 Lab B: Rotation Pattern (Versioned Secrets)

Use versioned names and switch consumers:

```
printf '%s' 'v1-value' | podman secret create db_password_v1 -  # print text without trailing newline
printf '%s' 'v2-value' | podman secret create db_password_v2 -  # print text without trailing newline
```

Run with v1, then update to v2:

```
podman run --rm --secret db_password_v1 docker.io/library/busybox:latest sh -lc 'test -f /run/secrets/d
```

```
podman run --rm --secret db_password_v2 docker.io/library/busybox:latest sh -lc 'test -f /run/secrets/d
```

Guideline:

- Keep the old secret around until the new deployment is verified.
- Remove old secrets only after rollback is no longer needed.

Rotation note:

- Many apps only read secrets on startup. Rotation often means “deploy a new container”.

10.8 Advanced: Build-Time Secrets (Optional)

Goal: authenticate to a private resource during image build without leaking tokens into image layers.

Because Podman/Buildah feature support varies by version, first check your toolchain:

```
podman build --help | sed -n '1,120p'  # build an image
```

If your Podman supports build secrets, prefer:

- Passing a secret to the build command
- Using a secret mount during a build step

Key rule: never use `ARG`/`ENV` for secret material.

If you cannot use build secrets on your version:

- do not work around it by embedding secrets
- fetch private dependencies in CI and copy artifacts into the build context instead

10.9 Common Failure Modes

- Your app expects a string env var but you mounted a file.
- File permissions/ownership do not match what the app runs as.
- You accidentally log the secret during debugging.

10.10 Checkpoint

- You can create and mount a secret without leaking it.
- You can describe a rotation plan that includes rollback.

10.11 Further Reading

- `podman-secret(1)`: https://docs.podman.io/en/latest/markdown/podman-secret.1.html
- OWASP Secrets Management Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/Secrets_Management_Cheat
- systemd credentials (service-provisioned files): https://www.freedesktop.org/software/systemd/man/latest/systemd.exec.
- Kubernetes Secrets (base64 caveat context): https://kubernetes.io/docs/concepts/configuration/secret/

11 Module 5: Storage (Volumes, Bind Mounts, Permissions)

11.1 Learning Goals

- Use volumes for persistent state.
- Use bind mounts for source code and config.
- Debug rootless permission issues (UID/GID mapping).
- Understand SELinux labeling at a high level.
- Know when to use `podman unshare`.

11.2 The Three Storage Types You Will Use

1) Writable layer (per-container, not persistence)
2) Named volume (Podman-managed path, best default for data)
3) Bind mount (you pick the host path)

Practical guidance:

- Use volumes for databases and state.
- Use bind mounts for configuration and source code.
- Avoid writing important data into the writable layer.

11.3 Volumes

Create/list/inspect:

```
podman volume create dbdata  # create a volume
podman volume ls  # list volumes
podman volume inspect dbdata  # inspect a volume
```

Use a volume:

```
podman run --rm -v dbdata:/data docker.io/library/alpine:latest sh -lc 'echo hi > /data/x'  # run a con
podman run --rm -v dbdata:/data docker.io/library/alpine:latest sh -lc 'cat /data/x'  # run a container
```

Clean up (be careful; this deletes data):

```
podman volume rm dbdata  # delete the volume (data loss)
```

11.4 Bind Mounts

Bind mounts are great for:

- configs
- certificates
- local source code in development

Example:

```
mkdir -p ./mnt-demo  # create directory
echo hi > ./mnt-demo/hello.txt  # print text
podman run --rm -v ./mnt-demo:/mnt:Z docker.io/library/alpine:latest cat /mnt/hello.txt  # run a contai
```

Read-only bind mount:

```
podman run --rm -v ./mnt-demo:/mnt:Z,ro docker.io/library/alpine:latest sh -lc 'cat /mnt/hello.txt; echo
```

SELinux note (Fedora/RHEL):

- `:Z` relabels the content for private container use.
- `:z` relabels for shared use.

If SELinux is enforcing and you omit labels, mounts can fail with “permission denied”.

11.5 Inspect Mounts

```
podman run -d --name mount1 -v ./mnt-demo:/mnt:Z docker.io/library/alpine:latest sleep 300  # run a con
podman inspect mount1 --format '{{json .Mounts}}' | less  # inspect container/image metadata
podman rm -f mount1  # cleanup the container
```

11.6 Rootless Permission Pitfalls

Common symptom:

- container process cannot write a mounted directory

Common causes:

- host path is owned by root and not writable by your user
- image runs as a non-root user and needs a matching UID

Debug steps:

```
podman unshare id  # run a command inside the user namespace
podman unshare ls -la <path>  # run a command inside the user namespace
```

Fix patterns:

- If the container runs as your UID, bind mounts are usually easiest.
- If the image runs as a dedicated user, prefer a named volume and let the image initialize ownership.
- If you must, adjust ownership inside the user namespace:

```
podman unshare chown -R 1000:1000 <path>  # run a command inside the user namespace
```

Do not blindly `chmod 777`.

11.7 SELinux Drill (Fedora/RHEL)

If SELinux is Enforcing, try this to learn the failure mode:

1) Attempt a bind mount without labels:

```
podman run --rm -v ./mnt-demo:/mnt docker.io/library/alpine:latest cat /mnt/hello.txt  # run a containe
```

2) If you get permission denied, fix it by adding `:Z`:

```
podman run --rm -v ./mnt-demo:/mnt:Z docker.io/library/alpine:latest cat /mnt/hello.txt  # run a contai
```

Do not disable SELinux as a workaround.

11.8 Lab: Persistent DB Data (Conceptual)

Run a database with a named volume (choose an image you are comfortable with). The key learning is:

- container can be replaced
- data remains

Checklist:

- Create a named volume
- Start the DB with that volume
- Create a table/record
- Remove the container
- Start a new container with the same volume
- Verify the record is still there

Stretch:

- show where the volume lives (`podman volume inspect`)
- document a backup and restore method (logical dump)

11.9 Checkpoint

- You can choose volume vs bind mount intentionally.
- You can explain when `:Z` matters on Fedora/RHEL.
- You can use `podman unshare` to debug permission issues.

11.10 Further Reading

- `podman-volume(1)`: https://docs.podman.io/en/latest/markdown/podman-volume.1.html
- Rootless storage and UID mapping: https://github.com/containers/podman/blob/main/docs/tutorials/rootless_tutorial.
- SELinux mount labeling for containers (RHEL docs): https://docs.redhat.com/en/documentation/red_hat_enterprise_li selinux-with-containers_using-selinux
- `subuid(5)` and `subgid(5)` (man7): https://man7.org/linux/man-pages/man5/subuid.5.html

12 Module 6: Networking (Ports, DNS, User-Defined Networks)

12.1 Learning Goals

By the end of this module you will be able to:

- Explain how rootless networking differs from rootful networking and why it matters.
- Publish container ports to the host and verify them.
- Create, inspect, and remove user-defined networks.
- Connect containers to multiple networks simultaneously.
- Use container DNS names for reliable inter-container service discovery.
- Choose the right network driver (bridge, host, none, macvlan) for a given scenario.
- Configure network-level security: isolate backends, expose only what you need.
- Troubleshoot DNS, port, routing, and firewall problems methodically.
- Connect containers across pods and across user-defined networks.
- Understand how networking interacts with Quadlet (systemd) deployments.

12.2 Minimum Path (If You Are Short on Time)

If you only do a small slice of this module, do these:

- Publish a port and verify with `podman port` (Section 3).
- Create a user-defined network and verify DNS name resolution (Sections 4-5).
- Practice the multi-network isolation pattern (Section 6).
- Run the troubleshooting checklist once (Section 13).

12.3 1 How Container Networking Works (Mental Model)

Before running commands, build the mental model. Every container gets:

1. **A network namespace** — an isolated network stack with its own interfaces, routes, and firewall rules.
2. **A virtual Ethernet pair (veth)** — one end lives inside the container, the other end connects to a virtual bridge (or the host).
3. **An IP address** assigned from the network's subnet.

When two containers are on the **same user-defined network**, the bridge lets them talk to each other by IP. Podman's embedded DNS resolver makes them also reachable **by name**.

When a container is only on the **default network** (Podman's built-in `podman` bridge), DNS-based discovery is disabled. This is a deliberate design choice — it motivates you to create explicit named networks.

12.3.1 1.1 The Four Network Drivers

Driver	What it does	When to use it
`bridge`	Virtual L2 bridge; default for user-defined networks	Almost everything
`host`	Container shares the host network namespace	Low-level tools, benchmarking, rootful only (rootless has caveats)
`none`	No network interface except loopback	Batch jobs, maximum isolation
`macvlan`	Container appears as a separate MAC on your LAN	IoT, legacy apps that need a real LAN address

Rootless note: `host` network mode has limited usefulness in rootless Podman because the container still cannot bind privileged ports without extra capability. `macvlan` requires root on most kernels. Stick to `bridge` unless you have a specific reason.

12.4 2 Rootless Networking In Depth

12.4.1 2.1 User-Mode Networking Helpers

In rootless mode Podman cannot create kernel-level bridges as a normal user. Instead it delegates packet forwarding to a user-space helper:

Helper	Notes
pasta	Newer, faster, preferred on modern distros; fewer quirks with UDP/ICMP
slirp4netns	Older, still common; slower but very portable

Check which backend your installation uses:

```
podman info --format '{{.Host.NetworkBackend}}'  # show Podman host configuration
```

Check which per-network helper is active:

```
podman info --format '{{.Host.Slirp4NetnsOptions}}'  # show Podman host configuration
podman info --format '{{.Host.PastaOptions}}'  # show Podman host configuration
```

You can switch the rootless backend in `~/.config/containers/containers.conf`:

```
[network]
default_rootless_network_cmd = "pasta"
```

12.4.2 2.2 What Rootless Networking Cannot Do (by default)

- Bind ports < 1024 without extra OS configuration.
- Create `macvlan` / `ipvlan` adapters (kernel requires `CAP_NET_ADMIN`).
- Use `host` network mode and see the real host interfaces in the traditional sense.

12.4.3 2.3 Allowing Privileged Ports for Rootless (When Needed)

Option A — lower the unprivileged port minimum (system-wide, only if you own the machine):

```
sudo sysctl -w net.ipv4.ip_unprivileged_port_start=80  # allow low ports for rootless (system-wide)
# make permanent
echo "net.ipv4.ip_unprivileged_port_start=80" | sudo tee /etc/sysctl.d/99-lowport.conf  # persist acros
sudo sysctl -p /etc/sysctl.d/99-lowport.conf  # apply the persistent config
```

Option B — use a high port and put a reverse proxy (nginx, Caddy) in front. Strongly preferred in production.

Option C — use `systemd` socket activation (covered in Module 11).

12.5 3 Port Publishing

12.5.1 3.1 Basic Port Mapping

Syntax: `-p <host-port>:<container-port>`

```
podman run -d --name web1 -p 8080:80 docker.io/library/nginx:stable  # run a container
curl -sS http://127.0.0.1:8080/ | head  # verify HTTP endpoint
```

12.5.2 3.2 Bind to a Specific Host Address

By default `-p 8080:80` listens on all host interfaces (`0.0.0.0`). To restrict to loopback only:

```
podman run -d --name web-lo -p 127.0.0.1:8080:80 docker.io/library/nginx:stable  # run a container
```

To listen on a specific network interface IP:

```
podman run -d --name web-iface -p 192.168.1.100:8080:80 docker.io/library/nginx:stable  # run a contain
```

This is important for security: a backend service should never be published to `0.0.0.0` when it only needs to be reachable by a local proxy.

12.5.3 3.3 Multiple Port Mappings

```
podman run -d --name multi -p 8080:80 -p 8443:443 docker.io/library/nginx:stable  # run a container
```

12.5.4 3.4 UDP Port Mapping

```
podman run -d --name dns-demo -p 5053:53/udp -p 5053:53/tcp docker.io/library/alpine:latest sleep 600
```

12.5.5 3.5 Random Host Port (Ephemeral)

```
podman run -d --name rand-port -p 80 docker.io/library/nginx:stable  # run a container
podman port rand-port          # see what port was assigned
```

12.5.6 3.6 Inspect Published Ports

```
# Quick view
podman port web1  # show published ports

# Full JSON (useful in scripts)
podman inspect web1 --format '{{json .NetworkSettings.Ports}}'  # inspect container/image metadata

# Everything in one JSON dump
podman inspect web1 | python3 -m json.tool | grep -A10 '"Ports"'  # inspect container/image metadata
```

Cleanup:

```
podman rm -f web1 web-lo web-iface multi rand-port  # cleanup containers
```

12.6 4 The Default Network vs User-Defined Networks

12.6.1 4.1 Why the Default Network Is Not Enough

When you run `podman run` without `--network`, the container joins the default `podman` bridge.

Problems with the default network:

1. **No automatic DNS.** Containers cannot find each other by name.
2. **Shared blast radius.** All containers on the default network can reach each other at the IP level.
3. **No isolation.** A compromised container can attempt connections to any other container on the same bridge.

12.6.2 4.2 Creating a User-Defined Network

```
podman network create appnet  # create a network
```

List networks:

```
podman network ls  # list networks
```

Expected output includes your new `appnet` plus built-in networks.

Inspect the network (shows subnet, gateway, driver):

```
podman network inspect appnet  # inspect a network
```

Key fields to understand:

```
"driver": "bridge"
"subnets": [{ "subnet": "10.89.x.0/24", "gateway": "10.89.x.1" }]
"dns_enabled": true
```

Notice `dns_enabled: true` — this is the key difference from the default network.

12.6.3 4.3 Custom Subnet and Gateway

```
podman network create --subnet 172.28.0.0/24 --gateway 172.28.0.1 myapp-net  # create a network
```

Use custom subnets when: - You need deterministic IPs (rare; prefer DNS names instead). - You need to avoid subnet collisions with your VPN or office network.

12.6.4 4.4 Internal Networks (No External Access)

An internal network has no route to the outside world. Containers on it cannot reach the internet.

```
podman network create --internal db-internal  # create a network
```

Use this for databases, caches, and any service that has no business reaching the internet.

Verify:

```
podman run --rm --network db-internal docker.io/library/alpine:latest sh -lc 'wget -qO- --timeout=3 http
```

Expected: connection times out or is refused. That is the intended behavior.

12.6.5 4.5 Remove a Network

```
podman network rm appnet  # remove the network
```

You cannot remove a network that has active containers attached. Stop and remove containers first:

```
podman network rm appnet           # may fail if containers are running
podman ps --filter network=appnet  # find connected containers
podman rm -f $(podman ps -q --filter network=appnet)  # force remove connected containers
podman network rm appnet  # remove a network
```

12.7 5 Container DNS and Service Discovery

12.7.1 5.1 How It Works

Podman runs an embedded DNS resolver (backed by **aardvark-dns** on modern versions). When `dns_enabled: true` on a network:

- Every container on that network is registered with its **container name** and any **network aliases**.
- DNS queries inside containers are answered by the Podman DNS resolver.

- The resolver is reachable at the network gateway address (usually the first usable IP on the subnet).

12.7.2 5.2 Basic DNS Lab

```
podman network create testdns  # create a network

# Start a named container
podman run -d --name server-a --network testdns docker.io/library/alpine:latest sleep 600  # run a cont

# From another container, resolve the name
podman run --rm --network testdns docker.io/library/alpine:latest sh -lc 'getent hosts server-a'  # run
```

Expected output: an IP address followed by `server-a`.

Test TCP connectivity:

```
podman run --rm --network testdns docker.io/library/alpine:latest sh -lc 'nc -zv server-a 80 2>&1 || ec
```

12.7.3 5.3 Network Aliases

An alias lets you give a container an **additional DNS name** on a specific network. This is useful for:

- Running multiple containers that all answer as `db` (blue/green rotation).
- Giving a container a short service name regardless of its actual container name.

```
podman network create alias-demo  # create a network

podman run -d --name primary-db --network alias-demo --network-alias db docker.io/library/alpine:latest

# Resolve by alias
podman run --rm --network alias-demo docker.io/library/alpine:latest sh -lc 'getent hosts db'  # run a
```

Both the container name (`primary-db`) and the alias (`db`) resolve to the same IP.

12.7.4 5.4 Multiple Containers Sharing an Alias (Load-Balancing Pattern)

When multiple containers share the same alias on a network, DNS returns **all IPs** (round-robin).

```
podman network create lb-demo  # create a network

podman run -d --name app-1 --network lb-demo --network-alias app docker.io/library/alpine:latest sleep
podman run -d --name app-2 --network lb-demo --network-alias app docker.io/library/alpine:latest sleep

# Resolve - you may see both IPs
podman run --rm --network lb-demo docker.io/library/alpine:latest sh -lc 'for i in 1 2 3 4; do getent h

# Cleanup
podman rm -f app-1 app-2  # cleanup containers
podman network rm lb-demo # remove the network
```

> This is primitive load balancing. For production you want a real load balancer in front. But the DNS pattern is real.

12.7.5 5.5 Custom DNS Servers

Override the DNS server used inside a container (useful on corporate networks or when using a split-horizon DNS):

```
podman run --rm --dns 1.1.1.1 docker.io/library/alpine:latest sh -lc 'cat /etc/resolv.conf'  # run a co
```

Add DNS search domains:

```
podman run --rm --dns-search corp.example.com docker.io/library/alpine:latest sh -lc 'cat /etc/resolv.c
```

Set a custom `/etc/hosts` entry:

```
podman run --rm --add-host myservice:10.0.1.50 docker.io/library/alpine:latest sh -lc 'getent hosts mys
```

12.8 6 Connecting Containers to Multiple Networks

A container can be a member of more than one network simultaneously. This is the correct way to build a tiered architecture:

`[internet] → [frontend-net] → [app] → [backend-net] → [db]`

- `app` is on both `frontend-net` and `backend-net`.
- `db` is only on `backend-net`.
- `frontend` is only on `frontend-net`.

12.8.1 6.1 Multi-Network Example

```
podman network create frontend-net  # create a network
podman network create backend-net  # create a network

# DB: only on backend
podman run -d --name db --network backend-net docker.io/library/alpine:latest sleep 600  # run a contai

# App: on both networks
podman run -d --name app --network frontend-net docker.io/library/alpine:latest sleep 600  # run a cont

# Connect app to backend AFTER it is running
podman network connect backend-net app  # attach a container to a network

# Frontend: only on frontend
podman run -d --name frontend --network frontend-net docker.io/library/alpine:latest sleep 600  # run a

# Verify: frontend can reach app
podman exec frontend sh -lc 'getent hosts app'  # run a command in a running container

# Verify: frontend CANNOT reach db (different network)
podman exec frontend sh -lc 'getent hosts db || echo "NOT REACHABLE"'  # run a command in a running con

# Verify: app CAN reach db
podman exec app sh -lc 'getent hosts db'  # run a command in a running container

# Cleanup
podman rm -f db app frontend               # stop and remove containers
podman network rm frontend-net backend-net # remove networks
```

12.8.2 6.2 Disconnect from a Network Without Stopping

```
podman network disconnect backend-net app  # detach a container from a network
```

Verify the container no longer has the interface:

```
podman exec app ip addr  # run a command in a running container
```

Reconnect:

```
podman network connect backend-net app  # attach a container to a network
```

12.9 7 Inspecting Network State

12.9.1 7.1 List All Networks

```
podman network ls  # list networks
```

12.9.2 7.2 Detailed Network Info

```
podman network inspect appnet  # inspect a network
```

Shows: driver, subnets, gateways, connected containers, DNS state.

12.9.3 7.3 Which Network Is a Container On?

```
podman inspect <name> --format '{{json .NetworkSettings.Networks}}'  # inspect container/image metadata
```

Or see all networks and their connected containers:

```
podman network inspect appnet --format '{{json .Containers}}'  # inspect a network
```

12.9.4 7.4 Show Container IP Address

```
podman inspect <name> --format '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}'  # inspect co
```

For multi-network containers:

```
podman inspect app --format '{{range $name, $net := .NetworkSettings.Networks}}{{$name}}: {{$net.IPAddr
```

12.9.5 7.5 View Interfaces Inside a Running Container

```
podman exec <name> ip addr  # run a command in a running container
podman exec <name> ip route  # run a command in a running container
podman exec <name> cat /etc/resolv.conf  # run a command in a running container
```

12.9.6 7.6 Host-Side View

On the host, Podman bridge networks appear as `podman` prefixed virtual bridges:

```
ip link show type bridge  # show network links
ip addr show  # show interfaces
```

12.10 8 Network Drivers — Deeper Look

12.10.1 8.1 Bridge (Default)

```
podman network create --driver bridge mybridge  # create a network
```

Characteristics: - Creates a Linux bridge on the host. - Uses NAT (masquerade) for outbound traffic. - Containers get private IPs; host reaches them via the bridge.

12.10.2 8.2 None (No Networking)

```
podman run --rm --network none docker.io/library/alpine:latest ip addr  # run a container
```

Only `lo` (loopback) is present. Useful for: - Batch jobs that need complete network isolation. - Security-sensitive workloads that must never dial out.

12.10.3 8.3 Host (Rootful Only — with Caveats)

```
# Note: limited usefulness in rootless mode
podman run --rm --network host docker.io/library/alpine:latest ip addr  # run a container
```

The container sees the host's network interfaces directly. There is no NAT, no port mapping needed. Avoid this in production rootless workloads.

12.10.4 8.4 macvlan (Requires Root or Capabilities)

```
# rootful or with NET_ADMIN capability only
podman network create --driver macvlan --opt parent=eth0 --subnet 192.168.1.0/24 --gateway 192.168.1.1 m
```

The container appears as a distinct host on your physical LAN. Useful for legacy protocols (DHCP from upstream, mDNS, etc.).

12.11 9 Network Security Patterns

12.11.1 9.1 The Principle: Expose Nothing You Don't Need To

Every port you publish is an attack surface. Every network link you create is a potential pivot point.

Default stance:

- **Databases** → no port published, internal network only.
- **Caches** → no port published, internal network only.
- **APIs** → port published to loopback or internal network, reverse proxy in front.
- **Reverse proxy** → the only container with a public port.

12.11.2 9.2 Segment Networks by Trust Zone

```
[public-net]   web / proxy containers only
[app-net]      app containers + proxy
[db-net]       db containers + app
```

The DB is never on `public-net`. The proxy is never on `db-net`.

12.11.3 9.3 Combine with `--internal` Flag

```
podman network create --internal private-db  # create a network
podman run -d --name postgres --network private-db -e POSTGRES_PASSWORD=secret docker.io/library/postgre
```

This DB can never initiate outbound connections. It cannot call home, exfiltrate data to an external server, or participate in an outbound botnet.

12.11.4 9.4 Use `--network-alias` for Service Contracts

Name your services after their role, not their implementation:

```
--network-alias db         # not "postgres-16-container-prod"
--network-alias cache      # not "redis-7.2"
--network-alias api        # not "my-app-v3"
```

When you upgrade a service, you swap the container and preserve the alias. Nothing else needs to change.

12.11.5 9.9 Avoid Publishing to 0.0.0.0 Unnecessarily

```
# Bad for an internal API
-p 8080:8080              # listens on all interfaces

# Better
-p 127.0.0.1:8080:8080    # loopback only
```

12.12 10 Full Lab: Three-Tier Isolated Stack

Build a realistic, isolated three-tier stack:

- **reverse proxy** (nginx): published to host on 8080, on `frontend-net`
- **app** (alpine with netcat): on `frontend-net` and `app-net`
- **db** (alpine simulating a database): on `app-net` only, no published port

12.12.1 Step 1 — Create Networks

```
podman network create frontend-net  # create a network
podman network create --internal app-net  # create a network
```

12.12.2 Step 2 — Start the "DB"

```
podman run -d --name db --network app-net docker.io/library/alpine:latest sh -lc 'while true; do echo "
```

12.12.3 Step 3 — Start the "App"

```
podman run -d --name app --network app-net --network-alias api docker.io/library/alpine:latest sleep 60
```

Connect app to the frontend network as well:

```
podman network connect frontend-net app  # attach a container to a network
```

12.12.4 Step 4 — Start the Reverse Proxy

```
podman run -d --name proxy --network frontend-net -p 127.0.0.1:8080:80 docker.io/library/nginx:stable
```

12.12.5 Step 5 — Verify Connectivity

App can reach DB:

```
podman exec app sh -lc 'getent hosts db && echo DNS OK'  # run a command in a running container
```

Proxy can reach app:

```
podman exec proxy sh -lc 'getent hosts api && echo DNS OK'  # run a command in a running container
```

Proxy CANNOT reach DB (different network):

```
podman exec proxy sh -lc 'getent hosts db 2>&1 || echo "ISOLATED: expected"'  # run a command in a runn
```

DB CANNOT reach the internet (internal network):

```
podman exec db sh -lc 'wget -qO- --timeout=3 http://example.com 2>&1 || echo "BLOCKED: expected"'  # ru
```

Host can reach proxy via published port:

```
curl -sSI http://127.0.0.1:8080/  # verify HTTP endpoint
```

12.12.6 Step 6 — Cleanup

```
podman rm -f db app proxy              # stop and remove containers
podman network rm frontend-net app-net # remove networks
```

12.13 11 Connecting Containers to Pods on a Network

Pods (covered in Module 7) and user-defined networks interact naturally. You can place an entire pod on a named network:

```
podman network create podnet  # create a network

podman pod create --name mypod --network podnet -p 8090:80  # create a pod

podman run -d --pod mypod --name pod-nginx docker.io/library/nginx:stable  # run a container

# A container outside the pod resolves the pod by its infra container's IP
# or by any container name inside:
podman run --rm --network podnet docker.io/library/alpine:latest sh -lc 'getent hosts pod-nginx'  # run

podman pod rm -f mypod  # stop and remove the pod and its containers
podman network rm podnet  # remove the network
```

12.14 12 Networking in Quadlet (systemd) Deployments

Quadlet `.network` unit files let you declare Podman networks as systemd-managed resources. This ensures networks exist before containers start.

12.14.1 12.1 Declare a Network Unit

Create `~/.config/containers/systemd/appnet.network`:

```
[Unit]
Description=Application private network

[Network]
Driver=bridge
Internal=true
```

12.14.2 12.2 Reference the Network in a Container Unit

In your `.container` unit file:

```
[Container]
Image=docker.io/library/nginx:stable
Network=appnet.network
```

Systemd will automatically create `appnet` before starting your container and the dependency chain is managed for you.

Full Quadlet networking is covered in Module 11.

12.15 13 Troubleshooting Networking

12.15.1 13.1 Symptom: Container Cannot Reach Another Container by Name

Checklist:

1. Are both containers on the **same user-defined network**? (Not the default `podman` network.)
2. Is `dns_enabled: true` on that network?
3. Are both containers **running** (not exited)?
4. Are you using the **container name** (or an alias), not the hostname?

```
# Check network membership
podman inspect <name> --format '{{json .NetworkSettings.Networks}}'  # inspect container/image metadata

# Verify DNS is enabled on the network
podman network inspect <net> --format '{{.DNSEnabled}}'  # inspect a network

# Try a live DNS lookup from a debug container
podman run --rm --network <net> docker.io/library/alpine:latest sh -lc 'getent hosts <target-name>'  #
```

12.15.2 13.2 Symptom: Cannot Connect Even Though DNS Resolves

DNS working but TCP failing means the service is not listening, is on the wrong port, or there is a firewall rule.

```
# Check if the port is open
podman run --rm --network <net> docker.io/library/alpine:latest sh -lc 'nc -zv <target> <port>'  # run

# Check what the container is actually listening on
podman exec <target> ss -tlnp  # run a command in a running container
# or
podman exec <target> netstat -tlnp  # run a command in a running container
```

12.15.3 13.3 Symptom: Port Published But Cannot Reach from Host

```
# Confirm the port mapping
podman port <name>  # show published ports

# Confirm the process is listening inside the container
podman exec <name> ss -tlnp  # run a command in a running container

# Check host firewall
sudo firewall-cmd --list-all   # firewalld
sudo iptables -L -n            # iptables / nftables

# Check the container's host binding
podman inspect <name> --format '{{json .NetworkSettings.Ports}}'  # inspect container/image metadata
# Look for "HostIp" - if it's 127.0.0.1, you can only reach from localhost
```

12.15.4 13.4 Symptom: `nc` or `wget` Not Available in Container

Use a debug sidecar with networking tools:

```
podman run --rm --network <net> docker.io/library/nicolaka/netshoot:latest curl -v http://<target>:<por
```

Or use a minimal alpine with a one-liner install:

```
podman run --rm --network <net> docker.io/library/alpine:latest sh -lc 'apk add -q curl && curl -v http
```

12.15.5 13.5 Symptom: Container Cannot Reach the Internet

```
# Verify DNS
podman exec <name> sh -lc 'cat /etc/resolv.conf'  # run a command in a running container

# Try pinging a well-known IP (not DNS-dependent)
podman exec <name> ping -c3 8.8.8.8  # run a command in a running container

# Try DNS resolution
podman exec <name> sh -lc 'getent hosts example.com'  # run a command in a running container

# Check if the network is internal
podman network inspect <net> --format '{{.Internal}}'  # inspect a network
```

If the network is `internal: true`, outbound traffic is intentionally blocked.

If DNS fails but the IP works, the problem is your DNS resolver configuration.

12.15.6 13.6 Symptom: Sporadic Connection Failures (Rootless)

This is often a pasta/slirp4netns quirk with UDP under high load, or a port exhaustion issue.

```
# Check for errors in the rootless network helper
journalctl --user -u podman.socket  # view user-service logs
podman events --filter type=network  # show Podman lifecycle events
```

12.15.7 13.7 Useful Debugging One-Liners

```
# All running container IPs
podman ps -q | xargs -I{} podman inspect {} --format '{{.Name}}: {{range .NetworkSettings.Networks}}{{.

# All networks and their subnets
podman network ls -q | xargs -I{} podman network inspect {} --format '{{.Name}}: {{range .Subnets}}{{.Su

# Which containers are on a given network
podman network inspect <net> --format '{{range $id, $c := .Containers}}{{$c.Name}} {{end}}'  # inspect

# Container's effective DNS config
podman exec <name> cat /etc/resolv.conf  # run a command in a running container
```

12.16 14 Common Patterns Reference

12.16.1 Pattern A — Single Shared App Network (Simple Stack)

```
podman network create app  # create a network
podman run -d --name db    --network app docker.io/library/postgres:16-alpine  # run a container
podman run -d --name cache --network app docker.io/library/redis:7-alpine  # run a container
podman run -d --name api   --network app -p 127.0.0.1:8000:8000 myapp:latest  # run a container
podman run -d --name proxy --network app -p 0.0.0.0:80:80     nginx:stable  # run a container
```

12.16.2 Pattern B — Segmented Networks (Recommended for Production)

```
podman network create --internal data-tier  # create a network
podman network create app-tier  # create a network
podman network create public-tier  # create a network
```

```
podman run -d --name db     --network data-tier   postgres:16-alpine  # run a container
podman run -d --name cache  --network data-tier   redis:7-alpine  # run a container
podman run -d --name api    --network app-tier    myapp:latest  # run a container
podman network connect data-tier api              # api reaches db and cache

podman run -d --name proxy  --network public-tier -p 80:80 nginx:stable  # run a container
podman network connect app-tier proxy             # proxy reaches api
```

12.16.3 Pattern C — Debug Sidecar (Ephemeral)

```
podman run --rm -it --network <same-net> docker.io/library/alpine:latest sh  # run a container
# Now you have a shell inside the network with tools
```

12.16.4 Pattern D — One-Time Migration Container

```
podman run --rm --network app-tier --env-file .env myapp:latest ./migrate.sh  # run a container
```

12.17 Checkpoint

You should be able to answer the following without looking at commands:

- What is the fundamental difference between the default `podman` network and a user-defined network?
- Why does `--internal` improve security for a database network?
- How do containers find each other by name (what Podman feature enables this)?
- When would you use a network alias instead of a container name?
- What flag restricts a port binding to loopback only?
- How do you connect a running container to a second network without restarting it?
- What is the first tool you reach for when a container cannot be found by DNS?

12.18 Quick Quiz

1. You have two containers on the default `podman` network. Container A tries to `curl http://container-b/`. It fails with "could not resolve host". What is the most likely cause and fix?
2. You start a database with `-p 5432:5432`. A security reviewer flags this as a problem. Why, and how do you fix it?
3. You have an `app` container that needs to talk to both a `frontend` network and a `backend` network. What is the cleanest way to set this up, and what command do you use to add the second network connection after the container is running?
4. Two containers share the network alias `api`. A third container queries `getent hosts api`. What does it get back? What does this enable?
5. A container can ping `8.8.8.8` but cannot resolve `example.com`. What is the most likely problem?
6. You check `podman port mycontainer` and it shows port 8080 mapped. But `curl http://127.0.0.1:8080/` fails. Name three things to check next.

12.19 Further Reading

- Podman networking documentation: `man podman-network`
- aardvark-dns project (embedded DNS): https://github.com/containers/aardvark-dns

- pasta (rootless networking helper): https://passt.top/
- slirp4netns: https://github.com/rootless-containers/slirp4netns
- CNI vs Netavark: https://podman.io/blogs/2022/05/05/podman-rootful-rootless.html
- nftables and container networking: see Module 13 (Troubleshooting)

13 Module 7: Pods and Sidecars

Podman pods group multiple containers so they share certain namespaces (especially networking).

13.1 Learning Goals

- Explain what a Podman pod is and when to use it.
- Run a pod with multiple containers.
- Understand the "sidecar" pattern.
- Know what the infra container is.

13.2 Why Pods

Pods are useful when:

- multiple containers need to share localhost networking
- you want a unit of deployment smaller than a full orchestrator

They are not required for most workloads. Use them when they simplify your stack.

13.3 Infra Container

Pods include an "infra" container that holds the shared namespaces.

You will often see it in `podman ps` for a pod.

13.4 Lab: Two Containers, One Pod

1) Create a pod and publish a port:

```
podman pod create --name webpod -p 8080:80  # create a pod
```

List pods:

```
podman pod ps  # list pods
podman ps --pod  # list containers
```

2) Run an HTTP server inside the pod:

```
podman run -d --pod webpod --name nginx docker.io/library/nginx:stable  # run a container
```

3) Add a "debug sidecar" container in the same pod:

```
podman run -it --rm --pod webpod docker.io/library/alpine:latest sh  # run a container
```

Inside the sidecar, verify you can reach nginx on localhost:

```
wget -qO- http://127.0.0.1:80/ | head  # fetch a URL
exit  # exit the shell
```

4) Cleanup:

```
podman pod rm -f webpod  # stop and remove the pod and its containers
```

13.5 Pod Lifecycle Notes

- Stopping a pod stops all containers.
- Removing a pod removes the infra container too.

13.6 Checkpoint

- You can use a sidecar for debugging without exposing new ports.
- You understand pods are not Kubernetes, but the idea rhymes.

13.7 Quick Quiz

1) Why does a pod show an extra container?

2) When would you prefer a network over a pod?

13.8 Further Reading

- `podman-pod(1)`: https://docs.podman.io/en/latest/markdown/podman-pod.1.html
- Kubernetes Pods concept: https://kubernetes.io/docs/concepts/workloads/pods/
- Sidecar containers (Kubernetes): https://kubernetes.io/docs/concepts/workloads/pods/sidecar-containers/

14 Module 8: Building Images (Containerfiles)

This module teaches you how to build images you can trust in production:

- reproducible enough to debug
- small enough to ship
- least-privilege by default
- free of secrets

The focus is Podman-first: `podman build`, `podman image`, `podman push`.

14.1 Learning Goals

By the end of this module you will be able to:

- Explain what an image layer is and why instruction order matters.
- Write a Containerfile that is secure-by-default (non-root, minimal writes).
- Build, tag, inspect, run, and push images with Podman.
- Use multi-stage builds and `--target` to keep runtime images small.
- Keep secrets out of build layers and out of the build context.
- Debug common build and runtime failures (missing files, permissions, wrong arch).

14.2 Minimum Path (If You Are Short on Time)

- Do Lab A (build + run a tiny image).
- Add a `.containerignore` and switch from `COPY . .` to explicit copies.
- Do the non-root lab (Lab B).
- Build a multi-stage image once (Go example or `examples/build/hello-bun`).

14.3 1 Images, Layers, and the Build Mental Model

An image build is a series of filesystem snapshots.

- Each instruction in a `Containerfile` creates a new **layer**.
- A layer is immutable; rebuilding changes layers above the first changed step.
- `COPY` and `RUN` are the two instructions that most often invalidate caching.

Practical implications:

- Put slow-changing steps early (base image selection, OS packages).
- Put fast-changing steps late (your app source code).
- Keep the build context small so `COPY` does not force expensive rebuilds.

Terminology:

- **Containerfile**: the build recipe (Dockerfile-compatible syntax).
- **Build context**: the directory sent to the builder for `COPY`/`ADD`.
- **Tag**: a mutable pointer (`myapp:1`, `myapp:latest`).
- **Digest**: immutable content address (`sha256:...`).

14.4 2 `podman build` Fundamentals

14.4.1 2.1 Basic Build

```
podman build -t localhost/myapp:1 .  # build an image
```

Common flags:

```
podman build -t localhost/myapp:1 -f Containerfile .  # build an image
podman build --pull=always -t localhost/myapp:1 .  # build an image
podman build --no-cache -t localhost/myapp:1 .  # build an image
podman build --layers -t localhost/myapp:1 .  # build an image
podman build --target runtime -t localhost/myapp:1 .  # build an image
```

Notes:

- `--pull=always` is useful in CI to ensure you build from the newest base.
- `--no-cache` is useful when debugging, but do not make it your default.
- `--target` builds only a named stage from a multi-stage Containerfile.

14.4.2 2.2 Naming: Why `localhost/` Is Used in Labs

Using `localhost/<name>` makes it explicit that the tag is local and not in a remote registry namespace.

```
podman images | head  # list images
```

You will see `localhost/myapp:1` locally even if you are not logged into a registry.

14.4.3 2.3 What Builds What

Podman builds are typically executed by Buildah under the hood.

You do not need to become a Buildah expert, but this matters when you search for docs and flags.

14.5 3 Containerfile Instructions: The Practical Subset

You can build most real images with these instructions:

- `FROM` choose base image (pin versions; consider digest pinning for prod)
- `WORKDIR` avoid brittle `cd` chains
- `COPY` bring in files (prefer explicit paths)
- `RUN` install/build steps
- `ENV` defaults (not secrets)
- `USER` run as non-root
- `EXPOSE` document ports (does not publish)
- `CMD` default command
- `ENTRYPOINT` fixed entry wrapper (use sparingly)
- `LABEL` attach metadata
- `HEALTHCHECK` basic liveness signal (optional)

14.5.1 3.1 `COPY` vs `ADD`

Rule of thumb:

- Use `COPY` almost always.
- Use `ADD` only when you specifically need its extra behaviors (such as extracting a local tar archive).

Do not use `ADD` to fetch URLs.

14.5.2 3.2 Shell Form vs Exec Form

Exec form (recommended for servers):

```
CMD ["nginx", "-g", "daemon off;"]
```

Shell form:

```
CMD nginx -g 'daemon off;'
```

Why exec form is better:

- Signal handling works as expected.
- Your process becomes PID 1 (no intermediate shell).
- Arguments are not re-parsed by a shell.

14.5.3 3.3 `ENTRYPOINT` vs `CMD`

- `CMD` is the default that users commonly override.
- `ENTRYPOINT` is for the command you almost never want overridden.

If you use both:

```
ENTRYPOINT ["/app/entrypoint.sh"]
CMD ["/app/server"]
```

14.5.4 3.4 `EXPOSE` Does Not Publish Ports

`EXPOSE 8080` is documentation inside the image.

Publishing is runtime:

```
podman run -p 8080:8080 localhost/myapp:1  # run a container
```

14.6 4 Lab A: Build a Tiny HTTP Image (Warm-Up)

Create a new directory:

```
mkdir -p ./image-lab  # create directory
cd ./image-lab  # change directory
```

Create `index.html`:

```
printf '%s\n' '<h1>Hello from Podman</h1>' > index.html  # print text without trailing newline
```

Create `Containerfile`:

```
FROM docker.io/library/nginx:stable
COPY index.html /usr/share/nginx/html/index.html
```

Build and run:

```
podman build -t localhost/hello-nginx:1 .  # build an image
podman run --rm -p 8080:80 localhost/hello-nginx:1  # run a container
```

Verify in another terminal:

```
curl -sS http://127.0.0.1:8080/  # verify HTTP endpoint
```

Inspect the image:

```
podman image inspect localhost/hello-nginx:1 | less  # inspect image metadata
podman image history localhost/hello-nginx:1  # show image layer history
```

Cleanup:

```
podman rmi localhost/hello-nginx:1  # remove the image from local storage
cd ..  # change directory
rm -rf ./image-lab                  # delete the lab directory
```

14.7 5 Build Context Hygiene (The Most Common Image Leak)

Your build context is everything in the directory you pass to `podman build`.

If your directory contains:

- `.env`
- SSH keys
- kubeconfigs
- `node_modules`
- build artifacts

. . . then a sloppy `COPY . .` can accidentally ship them inside your image.

14.7.1 5.1 Use `.containerignore`

Create `.containerignore` next to your `Containerfile`:

```
.git
.github
.env
*.key
*.pem
node_modules
dist
build
tmp
*.log
```

Podman commonly supports `.containerignore` and often also `.dockerignore`.

14.7.2 5.2 Prefer Explicit Copies

Instead of:

```
COPY . /app
```

Prefer:

```
COPY package.json bun.lockb /app/
COPY src/ /app/src/
```

This prevents accidental inclusion and improves caching.

14.8 6 Running as Non-Root (Image-Level Least Privilege)

Rootless Podman protects the host.

Running as non-root inside the container protects you from:

- container escape bugs that still require privileges inside the container
- accidental writes to system locations in the image
- overly-permissive defaults (root can write almost anywhere)

14.8.1 6.1 The Three Places You Usually Need Write Access

- `/tmp`
- an app state directory (like `/var/lib/myapp`)
- log directory (often better to log to stdout/stderr)

Best practice:

- treat the root filesystem as read-only when possible (Module 12)
- use a dedicated volume or tmpfs for the few paths that must be writable

14.8.2 6.2 Pattern: Create a User and Own the App Directory

```
FROM docker.io/library/alpine:3.20

RUN addgroup -S app && adduser -S -G app app
WORKDIR /app
COPY --chown=app:app . /app
USER app
CMD ["sh", "-lc", "id && ls -la"]
```

Notes:

- `COPY --chown=...` is often cleaner than `RUN chown -R ...`.
- Some minimal images do not include `adduser`/`addgroup` (use their native tools).

14.8.3 6.3 Lab B: Verify Non-Root Actually Works

```
mkdir -p ./nonroot-lab  # create directory
cd ./nonroot-lab  # change directory

cat > Containerfile <<'EOF'
FROM docker.io/library/alpine:3.20

RUN addgroup -S app && adduser -S -G app app
WORKDIR /app
COPY --chown=app:app . /app
USER app
CMD ["sh", "-lc", "echo user=$(id -u) group=$(id -g); touch /app/ok; ls -la /app"]
EOF

podman build -t localhost/nonroot:1 .  # build an image
podman run --rm localhost/nonroot:1  # run a container
podman rmi localhost/nonroot:1  # remove the image from local storage
cd ..  # change directory
rm -rf ./nonroot-lab            # delete the lab directory
```

If `touch /app/ok` fails, you did not set ownership correctly.

14.9 7 Multi-Stage Builds (Small Images, Fast Builds)

Multi-stage builds let you:

- compile/build in a stage that has toolchains
- copy only the runtime artifacts into a final minimal image

Key properties:

- stages have names: `FROM ... AS build`
- later stages can `COPY --from=build ...`
- `podman build --target <stage>` stops early (useful for debugging)

14.9.1 7.1 Lab C (Optional): Provided Go Multi-Stage Example

This repository includes:

- `examples/build/hello-go/Containerfile`
- `examples/build/hello-go/main.go`

Build and run:

```
podman build -t localhost/hello-go:1 examples/build/hello-go  # build an image
podman run --rm -p 8085:8080 localhost/hello-go:1  # run a container
curl -sS http://127.0.0.1:8085/  # verify HTTP endpoint
```

Inspect size:

```
podman images | head  # list images
podman image history localhost/hello-go:1  # show image layer history
```

14.9.2 7.2 Pattern: Build Dependencies First, Copy Source Later

This pattern maximizes cache reuse:

1. copy dependency manifests
2. install dependencies
3. copy application source
4. build

Even if you do not use multi-stage, the order still matters.

14.9.3 7.3 Example Pattern: Bun App (Build + Runtime)

This is an example Containerfile shape for Bun-based services. Adapt it to your project.

Try the repo-backed example:

- `examples/build/hello-bun/Containerfile`
- `examples/build/hello-bun/server.ts`

```
FROM docker.io/oven/bun:1.2.0 AS build
WORKDIR /app

# Install deps separately for caching
COPY package.json bun.lockb ./
RUN bun install --frozen-lockfile

COPY . ./
RUN bun run build

FROM docker.io/oven/bun:1.2.0 AS runtime
WORKDIR /app
ENV NODE_ENV=production

# Copy only the runtime artifacts you need
COPY --from=build /app/dist ./dist
COPY --from=build /app/package.json ./package.json
```

```
USER bun
EXPOSE 3000
CMD ["bun", "dist/server.js"]
```

Notes:

- If your build outputs different paths, adjust `COPY --from=build`.
- If you need native modules, your runtime base must be compatible.

14.9.4 7.4 Example Pattern: Static Web Build (Build Stage + nginx)

```
FROM docker.io/library/node:22-alpine AS build
WORKDIR /src
COPY package.json package-lock.json ./
RUN npm ci
COPY . ./
RUN npm run build

FROM docker.io/library/nginx:stable
COPY --from=build /src/dist/ /usr/share/nginx/html/
```

This keeps Node and build tools out of the runtime image.

14.10 8 Caching: Make Rebuilds Fast

Most slow builds are slow because caching is accidentally disabled.

14.10.1 8.1 Common Cache-Busters

- `COPY . .` early in the file
- including `node_modules/` or `target/` in the context
- running `apt-get update` in a separate layer from `apt-get install`
- using floating package versions

14.10.2 8.2 Linux Packages: One Layer, Clean Up

For Debian/Ubuntu bases:

```
RUN apt-get update \
  && apt-get install -y --no-install-recommends ca-certificates curl \
  && rm -rf /var/lib/apt/lists/*
```

For Alpine:

```
RUN apk add --no-cache ca-certificates curl
```

14.10.3 8.3 Use Stage Targets for Faster Debugging

If a multi-stage build fails late, rebuild only to the stage you care about:

```
podman build --target build -t localhost/myapp:build .  # build an image
```

Then you can run that stage as an image to inspect the filesystem:

```
podman run --rm -it localhost/myapp:build sh  # run a container
```

14.11 9 ARG, ENV, and Configuration

14.11.1 9.1 ARG Is Build-Time

ARG values exist during build, and can influence caching.

```
ARG APP_VERSION
LABEL org.opencontainers.image.version=$APP_VERSION
```

Build:

```
podman build --build-arg APP_VERSION=1.2.3 -t localhost/myapp:1 .  # build an image
```

14.11.2 9.2 ENV Is Runtime Default

```
ENV PORT=3000
EXPOSE 3000
CMD ["/app/server"]
```

Override at runtime:

```
podman run --rm -e PORT=8080 localhost/myapp:1  # run a container
```

14.11.3 9.3 Do Not Put Secrets in ARG or ENV

If you do this:

```
ARG API_KEY
RUN curl -H "Authorization: Bearer $API_KEY" ...
```

You risk leaking secrets into:

- image history
- build logs
- intermediate layers

Use runtime secrets (Module 4) or build-time secret mechanisms (next section).

14.12 10 Secrets and Private Dependencies (Build-Time)

Rules you can rely on:

- never COPY secret files into the image
- never commit secrets into the build context
- prefer fetching private dependencies outside the build and copying only artifacts

14.12.1 10.1 If Your Podman Supports Build Secrets

Some Podman/Buildah versions support `podman build --secret ...`.

Example shape (do not assume your version supports it):

```
podman build --secret id=npmrc,src=$HOME/.npmrc -t localhost/private-build:1 .  # build an image
```

In a Containerfile, the secret is mounted at build time (not copied into layers).

If your version does not support it, use the safe fallback below.

14.12.2 10.2 Safe Fallback: Fetch in CI, Copy Artifacts

Instead of cloning or downloading private content during image build:

1. fetch private deps in CI with credentials
2. produce a build artifact (binary, bundle, wheel)
3. copy only the artifact into the build context
4. build a runtime image that contains only that artifact

This keeps secrets entirely out of the image build process.

14.13 11 Labels, Metadata, and Image Introspection

Labels help you operate images later.

Recommended OCI labels:

- `org.opencontainers.image.title`
- `org.opencontainers.image.description`
- `org.opencontainers.image.source`
- `org.opencontainers.image.revision`
- `org.opencontainers.image.version`
- `org.opencontainers.image.licenses`

Example:

```
LABEL org.opencontainers.image.title="hello"
LABEL org.opencontainers.image.source="https://example.com/repo"
LABEL org.opencontainers.image.version="1.0.0"
```

Inspect labels:

```
podman image inspect localhost/myapp:1 --format '{{json .Labels}}'  # inspect image metadata
```

14.14 12 Tagging, Digests, and Promotion

14.14.1 12.1 Tags Are Mutable

`myapp:latest` can point to different content over time.

This is convenient, but it is not auditable.

14.14.2 12.2 Digests Are Immutable

Pull and run by digest:

```
podman pull docker.io/library/nginx@sha256:<digest>  # pull an image
podman run --rm docker.io/library/nginx@sha256:<digest>  # run a container
```

For production:

- build from pinned bases when you need repeatability
- promote images by digest (not by tag) when you need audit trails

14.14.3 12.3 A Simple Promotion Flow

1. build locally or in CI as `myapp:git-<sha>`
2. run tests
3. re-tag as `myapp:staging`

4. re-tag as `myapp:prod`

Commands:

```
podman tag localhost/myapp:git-abc123 localhost/myapp:staging  # add another tag/name
podman tag localhost/myapp:git-abc123 localhost/myapp:prod  # add another tag/name
```

14.15 13 Pushing Images to a Registry

14.15.1 13.1 Login

```
podman login <registry>  # log into a container registry
```

14.15.2 13.2 Tag for the Registry Namespace

```
podman tag localhost/myapp:1 registry.example.com/team/myapp:1  # add another tag/name
```

14.15.3 13.3 Push

```
podman push registry.example.com/team/myapp:1  # push an image to a registry
```

14.15.4 13.4 Pull and Verify

```
podman pull registry.example.com/team/myapp:1  # pull an image
podman image inspect registry.example.com/team/myapp:1 | less  # inspect image metadata
```

Production habit:

- record the digest you deployed
- configure your runtime (Quadlet, Kubernetes YAML) to pull by digest

---

14.16 14 Testing the Image You Built

Your build is not done when `podman build` finishes.

Minimum checks:

1. container starts
2. correct ports are exposed/published
3. process responds to a health endpoint
4. container runs as non-root (if intended)
5. container writes only to intended paths

14.16.1 14.1 Smoke Test

```
podman run --rm -p 8080:8080 localhost/myapp:1  # run a container
```

14.16.2 14.2 Confirm Effective User

```
podman run --rm localhost/myapp:1 id  # run a container
```

14.16.3 14.3 Healthcheck (If You Define One)

If your Containerfile includes `HEALTHCHECK`:

- You may see warnings that `HEALTHCHECK` is ignored in OCI image format.
- If you need the healthcheck stored in the image, build using docker format:

```
podman build --format docker -t localhost/myapp:1 .  # build an image
```

Alternative: define a healthcheck at runtime with `podman run --health-*` flags.

```
podman run -d --name hc localhost/myapp:1  # run a container
podman healthcheck run hc  # run the container healthcheck
podman inspect hc --format '{{json .State.Health}}'  # inspect container/image metadata
podman rm -f hc  # stop and remove the container
```

14.17 15 Troubleshooting Builds (Common Failures)

14.17.1 15.1 `COPY failed: file not found in build context`

Causes:

- wrong path (relative paths are relative to the build context)
- file excluded by `.containerignore`
- building from the wrong directory

Fix:

- confirm your build context: `podman build ... <context-dir>`
- list files in the context dir

14.17.2 15.2 Permission Errors in `RUN` Steps

Typical in rootless builds when scripts assume root-only locations.

Fix patterns:

- write into `$HOME` or your work directory, not `/root`
- ensure `WORKDIR` exists
- if you switch to `USER app`, do it after you finish root-only install steps

14.17.3 15.3 Container Starts Then Exits Immediately

Causes:

- `CMD` is missing or wrong
- app binary not executable
- wrong working directory

Debug:

```
podman run --rm -it --entrypoint sh localhost/myapp:1  # run a container
```

14.17.4 15.4 `exec format error`

Cause:

- architecture mismatch (built for amd64, running on arm64, or vice versa)

Fix:

- build for the target platform (if your environment supports it):

```
podman build --platform linux/amd64 -t localhost/myapp:amd64 .  # build an image
```

Cross-building often requires extra host setup (emulation). Treat it as an advanced topic.

14.17.5 15.5 Huge Images

Causes:

- build tools included in the runtime stage
- caches kept (package manager caches, build artifacts)
- copying your entire repo including junk

Fix:

- multi-stage builds
- `.containerignore`
- copy only artifacts into the final stage

Inspect what grew:

```
podman image history localhost/myapp:1  # show image layer history
```

14.18 16 Cleanup: Keep Your Machine Healthy

Image builds create intermediate images and caches.

Useful commands:

```
podman images  # list images
podman ps -a  # list containers

podman image prune        # remove unused images (frees disk)
podman container prune    # remove stopped containers
podman system prune       # remove unused objects (be careful)

# builder-specific cache (if supported)
podman builder prune      # remove build cache (if supported)
```

Be careful:

- `podman system prune` can delete data you care about if you store it in unnamed volumes.

14.19 17 Extended Lab: A Small "Real" Service Image

This lab builds a service image with:

- non-root runtime
- explicit copies
- reasonable labels
- a predictable port

It uses only shell + Python standard library so you do not need extra tooling.

14.19.1 17.1 Create a Small App

```
mkdir -p ./svc-lab  # create directory
cd ./svc-lab  # change directory
```

Create `server.py`:

```
cat > server.py <<'EOF'
import os
```

```
from http.server import BaseHTTPRequestHandler, HTTPServer

PORT = int(os.environ.get("PORT", "8080"))

class Handler(BaseHTTPRequestHandler):
    def do_GET(self):
        self.send_response(200)
        self.send_header("Content-Type", "text/plain; charset=utf-8")
        self.end_headers()
        self.wfile.write(b"ok\n")

HTTPServer(("0.0.0.0", PORT), Handler).serve_forever()
EOF
```

Create a `.containerignore`:

```
cat > .containerignore <<'EOF'
.git
.env
*.log
__pycache__
EOF
```

Create `Containerfile`:

```
cat > Containerfile <<'EOF'
FROM docker.io/library/python:3.13-alpine

LABEL org.opencontainers.image.title="svc-lab"
LABEL org.opencontainers.image.description="tiny HTTP server lab"

RUN addgroup -S app && adduser -S -G app app

WORKDIR /app
COPY --chown=app:app server.py /app/server.py

ENV PORT=8080
USER app
EXPOSE 8080

CMD ["python", "/app/server.py"]
EOF
```

Build and run:

```
podman build -t localhost/svc-lab:1 .  # build an image
podman run --rm -p 8088:8080 localhost/svc-lab:1  # run a container
```

Verify:

```
curl -sS http://127.0.0.1:8088/  # verify HTTP endpoint
```

Cleanup:

```
podman rmi localhost/svc-lab:1  # remove the image from local storage
cd ..  # change directory
rm -rf ./svc-lab               # delete the lab directory
```

14.20 Checkpoint

- You can explain what a build context is and why `.containerignore` matters.
- You can write a Containerfile that runs as non-root.
- You can explain why multi-stage builds keep runtime images small.
- You can inspect image history to understand what changed.
- You can explain why secrets do not belong in `ARG`, `ENV`, or `COPY`.

14.21 Quick Quiz

1) Your build works, but rebuilds are always slow even when nothing changes. What Containerfile patterns usually cause cache misses?

2) Why is `COPY . .` risky even when you think your repo contains no secrets?

3) You want to build a Go binary and ship it in a small runtime image. What key multi-stage instruction copies the artifact into the final image?

4) You set `EXPOSE 8080` in the image but the service is not reachable from the host. What did you forget?

5) An auditor asks you to prove exactly which base image you shipped. Why does pinning by digest help?

14.22 Further Reading

- `man podman-build`
- `man podman-image`
- `man Containerfile` (often via Buildah docs)
- OCI image spec labels: https://github.com/opencontainers/image-spec/blob/main/annotations.md

15 Module 9: Multi-Service Workflows

This module teaches patterns for running a small stack without jumping straight to a full orchestrator.

15.1 Learning Goals

- Choose between: separate containers, a user-defined network, or a pod.
- Build a repeatable "stack up / stack down" workflow.
- Know the tradeoffs of compose-style tooling.
- Know how to persist data and keep DB private.

15.2 Patterns

1) User-defined network + multiple containers

- Good default for small stacks.
- Clear service discovery (names on the network).

2) Pod (shared localhost)

- Great when sidecars need to hit `127.0.0.1`.

3) Kube YAML locally (`podman play kube`)

- Good when you want the YAML shape.
- Not the full Kubernetes API.

15.3 Lab: A Two-Service Stack (Network + Volumes)

Goal:

- DB: private, persistent volume
- App: published port

Checklist:

- Create network `appnet`
- Create volume `dbdata`
- Start DB on `appnet` with `dbdata`
- Start web app on `appnet` and publish 8080
- Confirm web app can resolve DB by container name

Suggested validation steps:

- `podman ps` shows both containers
- `podman network inspect appnet` shows both attached
- DB has no published ports (`podman port <db>` shows nothing)

15.4 Make It Repeatable (Script)

Use the provided example:

- `examples/stack/stack.sh`

It demonstrates:

- idempotent create of network/volume
- start/stop with stable names
- separate concerns: infra objects vs containers

Notes:

- The script uses Podman secrets and prompts for a DB password if missing.
- Treat it as a learning tool, not a production deploy mechanism.

Run:

```
bash examples/stack/stack.sh up  # run a shell script
bash examples/stack/stack.sh status  # run a shell script
bash examples/stack/stack.sh down  # run a shell script
```

15.5 Compose-ish Tooling (Context)

If your team already uses compose files, you may encounter helper tools.

Key rule for this course:

- Learn the primitives first (networks, volumes, pods, Quadlet).

If your org already standardized on compose files:

- treat compose as a convenient wrapper
- still learn how it maps to networks/volumes/secrets so you can debug it

15.6 Checkpoint

- You can stand up and tear down a small stack predictably.
- You can explain when you would reach for `podman play kube`.

15.7 Quick Quiz

1) Why should your DB usually not publish a port to the host?
2) What makes a stack script "safe" to run repeatedly?

15.8 Further Reading

- `podman-network(1)`: https://docs.podman.io/en/latest/markdown/podman-network.1.html
- `podman-pod(1)`: https://docs.podman.io/en/latest/markdown/podman-pod.1.html
- `podman-play-kube(1)`: https://docs.podman.io/en/latest/markdown/podman-play-kube.1.html
- Compose Specification (for mapping concepts): https://compose-spec.io/

16 Module 10: `podman play kube`

`podman play kube` runs a subset of Kubernetes YAML locally.

16.1 Learning Goals

- Understand what `play kube` can and cannot do.
- Run a pod from YAML.
- Use YAML as a repeatable definition of a local stack.
- Know how to tear it down cleanly.

16.2 Lab: Run a Pod from YAML

Use the example file:

- `examples/kube/webpod.yaml`

Commands:

```
podman play kube examples/kube/webpod.yaml      # create resources from YAML
podman ps                                       # list containers
curl -sS http://127.0.0.1:8080/ | head          # verify nginx responds
podman kube down examples/kube/webpod.yaml      # tear down resources created by play kube
```

Inspect created resources:

```
podman ps -a --pod  # list containers
podman pod ps  # list pods
```

Common gotcha:

- resources created by play kube are named based on YAML metadata
- use `podman kube down <yaml>` to clean up consistently

16.3 Production Note

You can also run Kube YAML under systemd using a Quadlet `.kube` unit. This can be useful for "YAML-defined" services without a full cluster.

Optional lab:

- copy `examples/quadlet/webpod.kube` and `examples/quadlet/webpod.yaml` into `~/.config/containers/systemd/`
- reload systemd and start `webpod.service`
- verify `http://127.0.0.1:8084/`

Commands:

```
mkdir -p ~/.config/containers/systemd  # create directory
cp examples/quadlet/webpod.kube ~/.config/containers/systemd/  # copy file
cp examples/quadlet/webpod.yaml ~/.config/containers/systemd/  # copy file
systemctl --user daemon-reload                  # regenerate units from Quadlet files
systemctl --user start webpod.service           # start the YAML-defined pod
curl -sS http://127.0.0.1:8084/ | head          # verify nginx responds
systemctl --user stop webpod.service            # stop the service
```

16.4 Secrets Note

Kubernetes YAML secrets often appear as base64-encoded strings.

- Base64 is not encryption.

- Do not treat YAML `Secret` objects as safe to commit unless you are using an encryption workflow (see the external secrets survey).

If you need repeatable dev secrets:

- keep secret material out of YAML
- provision secrets out-of-band (Podman secret, systemd creds, SOPS)

16.5 Limitations To Know

`podman play kube` is not full Kubernetes.

Examples of things that may differ or be unsupported:

- Services/Ingress behavior
- advanced controllers (Deployments, StatefulSets)
- cluster-level storage APIs

Treat it as a local runner for a subset of YAML.

16.6 Checkpoint

- You can run and stop a YAML-defined pod.
- You understand the security implications of YAML-stored secrets.

16.7 Quick Quiz

1) Why is base64 not a secret storage mechanism?

2) What is the safest teardown command for resources created by `play kube`?

16.8 Further Reading

- `podman-play-kube(1)`: https://docs.podman.io/en/latest/markdown/podman-play-kube.1.html
- `podman-kube-down(1)`: https://docs.podman.io/en/latest/markdown/podman-kube-down.1.html
- Kubernetes objects overview: https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/
- Kubernetes Secrets (base64 caveat): https://kubernetes.io/docs/concepts/configuration/secret/

17 Module 11: Production Baseline (systemd + Quadlet)

Quadlet lets you define containers/pods as declarative unit files that systemd manages.

17.1 Learning Goals

- Manage containers with systemd user services.
- Use Quadlet `.container`, `.pod`, `.network`, and `.volume` units.
- Make services reboot-safe with predictable restarts.
- Debug generator failures quickly.

17.2 Why Quadlet

- It is the most "Linux-native" way to operate Podman services.
- systemd gives you restart policies, ordering, logs, and boot integration.

17.3 Boot Safety (Rootless)

If you want user services to start on boot:

```
sudo loginctl enable-linger "$USER"  # allow user services to start at boot
```

17.4 Where Quadlet Files Live

- `~/.config/containers/systemd/`

systemd generates units from these files.

17.5 How "Enable" Works for Quadlet

Quadlet units are generated at daemon-reload time.

- You generally do not rely on `systemctl enable` for generated units.
- Instead, include an `[Install]` section in the Quadlet file (like `WantedBy=default.target`).
- After `systemctl --user daemon-reload`, systemd will have the symlinks it needs.

If you remove a Quadlet file, you must reload systemd to remove the generated unit.

17.6 Helpful Podman Commands

List discovered Quadlets:

```
podman quadlet list  # list discovered Quadlet definitions
```

Print the resolved Quadlet file:

```
podman quadlet print hello-nginx.container  # show the resolved Quadlet file
```

17.7 Lab: Your First Quadlet Container

Use the example unit:

- `examples/quadlet/hello-nginx.container`

Install it:

```
mkdir -p ~/.config/containers/systemd                    # Quadlet search path
cp examples/quadlet/hello-nginx.container ~/.config/containers/systemd/  # install unit
systemctl --user daemon-reload                           # regenerate units from Quadlet files
systemctl --user start hello-nginx.service               # start the service
```

```
systemctl --user status hello-nginx.service             # show status
curl -sS http://127.0.0.1:8081/ | head                  # verify HTTP response
```

Stop and disable:

```
systemctl --user stop hello-nginx.service               # stop the service
rm -f ~/.config/containers/systemd/hello-nginx.container  # remove the Quadlet definition file
systemctl --user daemon-reload                          # remove generated unit from systemd
```

Notes:

- The running container name is typically `systemd-<unit>` unless you set `ContainerName=`.
- Quadlet supports many `podman run` flags via `[Container]` keys.

17.8 Lab: Pre-Create a Network and Volume (Quadlet)

Use the example units:

- `examples/quadlet/labnet.network`
- `examples/quadlet/labdata.volume`

Install them:

```
mkdir -p ~/.config/containers/systemd                   # Quadlet search path
cp examples/quadlet/labnet.network ~/.config/containers/systemd/        # install network unit
cp examples/quadlet/labdata.volume ~/.config/containers/systemd/        # install volume unit
systemctl --user daemon-reload                          # regenerate units
systemctl --user start labnet-network.service           # create network
systemctl --user start labdata-volume.service           # create volume
```

Verify objects exist:

```
podman network ls | grep labnet  # list networks
podman volume ls | grep labdata  # list volumes
```

Cleanup:

```
systemctl --user stop labnet-network.service || true    # stop unit (ignore if missing)
systemctl --user stop labdata-volume.service || true    # stop unit (ignore if missing)
rm -f ~/.config/containers/systemd/labnet.network   # remove the Quadlet definition file
rm -f ~/.config/containers/systemd/labdata.volume   # remove the Quadlet definition file
systemctl --user daemon-reload                          # regenerate units
podman network rm labnet 2>/dev/null || true        # remove the network object
podman volume rm labdata 2>/dev/null || true        # remove the volume object (data loss)
```

Logs:

```
journalctl --user -u hello-nginx.service -n 50 --no-pager  # view user-service logs
```

17.9 Debugging Quadlet Syntax

If systemd cannot find your generated service, the generator may have failed.

Dry-run the generator:

```
/usr/lib/systemd/system-generators/podman-system-generator --user --dryrun
```

Show generator verification output:

```
systemd-analyze --user --generators=true verify hello-nginx.service  # analyze/verify systemd units
```

When debugging:

- confirm your file is in a searched directory

- confirm keys are spelled correctly (unknown keys can break generation)
- start with a minimal unit and add options incrementally

17.10 Dependencies Between Quadlets

Quadlet can translate dependencies written against `.network`/`.volume`/`.container` files.

Pattern:

- define infra objects (`.network`, `.volume`)
- make containers `Requires=` and `After=` those units

This prevents race conditions on boot.

17.11 Upgrades and Rollback

Baseline approach:

- pin images by digest in production
- upgrade by:
  - pulling a new digest
  - updating the Quadlet unit
  - restarting

Rollback is "switch back to previous digest and restart".

Operational note:

- record the previous digest so rollback is fast

17.12 Secrets

Do not store secret material in unit files.

Use:

- Podman secrets (local-first)
- external systems (survey module)

See:

- `modules/11-quadlet-secrets.md`

17.13 Checkpoint

- You can start/stop a container via systemd user services.
- You can find logs in journald.
- You can debug why a unit did not generate.

17.14 Further Reading

- Quadlet and Podman systemd integration: https://docs.podman.io/en/latest/markdown/podman-systemd.unit.5.html
- `podman-quadlet(1)`: https://docs.podman.io/en/latest/markdown/podman-quadlet.1.html
- systemd unit basics: https://www.freedesktop.org/software/systemd/man/latest/systemd.unit.html
- systemd user services: https://www.freedesktop.org/software/systemd/man/latest/systemd.service.html
- journald: https://www.freedesktop.org/software/systemd/man/latest/journald.html

18 Module 11 Add-On: Secrets with Quadlet + systemd (Rootless)

This add-on shows how to run reboot-safe services while keeping secret material out of unit files and environment variables.

18.1 Learning Goals

- Run a rootless systemd user service that consumes a secret as a file.
- Keep secret material out of:
  - unit files
  - `Environment=` lines
  - shell history
- Rotate secrets safely with rollback.

18.2 Recommended Pattern

- Store secret material as a Podman secret (local-first).
- Reference the secret by name in the Quadlet unit.
- The container reads from `/run/secrets/<name>`.

Guidelines:

- Treat secret names as part of your deployment config.
- Prefer versioned secret names for rotation (`db_password_v1`, `db_password_v2`).
- Assume many apps only read secrets at startup.

18.3 Lab: Quadlet Unit Consuming a Secret

Prereqs:

- Rootless Podman installed.
- systemd user session available.

If you want this service to start on boot (common for a server), enable lingering for your user:

```
sudo loginctl enable-linger "$USER"  # allow user services to start at boot
```

1) Create the secret (example only):

```
printf '%s' 'example-password' | podman secret create db_password -  # print text without trailing newl
```

2) Create a Quadlet `.container` unit.

Location (common):

- `~/.config/containers/systemd/`

Example filename:

- `~/.config/containers/systemd/example-app.container`

Example unit skeleton (adjust image/command to your app):

```
[Unit]
Description=Example app with secret

[Container]
Image=docker.io/library/busybox:latest
Secret=db_password
Exec=sh -lc 'test -f /run/secrets/db_password && sleep 3600'
```

```
[Service]
Restart=always

[Install]
WantedBy=default.target
```

3) Reload and start:

```
systemctl --user daemon-reload          # regenerate units from Quadlet files
systemctl --user start example-app.service   # start the service
systemctl --user status example-app.service  # show status
```

4) Verify logs do not contain secret values:

```
journalctl --user -u example-app.service -n 50 --no-pager  # view user-service logs
```

18.4 Rotation

Use versioned secret names and update the Quadlet unit to reference the new secret name, then restart.

Rule: do not delete the old secret until the new service instance is verified.

18.4.1 Rotation Procedure (Template)

1) Create the new secret:

```
printf '%s' 'new-value' | podman secret create db_password_v2 -  # print text without trailing newline
```

2) Update the Quadlet file:

- change `Secret=db_password_v1` to `Secret=db_password_v2`

3) Restart:

```
systemctl --user daemon-reload               # regenerate units after editing the Quadlet file
systemctl --user restart example-app.service # restart to pick up new secret reference
```

4) Verify behavior:

```
systemctl --user status example-app.service                 # show status
journalctl --user -u example-app.service -n 100 --no-pager  # view logs
```

5) Remove the old secret only after verification:

```
podman secret rm db_password_v1  # delete the old secret after verification
```

18.5 Further Reading

- `podman-secret(1)`: https://docs.podman.io/en/latest/markdown/podman-secret.1.html
- Quadlet and Podman systemd integration: https://docs.podman.io/en/latest/markdown/podman-systemd.unit.5.html
- systemd credentials (service-provisioned files): https://www.freedesktop.org/software/systemd/man/latest/systemd.exec.
- OWASP Secrets Management Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/Secrets_Management_Cheat_

18.6 Checkpoint

- You can run a Quadlet-managed service that reads a secret from `/run/secrets/...`.
- You can rotate a secret by switching the referenced secret name and restarting.
- You can verify journald logs do not contain secret values.

18.7 Quick Quiz

1) Why is it safer to mount secrets as files than to pass them in environment variables?

2) Why should you keep the old secret around until after verification?

19 Module 12: Security Deep Dive

This module focuses on reducing blast radius and making your container posture auditable.

19.1 Learning Goals

- Apply least privilege inside and outside containers.
- Use read-only filesystems and drop capabilities.
- Understand SELinux and why it matters on Fedora/RHEL.
- Understand image trust at a practical level.
- Build a secure-by-default run configuration.

19.2 Baseline Hardening Checklist

- Run rootless when possible.
- Run as non-root inside the container.
- Drop capabilities you do not need.
- Use a read-only root filesystem when possible.
- Use `tmpfs` for writable scratch.
- Limit resources (memory/CPU).
- Avoid mounting the Docker socket equivalent (do not give containers control of Podman).

Also consider:

- `--security-opt no-new-privileges`
- minimal mounts (avoid mounting home directories)
- immutable config (mount config read-only)

19.3 Lab: Drop Capabilities

Start with a simple container and drop all Linux capabilities:

```
podman run --rm --cap-drop=ALL docker.io/library/alpine:latest id  # run a container
```

If your workload breaks, add only the specific capabilities required.

Avoid using `--privileged` as a workaround.

19.4 Resource Limits

Example:

```
podman run --rm --memory 256m --pids-limit 200 docker.io/library/alpine:latest sh -lc 'echo ok'  # run
```

19.5 Lab: No-New-Privileges

```
podman run --rm --security-opt no-new-privileges docker.io/library/alpine:latest id  # run a container
```

19.6 Lab: Read-Only Root FS

Run nginx with a read-only root filesystem (this may require a writable temp location depending on image behavior):

```
podman run --rm -p 8080:80 --read-only --tmpfs /var/cache/nginx --tmpfs /var/run docker.io/library/nginx
```

If it fails:

- read the error
- identify the write path

- add a targeted `tmpfs` or volume

Try pairing with "drop everything":

```
podman run --rm -p 8080:80 --read-only --cap-drop=ALL --security-opt no-new-privileges --tmpfs /var/cac
```

If it fails, treat the error as a map of required writes/capabilities and add back only what is necessary.

19.7 SELinux (Fedora/RHEL)

If SELinux is enforcing:

- prefer `:Z` for private bind mounts
- prefer volumes when possible

Do not disable SELinux to "fix" containers in production.

If you see mount permission errors with bind mounts:

- try `:Z`
- prefer volumes

19.8 Image Trust (Practical)

Minimum viable practices:

- pin by digest
- use minimal base images when possible
- track upstream sources
- scan images (tooling varies by org)

Supply chain habits:

- prefer official images or your organization's curated base images
- avoid running unverified installer scripts in builds

19.9 Checkpoint

- You can explain the difference between rootless and non-root-in-container.
- You can make a service run read-only (or explain why it cannot).

19.10 Quick Quiz

1) Why is `--privileged` almost always the wrong answer?
2) Why is digest pinning useful even if you trust the upstream?

19.11 Further Reading

- Linux capabilities (man7): https://man7.org/linux/man-pages/man7/capabilities.7.html
- `seccomp(2)` (man7): https://man7.org/linux/man-pages/man2/seccomp.2.html
- SELinux with containers (RHEL docs): https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/9/html/u selinux-with-containers_using-selinux
- Podman security docs: https://github.com/containers/podman/blob/main/docs/tutorials/security.md

20 Module 13: Troubleshooting and Ops

This module is about turning "it doesn't work" into a short checklist.

20.1 Learning Goals

- Diagnose startup failures quickly.
- Know where to look: logs, inspect, events, journald.
- Debug networking and storage issues.
- Do failure drills and recover.

20.2 The Debug Loop

1) Confirm container state:

```
podman ps -a  # state, exit codes, recent failures
```

2) Read logs:

```
podman logs <name>  # app output / crash reason
```

3) Inspect config:

```
podman inspect <name> | less  # ports, mounts, env, command
```

4) Reproduce interactively:

```
podman run --rm -it <image> sh  # reproduce interactively
```

If the container exits too fast to exec into it:

- run the same image with an interactive shell
- or override command to `sleep 3600` and then `exec` in

20.3 More Tools

Events (what Podman is doing):

```
podman events  # see Podman lifecycle events
```

Live resource stats:

```
podman stats  # live CPU/mem/io stats
```

Show processes in a container:

```
podman top <name>  # processes inside the container
```

Disk usage:

```
podman system df  # disk usage summary
```

Cleanup (be careful on shared systems):

```
podman system prune  # remove unused objects (be careful)
```

20.4 Failure Drills (With Expected Observations)

Do these on purpose; they make you faster in real incidents.

1) Port conflict

- symptom: container fails to start, error mentions bind/listen
- check: `podman port <name>` or the error in logs
- fix: change host port

2) Bad command

- symptom: container exits immediately with non-zero
- check: `podman logs <name>` shows "not found" or usage
- fix: correct command/entrypoint/args

3) Permissions

- symptom: "permission denied" writing to mounted path
- check: `podman inspect <name> --format '{{json .Mounts}}'`
- fix: prefer volume or correct ownership in user namespace; add `:Z` for bind mounts on Fedora/RHEL

4) DNS

- symptom: app cannot resolve service name
- check: `podman network inspect <net>`; run a debug container and `getent hosts <name>`
- fix: ensure both containers are on the same user-defined network

20.5 systemd/Quadlet Troubleshooting

Check status:

```
systemctl --user status <service>  # systemd view: active/failed
```

Logs:

```
journalctl --user -u <service> -n 200 --no-pager  # service logs from journald
```

Reload after unit changes:

```
systemctl --user daemon-reload           # reload unit changes
systemctl --user restart <service>       # restart the service
```

20.6 Networking Checklist

- Is the port published to the host?
- Is the service listening on the expected interface?
- Are containers on the same network?
- Does DNS resolve container names?

20.7 Storage Checklist

- Is the correct volume mounted?
- Are permissions correct for the container user?
- On Fedora/RHEL: is SELinux labeling correct (`:Z`/`:z`)?

20.8 Failure Drills (Do These)

1) Port conflict

- start a service on 8080
- try to start another service on 8080
- fix by changing published port

2) Permission denied

- mount a directory your container cannot write
- fix with ownership/permissions or a different storage approach

3) Bad image tag

- deploy with a tag that gets replaced upstream

- fix by pinning to digest

20.9 Checkpoint

- You can debug a failed service without guessing.
- You have a repeatable recovery flow.

20.10 Quick Quiz

1) What is the difference between `podman logs` and `journalctl --user -u ...`?

2) What do you do when a container exits too quickly to exec into it?

20.11 Further Reading

- `podman-events(1)`: https://docs.podman.io/en/latest/markdown/podman-events.1.html
- `podman-stats(1)`: https://docs.podman.io/en/latest/markdown/podman-stats.1.html
- systemd journalctl: https://www.freedesktop.org/software/systemd/man/latest/journalctl.html
- `systemd-analyze(1)` (verify, generators): https://www.freedesktop.org/software/systemd/man/latest/systemd-analyze.html

21 Module 14: Maintenance and Auto-Updates

Auto-updates can be useful, but they are a policy decision.

21.1 Learning Goals

- Understand what Podman auto-update does.
- Configure a Quadlet-managed container for registry-based auto-update.
- Build a safe rollout + rollback plan.

21.2 What Auto-Update Is

Podman auto-update can:

- check for new images in a registry
- pull updates
- restart the systemd unit that runs the container

What it cannot do by itself:

- perform multi-step migrations safely
- coordinate multiple hosts without additional tooling
- guarantee compatibility between versions

21.3 Tags, Digests, and Policy

Auto-update works best with tags (because tags can move).

Digest pinning works best for reproducibility.

Choose intentionally:

- If you need "always latest patch": use a stable tag + auto-update, and invest in monitoring + rollback.
- If you need "always reproducible": pin digests and upgrade with a controlled change.

In regulated environments, digest pinning is often the default.

21.4 Lab: Enable Registry Auto-Update (Single Service)

Use:

- `examples/quadlet/autoupdate-nginx.container`

Install:

```
mkdir -p ~/.config/containers/systemd  # create directory
cp examples/quadlet/autoupdate-nginx.container ~/.config/containers/systemd/  # copy file
systemctl --user daemon-reload                    # regenerate units from Quadlet files
systemctl --user start autoupdate-nginx.service  # start the service
```

Run auto-update manually:

```
podman auto-update  # pull updates and restart labeled units
```

Watch what changed:

```
systemctl --user status autoupdate-nginx.service                # show status
journalctl --user -u autoupdate-nginx.service -n 100 --no-pager  # view logs
```

On many systems there is also a timer unit you can enable (name varies by distro). If present:

```
systemctl --user list-unit-files | grep auto-update  # look for an auto-update timer
```

If a timer exists and you choose to use it, enable it like a normal systemd unit.

21.5 Rollback Plan (Required If You Auto-Update)

Minimum rollback plan:

- know the previous working image reference
- be able to restart the service back to that version
- have monitoring to detect restart loops quickly

If you rely on tags:

- you may need to pin to a specific digest during rollback
- you may need a local cache/registry to keep old versions available

21.6 Safe Rollout Rules

- Prefer digest-pinned images for production unless you explicitly accept the risk.
- If you use auto-update:
  - add healthchecks
  - alert on restart loops
  - document rollback

Treat auto-update as an operational feature, not a convenience hack.

21.7 Checkpoint

- You can explain why auto-update is optional, not mandatory.
- You can trigger and observe an auto-update.

21.8 Quick Quiz

1) Why can auto-update increase risk for stateful services?
2) What must exist before you turn on auto-update in production?

21.9 Further Reading

- `podman-auto-update(1)`: https://docs.podman.io/en/latest/markdown/podman-auto-update.1.html
- systemd timers: https://www.freedesktop.org/software/systemd/man/latest/systemd.timer.html
- systemd service restart policies: https://www.freedesktop.org/software/systemd/man/latest/systemd.service.html

22 Capstone: Reboot-Safe Local Stack with Secrets, Backups, and Upgrades

This capstone focuses on operational excellence, not app development.

22.1 Goal

Run a small stack as rootless systemd user services (Quadlet-first) that:

- survives reboot
- keeps state in volumes
- uses secrets as files (not env vars)
- has a backup + restore flow
- has an upgrade + rollback flow (digest-pinned)

Success criteria:

- After a reboot, both services come back without manual intervention.
- You can produce a backup file and prove you can restore it.
- You can upgrade MariaDB/Adminer versions with a documented rollback.

22.2 Reference Stack

- DB: MariaDB
- UI: Adminer (web DB admin)

This gives you a realistic stateful service without writing code.

22.3 Deliverables

- Quadlet units for:
  - a private network
  - a DB container
  - a UI container
  - (optional) a backup timer/service
- A runbook:
  - first deploy
  - rotate DB password
  - take backup
  - restore from backup
  - upgrade pinned digests
  - rollback

22.4 Build It

Use the provided example units:

- `examples/quadlet/capnet.network`
- `examples/quadlet/mariadb-data.volume`
- `examples/quadlet/cap-backups.volume`
- `examples/quadlet/cap-mariadb.container`
- `examples/quadlet/cap-adminer.container`
- `examples/quadlet/cap-backup.container` (optional)

1) Create the DB root password as a Podman secret (example only):

```
read -s -p 'MariaDB root password: ' P  # prompt for input
printf '\n'  # print text without trailing newline
```

```
printf '%s' "$P" | podman secret create mariadb_root_password -  # print text without trailing newline
unset P  # unset an environment variable
```

2) Install Quadlet units:

```
mkdir -p ~/.config/containers/systemd  # create directory
cp examples/quadlet/capnet.network ~/.config/containers/systemd/  # copy file
cp examples/quadlet/mariadb-data.volume ~/.config/containers/systemd/  # copy file
cp examples/quadlet/cap-backups.volume ~/.config/containers/systemd/  # copy file
cp examples/quadlet/cap-mariadb.container ~/.config/containers/systemd/  # copy file
cp examples/quadlet/cap-adminer.container ~/.config/containers/systemd/  # copy file
```

3) Enable linger (boot start):

```
sudo loginctl enable-linger "$USER"  # allow user services to start at boot
```

4) Start services:

```
systemctl --user daemon-reload             # regenerate units from Quadlet files
systemctl --user start cap-mariadb.service # start DB
systemctl --user start cap-adminer.service # start UI
```

5) Validate:

- Adminer responds on `http://127.0.0.1:8082/`
- DB is not published to the host

Validate DB is private:

```
podman port cap-mariadb || true  # show published ports
```

Expected: no published ports.

22.5 First Data (Required)

Create a test database/table so you have something to back up:

```
podman run --rm --network capnet --secret mariadb_root_password docker.io/library/mariadb:11 sh -lc 'exp
```

22.6 Optional: Scheduled Backups

1) Install backup Quadlet and timer:

```
cp examples/quadlet/cap-backup.container ~/.config/containers/systemd/  # copy file
mkdir -p ~/.config/systemd/user  # create directory
cp examples/systemd-user/cap-backup.timer ~/.config/systemd/user/  # copy file
systemctl --user daemon-reload                 # reload new units
systemctl --user enable --now cap-backup.timer # enable scheduled backups
```

2) Trigger a backup immediately:

```
systemctl --user start cap-backup.service  # run a backup now
```

3) Verify backup files exist:

```
podman run --rm -v cap_backups:/backups docker.io/library/alpine:latest ls -la /backups  # run a contai
```

Note:

- backups are stored in the `cap_backups` volume
- the backup unit runs `mysqldump` inside a container

22.7 Backup and Restore (Required)

Backup requirements:

- backup output goes to a dedicated volume (or host path)
- backup command runs without exposing passwords in logs

Restore requirements:

- documented, tested procedure
- includes a rollback path

Restore sketch:

- start a throwaway client container on `capnet`
- feed a `.sql` file into `mysql -h db -u root`
- verify tables

22.7.1 Backup (Manual)

Trigger a backup:

```
systemctl --user start cap-backup.service  # run a backup now
```

Find the newest backup file:

```
podman run --rm -v cap_backups:/backups docker.io/library/alpine:latest sh -lc 'ls -1 /backups | tail -
```

22.7.2 Restore (Manual)

Pick a backup filename from the previous step, then restore:

```
BACKUP_FILE=all-<timestamp>.sql
podman run --rm --network capnet -v cap_backups:/backups --secret mariadb_root_password docker.io/libra
```

Verify the data is present:

```
podman run --rm --network capnet --secret mariadb_root_password docker.io/library/mariadb:11 sh -lc 'ex
```

Rollback idea:

- restore into a fresh volume and validate before switching (advanced)

22.8 Upgrade and Rollback (Required)

- Record current image digests.
- Upgrade by changing digests in units and restarting.
- Roll back by restoring previous digests and restarting.

22.8.1 Record Digests

Record what you are running:

```
podman images --digests | grep -E 'mariadb|adminer'  # list images
podman inspect cap-mariadb --format '{{.ImageName}}'  # inspect container/image metadata
podman inspect cap-adminer --format '{{.ImageName}}'  # inspect container/image metadata
```

22.8.2 Pin by Digest (Recommended)

In your `.container` files, set:

- `Image=docker.io/library/mariadb@sha256:<digest>`
- `Image=docker.io/library/adminer@sha256:<digest>`

Then:

```
systemctl --user daemon-reload                  # regenerate units after edits
systemctl --user restart cap-mariadb.service    # restart DB
systemctl --user restart cap-adminer.service    # restart UI
```

Rollback is the same procedure with the previous digests.

22.9 Password Rotation (Required)

Rotation plan for root password:

For this lab, choose a password without quotes or newlines to avoid shell/SQL escaping issues.

1) Create a new secret (versioned name):

```
read -s -p 'New MariaDB root password: ' P  # prompt for input
printf '\n'  # print text without trailing newline
printf '%s' "$P" | podman secret create mariadb_root_password_v2 -  # print text without trailing newli
unset P  # unset an environment variable
```

2) Change the password inside MariaDB while authenticated with the old one:

```
podman run --rm --network capnet --secret mariadb_root_password --secret mariadb_root_password_v2 docke
```

3) Update Quadlet to reference the new secret name and restart MariaDB.

4) Verify logins with the new secret.

5) Remove the old secret only after verification:

```
podman secret rm mariadb_root_password  # remove old secret after verification
```

22.10 Notes

- Password rotation often implies updating both the secret and the DB user credentials.
- Keep the old password available until the new one is verified.

22.11 Checkpoint

- You can bring the stack up via Quadlet and it survives reboot.
- DB has no published host ports; only the UI is exposed.
- You can produce a backup file and restore it successfully.
- You can upgrade using digest pinning and roll back to a previous digest.

22.12 Quick Quiz

1) Why is it important to test restore, not just backup?

2) What is the operational advantage of deploying by digest rather than by tag?

22.13 Further Reading

- `podman-secret(1)`: https://docs.podman.io/en/latest/markdown/podman-secret.1.html
- Quadlet and Podman systemd integration: https://docs.podman.io/en/latest/markdown/podman-systemd.unit.5.html
- MariaDB logical backup (`mysqldump`): https://mariadb.com/kb/en/mysqldump/
- Adminer project docs: https://www.adminer.org/
- systemd timers: https://www.freedesktop.org/software/systemd/man/latest/systemd.timer.html

23 External Secrets Survey (Thorough Intro, Optional Implementation)

Podman secrets are a good local-first baseline, but most teams eventually need one or more of:

- encryption-at-rest on the host
- multi-host distribution
- automated rotation
- auditing and policy

This module teaches the landscape so learners can choose an external approach confidently.

This is intentionally not a single "do this" recipe.

External secrets are an architecture and operations decision.

23.1 What You Are Optimizing For

Use this checklist to pick a system:

- How many hosts need the secret?
- How often does it rotate?
- Who/what is allowed to read it (policy)?
- Do you need audit logs?
- What happens when the secrets system is down?
- How do you bootstrap a brand new host?

Also consider:

- do you need dynamic credentials (leases) or static secrets
- how do you revoke access
- how do you handle break-glass scenarios

23.2 Option 1: systemd Credentials (Host-Native)

What it is:

- systemd can provision credentials to services as files.
- Credentials can be stored encrypted at rest on the host.

Why it fits this course:

- Production baseline already uses systemd user services (Quadlet-first).
- Delivery model matches the container best practice: read a file.

Typical pattern:

- Store encrypted credential material on the host.
- systemd materializes it to a runtime file.
- Container reads that file via a mount.

Operational notes:

- great for systemd-first deployments
- policy is typically "who can read files / run services" on that host
- still need a story for distributing the encrypted credential material to new hosts

When to choose it:

- Single host or small fleet.
- You want minimal moving parts.

23.3 Option 2: SOPS (GitOps-Friendly Encrypted Files)

What it is:

- Keep secrets encrypted in git.
- Decrypt on the host/CI using an identity (age or GPG, plus cloud KMS in some setups).

Benefits:

- Change history and reviews are straightforward.
- Bootstrapping is manageable for small teams.

Tradeoffs:

- Rotation is usually a process, not a lease.
- You must secure decryption keys carefully.

Bootstrap story:

- you need a way to provision the age/GPG/KMS identity onto a new host
- you need a safe place to store recovery keys

Best-fit pattern with containers:

- Decrypt to a root-owned file with `0600`.
- Mount into the container read-only.
- Never persist decrypted files into images.

23.4 Option 3: Vault-Class Secret Managers (Centralized)

What it is:

- Central policy + auth + audit.
- Dynamic secrets (leases) and automated rotation.

Benefits:

- Strong governance and scaling story.
- Short-lived credentials reduce blast radius.

Costs:

- Operational overhead.
- Availability becomes critical path for deploy/boot.

Common bridge patterns (recommended):

- Sidecar/agent writes a file to a shared volume; app reads the file.
- systemd service fetches secret at start, writes to a protected file, then starts the container.

Operational notes:

- design for availability: what happens on restart if Vault is unreachable
- treat auth methods (Kubernetes auth, AppRole, OIDC, etc.) as part of the threat model
- dynamic creds reduce blast radius but increase moving parts

23.5 Comparison Table (Mental Model)

Answer these questions:

- “Do I need secrets on more than one host?”
- “Do I need automatic rotation and revocation?”
- “Do I need centralized policy and audit?”

Typical outcomes:

- single host: systemd creds or local secrets may be enough
- small fleet, GitOps: SOPS is a strong fit
- larger org/compliance: Vault-class is common

23.6 What Does Not Change

Regardless of external system:

- The container should read secrets from files.
- Do not pass secret values in env vars, CLI args, or logs.
- Use rotation-friendly names and restart/roll strategies.

23.7 Migration Path from Podman Secrets

If you start with Podman secrets (local-first), the clean migration is:

- external manager writes/refreshes a file
- your service consumes that file (mount) or uses systemd credentials

This avoids rewriting applications that already expect file-based secrets.

23.8 Checkpoint

- You can explain the tradeoffs between: systemd credentials, SOPS, and Vault-class secret managers.
- You can describe a bootstrap story for a new host (how it gets the ability to decrypt/fetch).
- You can describe a file-based delivery pattern that keeps apps unchanged.

23.9 Quick Quiz

1) In one sentence: why is base64 not encryption?

2) What question best distinguishes SOPS-style encrypted files from Vault-style leased secrets?

23.10 Further Reading

- systemd credentials (service-provisioned files): https://www.freedesktop.org/software/systemd/man/latest/systemd.exec.l
- Mozilla SOPS: https://github.com/getsops/sops
- age (file encryption tool often used with SOPS): https://github.com/FiloSottile/age
- HashiCorp Vault: https://www.vaultproject.io/
- Kubernetes Secrets (baseline for comparison): https://kubernetes.io/docs/concepts/configuration/secret/

24 Cheatsheets

24.1 podman-cli.md

25 Podman CLI Cheat Sheet

25.1 Containers

```
podman run --rm <image> <cmd>                          # run a one-shot container
podman run -d --name <name> <image>                    # run in background with a stable name
podman ps -a                                           # list containers (including stopped)
podman logs <name>                                     # show container logs
podman exec -it <name> sh                              # run a shell in a running container
podman stop <name>                                     # stop a running container
podman rm -f <name>                                     # force remove container
```

25.2 Images

```
podman pull <image>                                    # download an image
podman build -t localhost/<name>:<tag> .                # build from Containerfile in current dir
podman tag localhost/<name>:<tag> <registry>/<ns>/<name>:<tag>  # add another name
podman push <registry>/<ns>/<name>:<tag>                # upload to a registry
podman image history <image>                           # show layer history
podman images                                          # list local images
podman inspect <image-or-container>                    # show JSON metadata
podman rmi <image>                                      # remove an image from local storage

# Cleanup
podman image prune          # remove unused images (frees disk)
podman system prune         # remove unused objects (be careful)
podman builder prune        # remove build cache (if supported)
```

25.3 Port Publishing

```
podman run -d -p 8080:80 <image>                  # all interfaces
podman run -d -p 127.0.0.1:8080:80 <image>        # loopback only
podman run -d -p 8080:80 -p 8443:443 <image>      # multiple ports
podman run -d -p 5053:53/udp <image>              # UDP port
podman run -d -p 80 <image>                       # random host port
podman port <name>                                # show active port mappings
podman inspect <name> --format '{{json .NetworkSettings.Ports}}'  # show raw port mapping JSON
```

25.4 Networks

```
# Create / list / inspect / remove
podman network create <net>                                  # create a user-defined network
podman network create --internal <net>                          # no outbound access
podman network create --subnet 172.28.0.0/24 --gateway 172.28.0.1 <net>  # create a network
podman network ls                                           # list networks
podman network inspect <net>                                # show network details
podman network rm <net>                                     # remove the network

# Connect / disconnect a running container
podman network connect <net> <name>                          # attach an existing container
podman network disconnect <net> <name>                       # detach an existing container

# Which networks is a container on?
```

```
podman inspect <name> --format '{{json .NetworkSettings.Networks}}'  # show network attachments

# Container IP per network
podman inspect <name> --format '{{range $n,$v := .NetworkSettings.Networks}}{{$n}}: {{$v.IPAddress}}{{"

# Which containers are on a network?
podman network inspect <net> --format '{{range $id,$c := .Containers}}{{$c.Name}} {{end}}'  # list cont

# Check DNS state of a network
podman network inspect <net> --format '{{.DNSEnabled}}'      # true if container name DNS works

# Live DNS lookup from a debug container
podman run --rm --network <net> docker.io/library/alpine:latest sh -lc 'getent hosts <target>'  # run a

# All container IPs (quick overview)
podman ps -q | xargs -I{} podman inspect {} --format '{{.Name}}: {{range .NetworkSettings.Networks}}{{.
```

25.5 Volumes

```
podman volume create <vol>  # create a volume
podman volume ls  # list volumes
podman volume inspect <vol>  # inspect a volume
```

25.6 Pods

```
podman pod create --name <pod> -p 8080:80  # create a pod
podman pod ps  # list pods
podman pod rm -f <pod>                              # stop and remove pod + containers
```

25.7 Secrets

```
podman secret create <name> -  # create a secret
podman secret ls  # list secrets
podman run --secret <name> <image>  # run a container
```

25.8 quadlet.md

26 Quadlet Cheat Sheet

26.1 Files

- put Quadlet files in: `~/.config/containers/systemd/`
- common extensions: `.container`, `.pod`, `.network`, `.volume`

26.2 Workflow

```
systemctl --user daemon-reload  # regenerate units from files
systemctl --user start <name>.service  # start a user service
systemctl --user status <name>.service  # show service status
journalctl --user -u <name>.service -n 100 --no-pager  # view user-service logs
```

26.3 Boot Start

```
sudo loginctl enable-linger "$USER"        # allow user services to start at boot
systemctl --user enable <name>.service     # enable the service for your user
```

26.4 rootless.md

27 Rootless Cheat Sheet

27.1 Key Idea

Rootless Podman runs as your user and uses user namespaces.

27.2 Useful Checks

```
podman info  # show Podman host configuration
grep "^$USER:" /etc/subuid /etc/subgid  # filter output
podman unshare id  # run a command inside the user namespace
```

27.3 Common Paths

- storage: `~/.local/share/containers/`
- runtime: `/run/user/<uid>/containers/`

27.4 Boot Start for systemd User Services

```
sudo loginctl enable-linger "$USER"  # allow user services to start at boot
```

27.5 security.md

28 Security Cheat Sheet

28.1 Baseline

- rootless when possible
- non-root user inside container
- do not pass secrets in env vars
- pin images by digest in production

28.2 Hardening Flags (Examples)

```
podman run --read-only --tmpfs /tmp <image>  # read-only root FS + writable temp
```

28.3 SELinux (Fedora/RHEL)

- bind mount with `:Z` (private) or `:z` (shared)

28.4 troubleshooting.md

29 Troubleshooting Cheat Sheet

29.1 Fast Triage

```
podman ps -a                  # is it running? exit code?
podman logs <name>            # app output / crash reason
podman inspect <name> | less # config: mounts, ports, command, env
```

29.2 systemd/Quadlet

```
systemctl --user status <service>                    # systemd view: active/failed
journalctl --user -u <service> -n 200 --no-pager     # service logs from journald
```

29.3 Network Checks

- verify ports: `-p host:container`
- verify container name DNS on the network

29.4 Storage Checks

- volume mounted where the app expects
- permissions for the container user
- on Fedora/RHEL: use `:Z` for private bind mounts

30 Appendix

30.1 ASSESSMENTS.md

31 Assessments

These assessments focus on practical skills.

31.1 Module Checkpoints

Each module ends with a checkpoint. Treat it as "must be able to do without notes".

31.2 Practical Exam A (Mid-Course)

Scenario:

- You are given a container that exits immediately.

Requirements:

- Determine why it exits.
- Fix it without rebuilding the image.
- Provide a short runbook: commands used, what you observed, final fix.

Rubric:

- Uses `podman ps -a`, `podman logs`, `podman inspect` effectively.
- Fix is minimal and reproducible.
- No secrets printed.

31.3 Practical Exam B (Final)

Scenario:

- You are given a two-service stack: web + db.
- The stack must survive reboot.

Requirements:

- Use Quadlet (systemd user service) to run both services.
- Use a named volume for state.
- Use a secret mounted as a file (not env vars).
- DB is private; only web is published.
- Provide backup + restore steps.

Rubric:

- Rootless and reboot-safe (linger configured if required).
- Correct storage and networking.
- Secrets handled safely.
- Clear, testable runbook.

31.4 GLOSSARY.md

32 Glossary

- container: a running (or stopped) instance of an image with its own writable layer and runtime config
- image: an OCI artifact composed of layers + config; used as a template for containers
- registry: a service that stores and distributes images
- tag: a movable name that points to an image (example: `:latest`)
- digest: a content-addressed identifier for an image (example: `@sha256:...`)
- rootless: running Podman as a normal user using user namespaces
- Quadlet: systemd integration that generates service units from `.container`/`.pod`/`.volume`/`.network` files
- SELinux: a Linux MAC system; on Fedora/RHEL it can affect mounts and container permissions
- namespace: a Linux kernel isolation feature; containers commonly use PID, mount, and network namespaces
- cgroups v2: the kernel resource-control mechanism used for CPU/memory limits and systemd integration
- user namespace: a namespace that remaps UIDs/GIDs; enables rootless containers to have "root" inside without host root
- subuid/subgid: per-user UID/GID ranges used for user namespace mappings (rootless)
- writable layer: the per-container filesystem layer on top of the image; it is not durable persistence
- volume: Podman-managed persistent storage intended for stateful data
- bind mount: a host path mounted into a container; often used for config and source code
- network (user-defined): a named bridge network with DNS enabled for container name resolution
- aardvark-dns: Podman's embedded DNS service for user-defined networks
- netavark: Podman's networking stack used to configure networks and DNS (modern Podman)
- slirp4netns: a user-mode networking helper commonly used for rootless containers
- pasta: a newer user-mode networking helper often used for rootless containers
- healthcheck: an image or runtime-defined command that reports container health (used by tooling/systemd policies)
- linger: systemd feature that allows user services to run at boot without an interactive login

32.1 FAQ.md

33 FAQ / Gotchas

This file collects common failure modes and the fastest fixes.

33.1 Rootless: "permission denied" publishing port 80

Rootless users typically cannot bind ports below 1024.

Fix options:

- Use a high port (recommended for labs): `-p 8080:80`
- If you own the system, lower the unprivileged port start:

```
sudo sysctl -w net.ipv4.ip_unprivileged_port_start=80  # allow low ports for rootless (system-wide)
```

See: `modules/06-networking.md`

33.2 Container name DNS does not work

Container DNS names work on user-defined networks (DNS enabled), not on the default network.

Fix:

```
podman network create appnet  # create a network
podman run --network appnet ...  # run a container
```

See: `modules/06-networking.md`

33.3 SELinux: bind mounts fail with permission denied (Fedora/RHEL)

If SELinux is enforcing, bind mounts may require labels.

Fix patterns:

- Private mount: `-v ./dir:/mnt:Z`
- Shared mount: `-v ./dir:/mnt:z`

See: `modules/05-storage.md`

33.4 Local registry lab fails with TLS/HTTPS errors

Some Podman configs treat an HTTP registry as insecure and require explicit configuration.

For the learning lab only:

```
podman push --tls-verify=false localhost:5000/alpine:course  # push an image to a registry
```

See: `modules/03-images-registries.md`

33.5 HEALTHCHECK missing after build

Podman warns that `HEALTHCHECK` metadata is not stored in the OCI image format.

Fix options:

- Build in docker format when you need the healthcheck stored in the image:

```
podman build --format docker -t localhost/myapp:1 .  # build an image
```

- Or define a runtime healthcheck using `podman run --health-*` flags.

See: `modules/08-building-images.md`, `examples/build/hello-bun/README.md`

33.6 `exec format error`

This almost always means an architecture mismatch (built for amd64, running on arm64, or vice versa).

Fix:

- Build on the target architecture, or
- Use a platform-aware build workflow (advanced topic)

See: `modules/08-building-images.md`, `examples/build/hello-go/README.md`

33.7 Quadlet service does not exist after adding a file

Quadlet units are generated at reload time.

Fix:

```
systemctl --user daemon-reload  # regenerate units from files
systemctl --user status <unit>  # show service status
```

See: `modules/11-quadlet.md`