Containers 2/3: Containers in Practice

Engines, images, storage, networking, builds and Compose

Walter Rudametkin

Walter.Rudametkin@univ-rennes.fr

https://rudametw.github.io/teaching

ESIR 2 SI — Architectures Logicielles

Université de Rennes







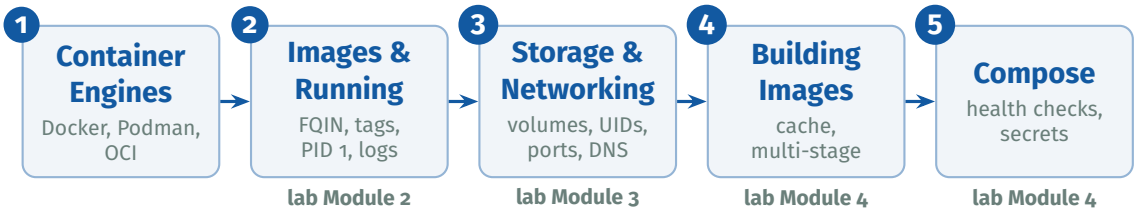






Lecture 2: Road Map

1. DevOps and Why Containers	2. Containers in Practice	3. Under the Hood and Production
lab Module 0	today · lab Modules 2–4	lab Modules 0–1 and 5



Goal: usage first: what you ran in the lab, and why.

Previously: DevOps, why containers, containers vs VMs (lecture 1).

appendix: deep dives **field guide**: recipes, troubleshooting

Build, Ship, Run



Dockerfile
(Build)



Docker Image
(Ship)

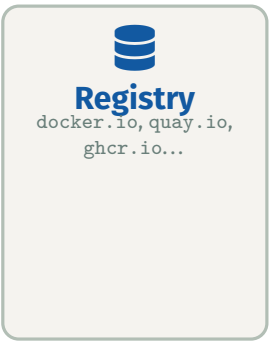


Containers
(Run)

Client → Engine → Registry







You type commands; the **engine** does the work; the **registry** stores images.

Podman: client and engine in one program, no daemon.

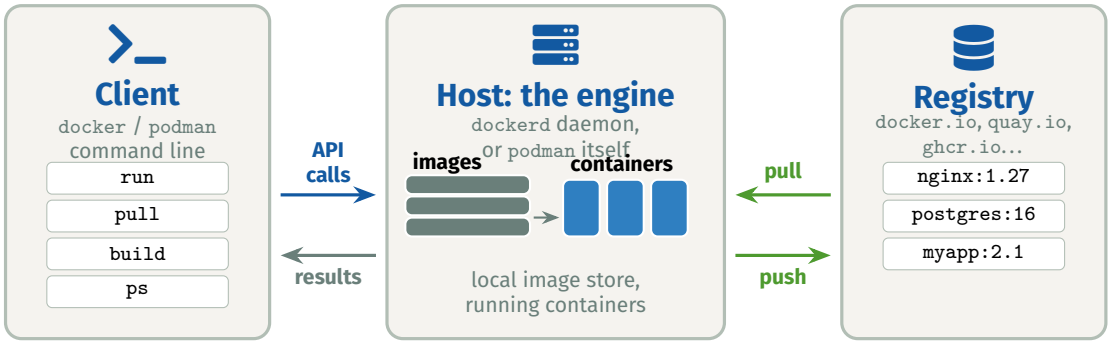
Client → Engine → Registry



The client only sends requests to the engine (an API) and prints the results.

Podman: client and engine in one program, no daemon.

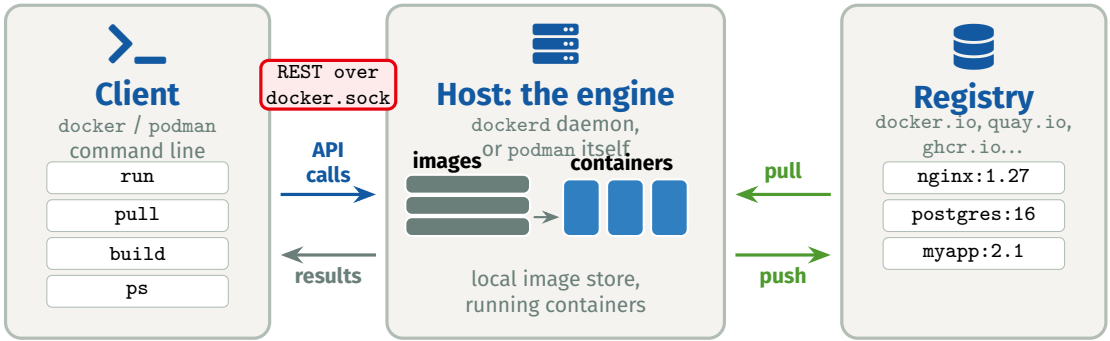
Client → Engine → Registry



`podman run docker.io/library/nginx:1.27`:
the engine **pulls** the image if missing, then **creates** the container.

Podman: client and engine in one program, no daemon.

Client → Engine → Registry



Docker: `dockerd`, a **root** daemon (rootless mode: opt-in), serves this API on `/var/run/docker.sock`.

⚠️ Access to `docker.sock` (`docker` group) ≈ root on the host.

Podman: client and engine in one program, no daemon.

The Open Container Initiative (OCI)

Founded in 2015 (Docker, CoreOS, Google, Red Hat…) to standardize:

- **Image spec**: layers (tarballs) + JSON config + manifest
  - a "Docker image" is an OCI image
- **Runtime spec**: how to run a "filesystem bundle"
  - `config.json`: process, mounts, namespaces, cgroups, capabilities…
  - low-level OCI runtimes: `runc`, `crun` (details: appendix)
- **Distribution spec**: the HTTP API of registries (push/pull)

✔ Docker, Podman, Kubernetes…all pull and run *the same images*.

Podman

Daemonless

- no background service
- runs `conmon` + `crun`
- children of *your* session

Rootless by default

- runs as an unprivileged user
- “root” inside = you outside

Docker-compatible

- same OCI images, same CLI
- optional Docker API socket

Beyond Docker

- pods, `podman kube play`
- systemd integration: Quadlets
- GUI: Podman Desktop

```
$ systemctl --user enable --now podman.socket    # Docker API
$ export DOCKER_HOST=unix://$XDG_RUNTIME_DIR/podman/podman.sock
```

Then tools written for Docker (`docker-compose`, IDEs, Testcontainers) drive Podman.

Docker ≡ Podman: Everyday Commands

Docker	Podman	Purpose
`docker pull`	`podman pull`	download an image
`docker images`	`podman images`	list local images
`docker run`	`podman run`	create + start a container
`docker ps -a`	`podman ps -a`	list containers (incl. exited)
`docker logs`	`podman logs`	stdout/stderr of PID 1
`docker exec -it`	`podman exec -it`	run a command inside
`docker inspect`	`podman inspect`	full JSON config/state
`docker rm -f`	`podman rm -f`	kill + remove a container
`docker build`	`podman build`	build an image
`docker compose`	`podman compose`*	multi-container stacks
—	`podman unshare`	enter the rootless user ns.
—	`podman machine`	Linux VM (macOS/Windows)

* a wrapper: runs `podman-compose` or `docker-compose`.

Our First Containers

```
$ podman run docker.io/library/busybox:1.36 echo hello world
hello world
$ podman run -it docker.io/library/ubuntu:24.04
root@04c0bb0a6c07:/# apt-get update && apt-get install -y figlet
root@04c0bb0a6c07:/# figlet hello; exit
$ podman run -it docker.io/library/ubuntu:24.04     # again
root@b13c164401fb:/# figlet
bash: figlet: command not found
```

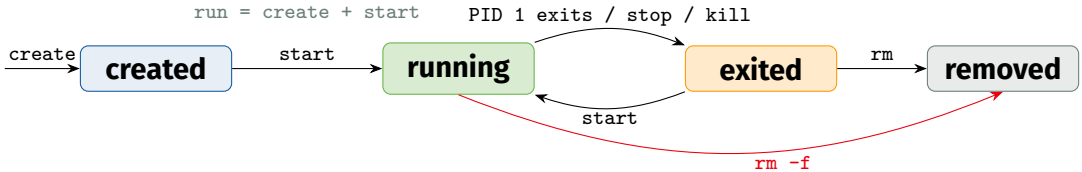
- `run` = pull (if missing) + create + start + attach; `-it` = stdin + a terminal
- each `run` = a new container: `figlet` stayed in the *old*, stopped one
- need `figlet`? build an image: containers are cattle, not pets

Background Containers, Names and Logs

```
$ podman run -d --name ticker docker.io/library/alpine:3.20 \
      sh -c 'while true; do date; sleep 1; done'
$ podman ps               # running containers only
$ podman ps -a            # all, including exited ones
$ podman logs --tail 3 -f ticker
$ podman run --rm -it docker.io/library/alpine:3.20 sh
```

- `-d` (*detached*): prints the container ID and returns
- the engine keeps stdout/stderr of PID 1 → `podman logs`
- `--name`: otherwise a random name (`nostalgic_turing`)
- `--rm`: remove the container when it exits

Container Lifecycle: It Lives as Long as Its PID 1



- PID 1 (`Entrypoint` + `Cmd`) exits ⇒ all its processes are killed
- servers run in the foreground; `ubuntu` without `-it`: `bash` reads EOF → `Exited (0)`
- `stop`: `SIGTERM`, `SIGKILL` after 10 s; `kill`: `SIGKILL`; `start`: the same container
- exit `0` OK, `1` app error, `137` = 128+`SIGKILL` (kill, OOM); `--init`: tiny init as PID 1

Lab: Module 2, Ex. 2.2
Adapted from `https://container.training/` **(J. Petazzoni et al., CC-BY)**

Fully Qualified Image Names (FQIN)

Registries serve images: `docker.io` (Docker Hub; `library/` = official images), `quay.io`, `ghcr.io`, private ones…



Short names are a trap: what is `postgres`?

- **Docker**: silently `docker.io/library/postgres:latest`
- **Podman**: a known alias (`000-shortnames.conf`), else a prompt in a terminal, an error in scripts
- whoever owns that name on a searched registry wins

Rule: always write the full name.

Lab: Module 2, Conceptual foundation

Tags Are Mutable, Digests Are Not: Pin Your Versions

Git	Images	
branch	tag `:latest`, `:16`, `:16-alpine`	moves with each release
tag	tag `:16.4-alpine`	should not move (but can!)
commit hash	digest `@sha256:4f6e...`	immutable: hash of the manifest

- a tag is a pointer the maintainer can move; `latest` = the default name, not "the newest"
- risk: `postgres:latest` is now 18, with a new data directory: your old volume no longer matches
- pin: `postgres:16.4-alpine` (production), `postgres@sha256:...` (reproducible builds)
- digest: `podman image inspect --format '{{.Digest}}' IMG`; upgrade on purpose

Lab: Module 2, Conceptual foundation

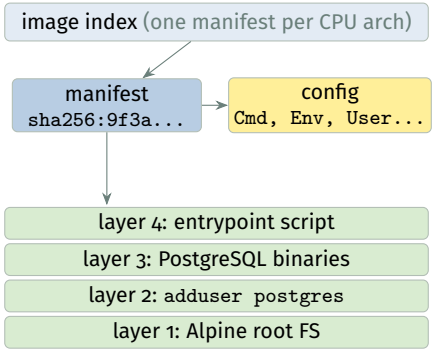
What Is a Container Image, Really?

Not a VM disk, but:

- layers: `tar`s of file changes
- a config: what to run, and how
- a manifest: lists them by `sha256`

Standard format: the OCI image spec

See the layers: `podman image tree`, `podman history` (details: appendix)



stacked by OverlayFS (lecture 3)

https://github.com/opencontainers/image-spec

Auditing an Image: The Fields to Check

```
$ podman image inspect --format '{{json .Config}}' \
      docker.io/library/postgres:16-alpine | jq .
```

Config field	Containerfile	go-httpbin	postgres
`ExposedPorts`	`EXPOSE`	`8080/tcp`	`5432/tcp`
`Volumes`	`VOLUME`	—	`/var/lib/postgresql/data`
`User`	`USER`	`0` (root)	empty: root, then UID 70
`Entrypoint`	`ENTRYPOINT`	—	`docker-entrypoint.sh`
`Cmd`	`CMD`	`/bin/go-httpbin`	`postgres`
`Env`	`ENV`	`SSL_CERT_FILE`	`PGDATA`, `PG_MAJOR`
`StopSignal`	`STOPSIGNAL`	(`SIGTERM`)	`SIGINT`

`ExposedPorts` → your `-p`; `Volumes` → `-v`; `User` → file owners; `Entrypoint` + `Cmd` = PID 1 (`podman run IMG a b` replaces `Cmd`). All overridable (`--user`, `--entrypoint`…).

Lab: Module 2, Ex. 2.1

Diagnosing a Container That Exits Immediately

```
$ podman run -d --name broken-db docker.io/library/postgres:16-alpine
$ podman ps                                          # empty!
$ podman ps -a --format '{{.Names}} {{.Status}}'
broken-db Exited (1) 3 seconds ago
$ podman logs broken-db
Error: Database is uninitialized and superuser password is not
       specified. You must specify POSTGRES_PASSWORD...
```

- `podman ps` hides exited containers: check `ps -a`
- `logs` works on stopped containers: first reflex when it fails

Resource limits (`--memory`, `--cpus`…): appendix; their cgroups: lecture 3.

Lab: Module 2, Ex. 2.2

Configuring with `-e`, Entering with `exec`

```
$ podman rm broken-db
$ podman run -d --name working-db \
      -e POSTGRES_USER=esir -e POSTGRES_PASSWORD=secretpass \
      -e POSTGRES_DB=coursedb \
      docker.io/library/postgres:16-alpine
$ podman exec -it working-db psql -U esir -d coursedb
$ podman exec -it working-db sh        # interactive shell
$ podman diff working-db               # files changed vs image
$ podman rm -f working-db              # done with it: kill + remove
```

- `-e KEY=VALUE`: environment of PID 1; shown by `podman inspect` (secrets: see Compose)
- `exec`: a new process in the same namespaces and cgroup; no SSH needed
- only the image's tools: *distroless* has no shell

Lab: Module 2, Ex. 2.2

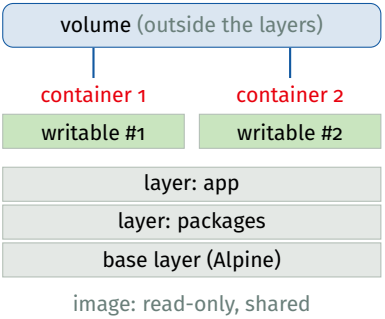
Adapted from `https://container.training/` **(J. Petazzoni et al., CC-BY)**

The Writable Layer Is Ephemeral

- writes go to a thin writable layer (OverlayFS)
- `podman rm` ⇒ it is deleted
- copy-on-write is slow for databases

⇒ **keep data *outside***

- named volumes (engine-managed)
- bind mounts (host directory)



Named Volumes and Bind Mounts

```
$ podman run -d -e POSTGRES_PASSWORD=secretpass \
      -v pgdata:/var/lib/postgresql/data \
      docker.io/library/postgres:16-alpine          # named volume
$ podman run -d -v ~/site:/usr/share/nginx/html:ro,Z \
      docker.io/library/nginx:1.27                  # bind mount
```

`-v SRC:DST[:OPTS]`: `SRC` without `/` = a volume name, otherwise a host path

Named volume (`pgdata`)

- engine-managed: `podman volume ls` / `inspect` / `rm`
- survives `podman rm`
- filled from the image (content + owner): rootless-friendly
- databases, app state

Bind mount (`~/site`)

- any host path, edited from the host
- host owners and modes as is → rootless trap (next)
- SELinux: `:Z` (private), `:z` (shared); `:ro`: read-only
- source code, config files

Rootless volumes live in `~/.local/share/containers/storage/volumes/`

Rootless Storage: The Permission Trap

Symptom: writing into a bind-mounted directory → `Permission denied` (e.g. `redis` as UID 999, in `/data`)

Cause: container UID 999 = host UID $100000 + 999 - 1 = 100998$ (`/etc/subuid`: `me:100000:65536`); the directory belongs to you (1000) = container `root`: 100998 is "others" (`r-x`)

Three fixes

```
$ podman run -v ~/redis-bind:/data:U ...     # 1. Podman chowns it
$ podman unshare chown 999:999 ~/redis-bind  # 2. chown in your userns
$ podman run -v redis-data:/data ...         # 3. named volume
```

SELinux (Fedora, RHEL): add `:Z` (`:z` if shared): `-v ~/redis-bind:/data:Z,U`

Worked example: appendix.

Lab: Module 3, Ex. 3.1

Publishing Ports

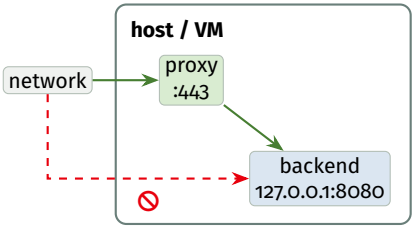
```
$ podman run -d --name web -p 8080:80 docker.io/library/nginx:1.27
$ curl http://localhost:8080/
$ podman port web
80/tcp -> 0.0.0.0:8080
```

- `-p HOST:CONTAINER`: host port 8080 → container port 80
- without `-p`: unreachable from outside the host
- `EXPOSE` / `ExposedPorts`: metadata only, publishes nothing…
- …except `-P`: *all* exposed ports, on random host ports
- rootless: host ports < 1024 are privileged

Lab: Module 3, Ex. 3.2

Port Binding Security: `0.0.0.0` vs `127.0.0.1`

- `-p 8080:8080` binds all interfaces (`0.0.0.0`): on a shared VM, your backend is public
- behind a local reverse proxy, bind to loopback only




```
$ podman run -d --name http-echo -p 127.0.0.1:8080:8080 \
      docker.io/mccutchen/go-httpbin:v2.15.0
$ curl 127.0.0.1:8080/ip && podman port http-echo
```

Firewall: rootless Podman (pasta) ports are ordinary sockets, the host firewall applies; rootful Docker's own `iptables` rules bypass `ufw`/`firewalld`.

Lab: Module 3, Ex. 3.2

User-Defined Networks and DNS

```
$ podman network create esir-net
$ podman network inspect esir-net     # subnet 10.89.0.0/24, DNS on
$ podman run -d --name backend-api --network esir-net \
      docker.io/mccutchen/go-httpbin:v2.15.0          # no -p!
$ podman run --rm --network esir-net \
      docker.io/library/alpine:3.20 \
      wget -qO- backend-api:8080/headers
{ "headers": { "Host": ["backend-api:8080"], ... } }
```

- the container name is its DNS name (not on the default network; replaces `--link`)
- reachable on its network, invisible from outside the host

Podman: netavark (bridges, firewall) + aardvark-dns; rootless: a private namespace linked out by pasta, container IPs unreachable from the host (appendix)

Lab: Module 3, Ex. 3.2

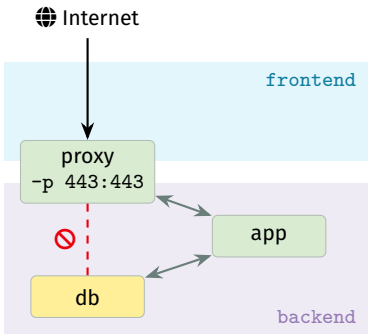
Networks: Who Can Talk to Whom

Simple rule: containers talk only if they share a network.

Segmentation

- one network per role or per stack
- only the reverse proxy publishes ports
- backends, databases: no `-p`
- a hacked proxy cannot reach the DB

(Compose does this for you)



Lab: Module 3, Ex. 3.2

Containerfile (a.k.a. Dockerfile)

A **build recipe**, run top to bottom. `Containerfile` = `Dockerfile` (same syntax); `podman build -t myimage:1.0 .` ≡ `docker build`

- each `RUN`, `COPY`, `ADD` = an immutable layer (snapshot)
- `ENV`, `CMD`, `EXPOSE`…: metadata only (0 B)
- layers are **cached and reused** between builds

```
$ podman history esir-app:naive                (shortened)
CREATED BY                                     SIZE
/bin/sh -c #(nop) CMD ["/app"]                 0 B
/bin/sh -c go build -o /app main.go            7.8 MB
/bin/sh -c #(nop) COPY file:... .              1.5 kB
(golang:1.23-alpine layers)                    250 MB
```

Anti-pattern: set up a container by hand + `commit`: not reproducible.

Lab: Module 4, Ex. 4.1

Build, Run, Push, Pull

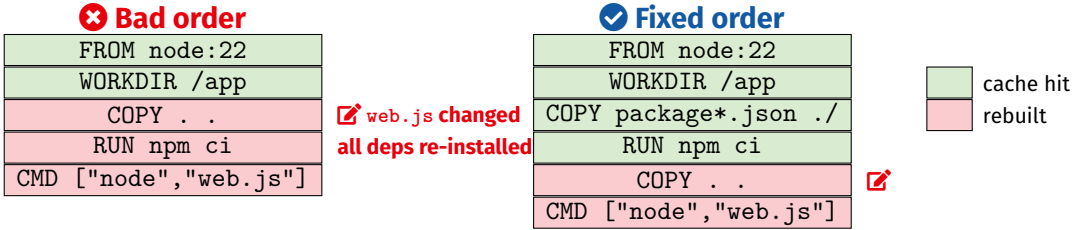
```
# Build an image from ./Containerfile (or ./Dockerfile)
podman build -t quay.io/esir/web:1.0 .
# Run it: host 127.0.0.1:8080 -> container port 3000
podman run -d -p 127.0.0.1:8080:3000 quay.io/esir/web:1.0
# Push to a registry                              (developer)
podman login quay.io
podman push quay.io/esir/web:1.0
# Pull it anywhere else                           (operations)
podman pull quay.io/esir/web:1.0
```

The image name says where it is pushed (`podman tag SRC NEW` adds one); no registry prefix: `-t web:1.0` → `localhost/web:1.0`.

Docker: same sub-commands.

The Build Cache: Order Matters

A layer is **reused** if same parent + same instruction: `RUN` compares the text only (`apt-get update` is not re-run!), `COPY`/`ADD` a **checksum of the files**. After the first miss, every following step is rebuilt. (`--no-cache`: rebuild all.)



Dependency manifests first (`package*.json`, `go.mod`, `pom.xml`…), source code last.

(`node:22` = `docker.io/library/node:22`)

Lab: Module 4, Ex. 4.1

Deleting a File Later Does Not Shrink the Image

Instruction	Layer size	Image size
`FROM docker.io/library/debian:13-slim`	base	base
`RUN apt-get install -y gcc`	+200 MB	base + 200 MB
`RUN apt-get purge -y gcc`	≈ 0 (whiteouts)	base + 200 MB

Lower layers are **immutable**: a deletion only adds whiteouts. The files are still shipped and extractable: a secret `COPY`'d then `rm`'d is still there. `RUN chown`/`chmod`/`mv`: the whole file is copied again.

Fix 1: create and delete in the same `RUN`

```
RUN curl -fsSLO https://example.org/foo.tgz \
 && tar -xzf foo.tgz -C /usr/local \
 && rm foo.tgz
```

Fix 2: multi-stage builds

build in a "fat" stage, `COPY` only the result: build tools never enter the final image

Lab: Module 4, Ex. 4.1

Adapted from `https://container.training/` **(J. Petazzoni et al., CC-BY)**

The Naive Build (Anti-Pattern)

`main.go`: a tiny web server (Go standard library; port from `$PORT`; code: appendix).
`Containerfile.naive`:

```
FROM docker.io/library/golang:1.23-alpine
WORKDIR /src
COPY main.go .
RUN go build -o /app main.go
EXPOSE 3000
CMD ["/app"]
```

```
$ podman build -t esir-app:naive -f Containerfile.naive .
```

It works, but the image:

- weighs more than 250 MB for a program of a few MB
- ships the Go compiler, a shell, `apk`; runs as root (no `USER`)

Lab: Module 4, Ex. 4.1

The Production Multi-Stage Build

Each `FROM` starts a stage; only the last one becomes the image.

```
# Stage 1: build environment, discarded afterwards
FROM docker.io/library/golang:1.23-alpine AS builder
WORKDIR /src
COPY main.go .
RUN CGO_ENABLED=0 GOOS=linux \
    go build -ldflags="-s -w" -o /app main.go
# Stage 2: minimal runtime image
FROM docker.io/library/alpine:3.20
RUN adduser -D -u 10001 appuser
USER 10001
COPY --from=builder /app /usr/local/bin/app
ENV PORT=3000
EXPOSE 3000
ENTRYPOINT ["/usr/local/bin/app"]
```

Lab: Module 4, Ex. 4.1

The Production Multi-Stage Build

Each `FROM` starts a stage; only the last one becomes the image.

```
# Stage 1: build environment, discarded afterwards
FROM docker.io/library/golang:1.23-alpine AS builder ← stage name
WORKDIR /src
COPY main.go .
RUN CGO_ENABLED=0 GOOS=linux \      ← pure Go: a static binary, no C library
    go build -ldflags="-s -w" -o /app main.go ← strip debug info
# Stage 2: minimal runtime image
FROM docker.io/library/alpine:3.20      ← ≈ 8 MB base (still a shell + apk)
RUN adduser -D -u 10001 appuser         ← unprivileged account, no password
USER 10001                              ← numeric UID: no /etc/passwd lookup
COPY --from=builder /app /usr/local/bin/app ← only the binary
ENV PORT=3000
EXPOSE 3000
ENTRYPOINT ["/usr/local/bin/app"]       ← exec form: PID 1, gets SIGTERM
```

Lab: Module 4, Ex. 4.1

Compare the Two Images: Size and Attack Surface

```
$ podman build -t esir-app:prod -f Containerfile.prod .
$ podman images esir-app                        (shortened)
REPOSITORY          TAG     SIZE
localhost/esir-app  prod    ~10MB
localhost/esir-app  naive   >250MB
```

	`:naive`	`:prod`	**With an RCE bug, an attacker…**
Size	> 250 MB	≈ **10 MB**	(pull time, disk, CVEs to patch)
Go compiler + sources	yes	no	naive: uses the bundled compiler
Shell, package manager	yes	yes (Alpine)	opens a shell, `apk add`s miners…
Runs as	root	**UID 10001**	prod: can't install or alter system files

Rootless: UID 10001 is an **unprivileged subuid** on the host. Further: `gcr.io/distroless/static` (no shell) or `FROM scratch`.

Lab: Module 4, Ex. 4.1

Containerfile Checklist

- ✔ **Pin** FQ base images (tag, or better, digest)
- ✔ **Dependency manifests first**, source code last
- ✔ `update && install && cleanup` in one `RUN`
- ✔ No secrets in `COPY`, `ARG`, `ENV` (history shows them)
- ✔ **Multi-stage**: build tools stay behind
- ✔ Run as a **non-root**, numeric `USER`
- ✔ **Exec form** for `ENTRYPOINT`/`CMD`
- ✔ `COPY --chown=10001`, not `RUN chown`
- ✔ Configure via `ENV`, override with `-e`

Start simple, optimize when it matters. All instructions, build secrets, signing: appendix.

Build context: the `.` in `podman build -t web:1.0 .`
the only files `COPY`/`ADD` see, sent to the builder; `-f`: another Containerfile

`.containerignore` (or `.dockerignore`)

```
.git
node_modules
*.log
.env
```

`.gitignore` syntax: faster builds, secrets stay out of the image

Compose

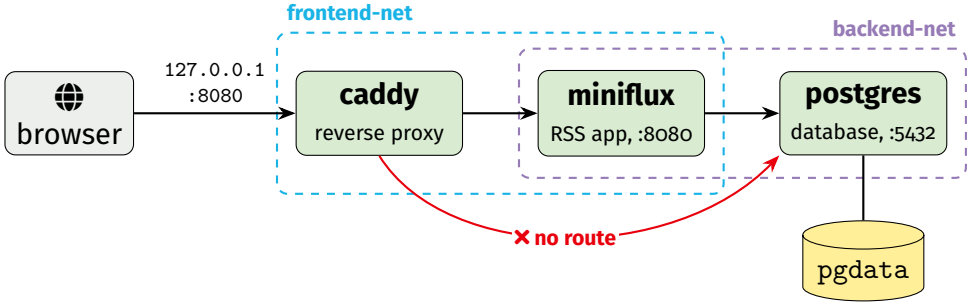
One container = one concern (one main process): an app = several containers, each with its own options, connected together.

Compose: define and run **multi-container applications** on one host

- one YAML file, `compose.yaml`: `services`, `networks`, `volumes`
- service keys = `podman run` options (`ports`: `-p`, `volumes`: `-v`…)
- **declarative**: `podman compose up` converges to the file
- open standard: the **Compose Specification**
  - `docker compose` (Go plugin)
  - `podman compose` (wrapper: `podman-compose` or `docker-compose`)

`version:` is obsolete. The *project name* (directory name) prefixes everything: `selfhosted-stack_pgdata`.

Lab Capstone: A Self-Hosted RSS Reader



- Only `caddy` publishes a port, and only on **loopback**
- `caddy` cannot reach `postgres`: they share no network
- Data lives in the **named volume** `pgdata`

Lab: Module 4, Ex. 4.2

Configuration Next to `compose.yaml`

`Caddyfile`

```
:8080 {
  reverse_proxy miniflux:8080
}
```

- listen on 8080 (rootless: port > 1024)
- forward to `miniflux` **by name** (network DNS)
- the place for TLS, auth, rate limits

`.env` (read by Compose)

```
DB_USER=miniflux
DB_PASSWORD=<random secret>
DB_NAME=miniflux
ADMIN_USER=admin
ADMIN_PASS=<another secret>
```

```
chmod 600 .env     # owner only
```

- `${DB_USER}` in `compose.yaml` ← value
- never commit it (`.gitignore`)
- values still visible: see *Secrets*

Lab: Module 4, Ex. 4.2

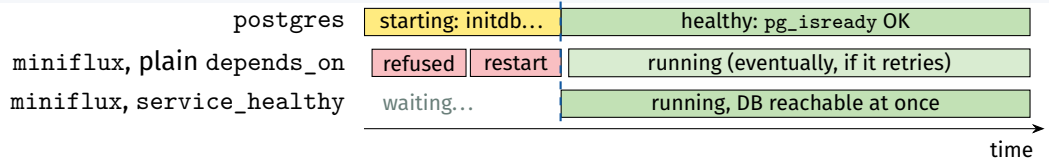
`compose.yaml` (1/4): The Database

```yaml
services:
  postgres:
    image: docker.io/library/postgres:16-alpine
    container_name: stack-postgres
    restart: unless-stopped                # restart policy
    environment:                           # values from .env
      POSTGRES_USER: ${DB_USER}
      POSTGRES_PASSWORD: ${DB_PASSWORD}
      POSTGRES_DB: ${DB_NAME}
    volumes:
      - pgdata:/var/lib/postgresql/data    # NAMED volume
    networks:                              # no "ports:" at all
      - backend-net
    mem_limit: 256m                        # cgroup memory.max
```

Lab: Module 4, Ex. 4.2

`compose.yaml` (2/4): Health Checks, Started vs Healthy

```yaml
postgres:
  healthcheck:      # runs INSIDE the container; exit 0 = healthy
    test: ["CMD-SHELL", "pg_isready -U ${DB_USER} -d ${DB_NAME}"]
    interval: 5s    # run every 5 s
    timeout: 3s     # slower = a failure
    retries: 5      # 5 in a row = unhealthy (podman ps)
```



Running ≠ ready: `depends_on` alone only waits for started.

Containerfile `HEALTHCHECK`: dropped by `podman build` (OCI) unless `--format docker`. Declare it here, or Quadlet `HealthCmd=`.

Lab: Module 4, Ex. 4.2

`compose.yaml` (3/4): The Application

```yaml
  miniflux:
    image: docker.io/miniflux/miniflux:latest   # pin it in prod
    depends_on:
      postgres:
        condition: service_healthy      # wait for pg_isready
    environment:
      DATABASE_URL:
    ↪ postgres://${DB_USER}:${DB_PASSWORD}@postgres/${DB_NAME}?sslmode=disable
      RUN_MIGRATIONS: "1"      # + CREATE_ADMIN, ADMIN_USERNAME...
    networks:
      - backend-net                     # to reach postgres
      - frontend-net                    # to be reached by caddy
    mem_limit: 128m
```

`@postgres`: the **service name**, resolved by the network's DNS (details: appendix)

Lab: Module 4, Ex. 4.2

`compose.yaml` (4/4): The Proxy, Volumes, Networks

```yaml
  caddy:
    image: docker.io/library/caddy:2-alpine
    depends_on:
      - miniflux            # only "started"
    ports:
      - "127.0.0.1:8080:8080"
    volumes:
      - ./Caddyfile:/etc/caddy/Caddyfile:ro,Z
    networks:
      - frontend-net
    mem_limit: 64m
```

```yaml
volumes:
  pgdata:

networks:
  frontend-net:
  backend-net:
  # internal: true
```

`127.0.0.1:8080:8080`: the **only** published port, loopback only.
`:ro,Z`: read-only **bind mount**, relabelled for SELinux.
`internal: true` (optional): `backend-net` gets no route out.

Lab: Module 4, Ex. 4.2

Running the Stack

```
cd ~/lab-data/selfhosted-stack
podman compose up -d                # networks, volume, containers
podman compose ps                   # this project's containers
podman compose logs -f miniflux     # follow one service
curl -i http://127.0.0.1:8080/login    # through caddy
podman compose up -d    # after an edit: re-creates changed services*
podman compose down     # stop + remove containers and networks
```

`restart:` when PID 1 exits

`no` (default), `on-failure`, `always`, `unless-stopped`. Needs a watcher: `dockerd`, `conmon` (Podman)… while your session lives (lecture 3)

`mem_limit:` ⇒ cgroup v2 `memory.max`

Over it: OOM killer (exit 137 = 128 + 9); a runaway service cannot starve the VM (also `cpus:`, `pids_limit:`)

* Provider-dependent: check `podman compose version`.

Lab: Module 4, Ex. 4.2

What Survives `compose down`?

	`down`	`down -v`
Containers	removed	removed
Networks of the project	removed	removed
Named volumes (`pgdata`)	**kept**	removed
Bind-mounted files (`./Caddyfile`)	kept	kept
Images	kept	kept

- Containers are **disposable**; the state lives in the **volume**
- `down` then `up -d`: new containers, same data
- `down -v` is the "factory reset": the database is gone

Lab: Module 4, Ex. 4.2
Adapted from `https://container.training/` **(J. Petazzoni et al., CC-BY)**

Secrets: Keep Them Out of the Environment

Environment variables (`-e`, `environment:`, `.env` values) are easy to read:

```
$ podman inspect stack-postgres --format '{{.Config.Env}}'
[... POSTGRES_PASSWORD=s3cr3t ...]          # also in /proc/1/environ
$ printf '%s' "$DB_PASSWORD" | podman secret create db_pw -
$ podman run --secret db_pw ...     # a FILE: /run/secrets/db_pw
```

Compose: a `secrets:` file, and the image's `_FILE` variant

```
  postgres:
    secrets: [db_pw]
    environment:
      POSTGRES_PASSWORD_FILE: /run/secrets/db_pw
secrets:
  db_pw: {file: ./db_pw.txt}        # chmod 600, not in git
```

Quadlet (lecture 3): `Secret=db_pw,type=env,target=POSTGRES_PASSWORD` (hidden from `inspect`)

`.env` + `chmod 600` only hides the *file* from other users. Recipes: the field guide.

Review This File: Find 4 Problems

```yaml
services:
  redis:
    image: 'redislabs/redismod'
    ports:
      - '6379:6379'
  web1:
    restart: on-failure
    build: ./web
    hostname: web1
    ports:
      - '81:5000'
```

```yaml
  web2:
    restart: on-failure
    build: ./web
    hostname: web2
    ports:
      - '82:5000'
  nginx:
    build: ./nginx
    ports:
    - '80:80'
    depends_on:
    - web1
    - web2
```

First: which services are deployed? What is `nginx` for?

Review: The Answers

Reading it: 4 services; `web1`, `web2`: the same image (`./web`); `nginx`: a **load balancer** (custom config) reaching them as `http://web1:5000` (DNS)

1. `redislabs/redismod`: short name, no tag (`:latest`) → FQIN, pinned version
2. every port published on `0.0.0.0`, Redis included → only the proxy needs `ports:`; separate networks
3. port 80: refused to rootless Podman (< 1024) → e.g. `127.0.0.1:8080:80`
4. plain `depends_on`: started, not ready → `healthcheck` + `condition: service_healthy`

Also: no volume for Redis data, no `mem_limit`.

Lecture 2: What to Remember

- **Images**: FQIN + pinned tags (or digests); a container lives as long as its PID 1
- **Data**: the writable layer is ephemeral → volumes; rootless permission trap → `:U`, `:Z`
- **Networking**: publish on `127.0.0.1`; user-defined networks + DNS
- **Builds**: cache order, multi-stage, non-root `USER`
- **Compose**: one file per stack, health checks, secrets out of the environment

→ Next time: Under the Hood and Production lab Modules 0–1 and 5

namespaces, cgroups, security, OverlayFS; linger, Quadlets (systemd services)