

Container

Build and Deploy -- aber wie?

Frank Engel

August 2026

Day 1 — Theory & Fundamentals

Today:

- Why Containers?
- How do Containers work?
- Docker vs. Podman
- Your first Containers

Goal of the Day:

You understand what containers are, how they differ from VMs, and can independently start and inspect containers.

Introduction

Frank Engel

- ATIX AG — ~10 years
- Cartographer by trade
- 25+ years Linux
- Containers → Kubernetes
- Loves large data centres

The human behind the slides

- Cat dad
- Father of two
- Loves to travel
- Has been poking at containers long enough to know where they bite



- Open Source focused IT company
- Consulting, support, training
- Red Hat, SUSE, Foreman/Katello
- Based in Munich

www.atix.de



Orcharhino — ATIX's enterprise management platform:

- Lifecycle management for RHEL, Debian, SUSE
- Patch management, provisioning, configuration
- Built on Foreman + Katello

Introductory Round

- Who are you?
- Why are you working with containers?
- What is your company role?
- What is your level of experience with containers and Linux in general?

When shall we have breaks?

Containerization has consequences:

- full-stack knowledge really helpful
- ephemeral instances:
 - ip management?
 - data persistence?
- rights management
- and many positive aspects as well

- Container tools: Podman, Docker, and the OCI specifications
- Immutable Design

Virtualisation Basic Terms

- Bare Metal: processes directly on host OS/hardware
- VMs: hypervisor + guest OS with its own kernel
- Containers: OS-level isolation, share the host kernel, isolate processes/resources

Excursion: Not Hype — an Evolution

- 1979** chroot — Unix V7: process gets its own root directory
- 2000** FreeBSD Jails: first real process isolation
- 2002** Linux Mount Namespaces (Kernel 2.4)
- 2008** cgroups (Google → Linux Kernel 2.6): resource limits
- 2008** LXC: first Linux container tooling
- 2013** Docker: UX + ecosystem around the same kernel features
- 2015** OCI: standardisation (Image, Runtime, Distribution)
- 2018** Podman: daemonless, rootless-first

The kernel has had this for decades. The tooling got better.

Excursion: chroot — the Ancestor

chroot (1979) changes the root directory for a process and its children:

Minimal chroot example

```
mkdir -p /tmp/jail/bin
```

```
cp /bin/bash /tmp/jail/bin/
```

```
chroot /tmp/jail /bin/bash
```

→ bash runs, only sees /tmp/jail as /

What chroot can do: filesystem isolation

What chroot cannot do:

- No network isolation
- No process isolation (PID namespace)
- No resource limits
- Root can escape

⇒ Namespaces + cgroups close these gaps — that is a modern container.

Excursion: Namespaces + cgroups = Container

The Linux kernel isolates via **namespaces** (what a process sees):

Namespace	isolates	since Kernel
mnt	filesystem mounts	2.4 (2002)
pid	process IDs	2.6.24 (2008)
net	network interfaces	2.6.29 (2009)
user	UIDs/GIDs	3.8 (2013)

cgroups limit what a process may consume (CPU, RAM, I/O).

Container = chroot + Namespaces + cgroups + Image-Format + Tooling

Containers Evolution

- Bare Metal / VM: strong isolation, but heavy (own kernel per VM)
- Containers / LXC: lightweight, fast startup, higher density
- Docker popularised containers + tooling; OCI standards enable tool choice
- Rootless: unprivileged operation via user namespaces — reduced risk
 - Docker: daemon-based; rootless mode available
 - Podman: daemonless, rootless-first, CLI-compatible

VMs vs. Containers

VMs:

- Hypervisor, dedicated guest OS per VM
- Higher startup time, larger resource footprint
- Very strong isolation; heterogeneous OS possible

Containers:

- Share host kernel, isolate via kernel features
- Very fast startup, resource-efficient
- Good isolation, kernel hardening important

VMs vs. Containers



- Every house has its own foundation, walls, utilities
- More land, more materials
- Takes long to construct
- Private space, separate address



- Apartments share structure, utilities
- Many homes in one building
- “Copy/paste” more units easily
- Walls separate, but same building services

Apropos: Immutable Design

Virtualization:

- Spinning up an environment is expensive
- Treat environment *like a bare metal server*
- Fix problems without rebuilding
- Install updates, patches, etc.

Containerization:

- Spinning up an environment is cheap
- Treat environment as *immutable*
- Problems? → rebuild / redeploy
- Use up-to-date images

Benefits of Immutable Design

- **Security:** Avoid unwanted mutations, drop unnecessary privileges.
- **Transparency:** Dockerfiles completely describe images.
- **Repeatability:** Avoid building “snowflakes”.

Best Practices

- Use version control on Dockerfiles.
- Identify images by their checksums.
- Rebuild and -deploy instead of reconfigure.

→ Need for orchestration tooling

When to Use What?

- VMs: strongest isolation, heterogeneous operating systems, legacy workloads
- Containers: microservices, CI/CD, horizontal scaling, high density
- Sandboxed containers: multi-tenant/compliance with extra isolation requirements

Open container initiative

- Founded 2015 founded by Docker and the Linux Foundation
- specifies OCI Image Spec, OCI Runtime Spec and OCI Distribution Spec
- provides runc
- containerd provided by the CNCF

Minimum:

- Manifest
- Configuration
- Layers

Manifest list can bundle single manifests Multi arch images for example

<https://github.com/opencontainers/image-spec/blob/master/spec.md>

Container $\neq$ Image

Image: binary object, container: process

- Create file system structure from image
- Create control groups and namespaces
- Start the process inside namespace/control group
- $\Rightarrow$ Container is up and running

Container Tools: Podman & Docker

Podman (our primary tool):

- Daemonless, rootless-first, native pod concept (K8s-compatible)
- systemd integration via Quadlets
- OCI-compliant; CLI mirrors Docker

Docker (you may know it already):

- Daemon-based (dockerd); industry pioneer since 2013
- BuildKit; Desktop for macOS/Windows (requires license for large companies)

Also in the ecosystem:

- containerd + nerdctl: lightweight engine stack
- CRI-O: K8s-focused runtime (OpenShift, Kubernetes nodes)

Why Podman?

Security:

- No root daemon — no daemon socket to exploit
- Rootless by default → least privilege
- User namespaces isolate processes per user

K8s alignment:

- Native pod concept mirrors K8s pods
- podman generate kube exports to K8s YAML
- No daemon = same philosophy as K8s nodes
- Default runtime in RHEL, Fedora, OpenShift

Learn Podman → understand Kubernetes better.

- Important ingredient: Docker Hub
 - ghcr.io
 - quay.io
- Nowadays multiple implementations of image registries:
 - Harbor
 - Nexus
 - Artifactory
 - ECR, GCR, Azure Container Registry
 - GitLab

- Podman:
 - `podman run -d --name webdemo -p 8080:80 nginx:1.27`
 - `podman inspect webdemo`
 - `podman inspect --format '{{.Name}}: {{.State.Status}} (IP: {{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}})' webdemo`
- Docker:
 - `podman run -d --name webdemo -p 8080:80 nginx:1.27`
 - `podman inspect webdemo`
 - `podman inspect --format '{{.Name}}: {{.State.Status}} (IP: {{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}})' webdemo`

Takeaways

- Containers = processes + isolation, no own kernel
- Rootless + least privilege increase security
- OCI standards allow tool choice (Docker/Podman/...)

Using Docker

- Docker CLI `docker`
- Podman CLI `podman`
- Compatible syntax: `podman` commands mirror `docker` (can alias `docker=podman`)
- Both support OCI images and runtimes (`runc/crun`)

Exercise 1

Connect to your docker host!

- Run a container from the image hello-world

Walk-through

```
> podman run hello-world
```

```
Resolved "hello-world" as an alias (/etc/containers/registries.conf.d/000-shortnames.conf)
```

```
Trying to pull quay.io/podman/hello:latest...
```

```
Getting image source signatures
```

```
Copying blob sha256:1ff9adef4443b503b304e7aa4c37bb90762947125f4a522b370162a7492ff47
```

```
Copying config sha256:83fc7ce1224f5ed3885f6aaec0bb001c0bbb2a308e3250d7408804a720c72a32
```

```
Writing manifest to image destination
```

```
!... Hello Podman World ...!
```

```
      .--"---.
     /  -   -  \
    / (0)   (0) \
   ~~~|  -=(,Y,)=- |
     .---. /`  \   |~~
    ~/  0  0  \~~~~ ~~~~ ~~~
```

Exercise 2

Examine your podman system! Play with the following commands:

- `podman help`
- `podman inspect hello-world`
- `podman images`
- `podman ps -a`

Walk-through

- `podman help`: is an extensive in-line help
- `podman inspect hello-world`: inspects the image hello-world
- `podman images`: lists all present images
- `podman ps -a`: lists all containers (running and stopped)

Commands

- `podman pull` fetches images from registry
- `podman create` creates container from image
- `podman start` starts container
- `podman run` combines the previous commands
- `podman stop/kill` stops containers
- `podman ps [-a]` lists containers
- `podman images` lists present images
- `podman logs` displays container's logs
- `podman help` displays help page

Local documentation: `podman help <COMMAND>`, `man podman <COMMAND>`

Useful during development: `podman exec -it <CONTAINER-NAME> /bin/bash`

Exercise 3

The interior of a container

Run an interactive terminal session in a container, use the image `bash`. Inspect the network interface, name resolution, ...

Walk-through

```
> docker run -ti --rm bash
> ip a
> cat /etc/hosts
> cat /etc/resolv.conf
> mount
> ls -l /dev
> cat /etc/os-release
```

Exercise 4

Autorestart Containers

- Run a webdemo container with
- `podman run -d --name webdemo nginx:1.27`
- `podman ps`
- And exec a shell inside of the container
- `podman exec -ti webdemo bin/bash`
- Kill the main process inside of the container
- `kill 1`
- What happen to the container?
- `podman ps`

Start the Pod in a way that podman will restart it on failure (kill)

Walk-through

```
> podman run --help (| grep restart)
> podman run -d --name webdemo --restart=always nginx:1.27
> podman exec -ti webdemo bin/bash
> kill 1
> podman ps
```

Day 2 — Hands-on: Building an App

Day Overview

Morning:

- Recap: imperative commands
- Networking (imperative)
- Storage & Volumes (imperative)
- Environment Variables
- Compose: same concepts, declarative

Afternoon:

- Container Runtimes
- Building custom images
- User & Permissions
- Security Scanning

Recap: What We Covered Yesterday

- Container basics: images, layers, namespaces, cgroups
- Podman CLI: run, exec, ps, logs, inspect
- Interactive containers, restart policies

Today we fill in the gaps: Networking, Storage, Environment Variables — imperatively first, then the same concepts in a Compose file.

Why Container Network?

- Isolation: each container has its own network namespace (separate IP, routing, firewall rules)
- Communication: containers need to talk to each other and external services
- Portability: abstract networking from host configuration
- Security: control network access per container

Networking modes

- Default: containers on same bridge can communicate; isolated from host by default
- Port mapping (-p): selectively expose container ports to host/external network
- User-defined networks: create isolated network segments for multi-tier apps
- `podman network create <name>`
- **Network Modes Explained**
 - bridge: default, isolated namespace with NAT (most common)
 - host: share host's network stack (no isolation, better performance)
 - none: no networking (for specialized use cases)
 - container:NAME: share another container's network namespace

Inspecting Networks

```
podman network ls                # list all networks
podman network inspect <name>    # details: subnet, gateway, connected containers
podman inspect <container> \
  --format '{{json .NetworkSettings.Networks}}' | jq
```

Container Networking

- Create a network appnet
- Create two containers using `nginx:1.27` — name them frontend and backend
- Connect both to appnet
- Curl from frontend to backend by container name
- Expose frontend on host port 8080

Walk-through

```
podman network create appnet
podman run -d --name backend --network appnet nginx:1.27
podman run -d --name frontend --network appnet -p 8080:80 nginx:1.27
podman network inspect appnet
```

Container-to-container (via DNS in appnet)

```
podman exec -ti frontend curl backend
```

Host-to-container (via Port-Mapping)

```
curl localhost:8080
```

Overview – persistent storage

- Not all services can be stateless.
- Data in containers is ephemeral.

Where do the files live?

Two basic ways, storage and container work together:

- overlayfs
 - union of lower and upper filesystems
 - only upper fs can be writable
 - if files exist in both layers, upper one wins
 - lower layer can be another overlayfs
 - this layer is managed per container by the runtime
 - image → container; runtimes creates ephemeral top layer
- bind-mounts
 - access a directory on the host within a container
 - security restricts accessible paths
 - mounts can be shared between containers

Embedding storage

- Bind Mount
 - mount a local directory /source into the container's /dest
- Volume
 - mount an object into the container's /dest
 - Docker manages volumes independently of file system
- tmpfs
 - stores data locally in memory
 - prevents multiple access

Embedding storage – Bind mounts

podman run ...

- -v /source:/target ...
- obscures directory's contents (of the image)
- multiple access allowed
- mount readonly directory
- -v /source:/target:ro ...

Embedding storage – Bind mounts: Examples

```
# Serve local HTML files with nginx
podman run -d \
  -v ./html:/usr/share/nginx/html:ro \
  -p 8080:80 \
  nginx:1.27
```

```
# Mount config file from host (read-only)
podman run -d \
  -v /etc/myapp/config.yml:/app/config.yml:ro \
  myapp:latest
```

Exercise 6

communication through volume

Run two containers:

- image busybox
- bind-mount the same path in both containers to /data
- command for first container: `while true; do sleep 1; date >> /data/output; done`
- command for second container: `tail -f /data/output`

Walk-through

```
> podman run -d --name writer -v /tmp:/data busybox \  
    sh -c 'while true; do sleep 1; date >> /data/output; done'  
  
> podman run --rm -v /tmp:/data busybox tail -f /data/output  
Fri Nov 06 10:54:41 UTC 2025  
Fri Nov 06 10:54:42 UTC 2025  
Fri Nov 06 10:54:43 UTC 2025  
# Ctrl+C to exit – then: podman stop writer && podman rm writer
```

Embedding storage – Volumes

```
podman volume create myvol podman run ...
```

- `-v myvol:/target ...`
- prepopulated from image on first run if empty
- can mount remote storage via `nfs`, `cifs` or others
- multiple access allowed

Note: this differs a little from how to use volumes in `k8s`.

Inspecting Volumes

```
podman volume ls                # list all volumes
podman volume inspect <name>    # details: mountpoint, labels, size
podman inspect <container> \
  --format '{{json .Mounts}}' | jq
```

Exercise 7

communication through volume

Create a named volume using podman

Run two containers:

- image busybox
- mount the same volume in both containers to /data
- command for first container: `while true; do sleep 1; date >> /data/output; done`
- command for second container: `tail -f /data/output`

Walk-through

```
> podman volume create pizza
> podman run -d --name writer -v pizza:/data busybox \
    sh -c 'while true; do sleep 1; date >> /data/output; done'

> podman run --rm -v pizza:/data busybox tail -f /data/output
Fri Nov 06 11:50:58 UTC 2025
Fri Nov 06 11:50:59 UTC 2025
Fri Nov 06 11:51:00 UTC 2025
# Ctrl+C to exit – then: podman stop writer && podman rm writer
```

Exercise 8

communication through volume

Create a named volume using podman

Create a Volume and run a containers:

- volume create
- image busybox
- create a file inside the container in the mounted volume
- destroy and remove the container
- create a second container with the same volume mount (mount to different location)
- check content of mounted volume inside of the container

Walk-through

```
> podman volume create pasta
> podman run -v pasta:/data --name pasta --rm -ti busybox
> podman exec -ti pasta touch /data/hello
> podman rm --force pasta
> podman run -v pasta:/data --name pasta2 --rm -ti busybox
> podman exec -ti pasta2 ls -la /data
```

Embedding storage – Volumes

- Docker defaults:
 - Linux: overlay2 (requires xfs/ext4 with d_type support)
 - Alternative drivers: btrfs, zfs, vfs
 - Configure via daemon.json ("storage-driver": "overlay2")
- Podman defaults:
 - overlay (uses fuse-overlayfs when running rootless)
 - Configured per user in containers-storage.conf

Embedding storage – tmpfs

podman run ...

- `--tmpfs /target:size=128m ...`
- `volatile`
- `unique` – cannot be shared between containers

Summary and usage hints

- Store persistently needed files in volumes
- Use storage to exchange data
- Do not embed storage manipulation in main container:
 - initialization
 - migrations
 - backup and restore

Passing Configuration to Containers

- Containers should not contain environment-specific config in the image
- Pass config at runtime via environment variables

```
> podman run -e MY_VAR=hello nginx:1.27 env
```

```
> podman run -e DB_HOST=db -e DB_PORT=5432 nginx:1.27 env
```

- `--env-file` for multiple variables:

```
> cat app.env
```

```
DB_HOST=localhost
```

```
DB_PORT=5432
```

```
DB_USER=app
```

```
DB_PASS=secret
```

```
> podman run --env-file app.env nginx:1.27 env
```

Pass environment variables to a container

- Run a postgres:16-alpine container imperatively
- Pass POSTGRES_DB, POSTGRES_USER, POSTGRES_PASSWORD via -e
- Verify the database is reachable: `podman exec -ti <name> psql -U app myapp`

Walk-through

```
> podman run -d --name mydb \  
    -e POSTGRES_DB=myapp \  
    -e POSTGRES_USER=app \  
    -e POSTGRES_PASSWORD=secret \  
    postgres:16-alpine  
  
> podman exec -ti mydb psql -U app myapp -c "\l"  
> podman stop mydb && podman rm mydb
```

What is Docker Compose?

- **Definition**

- Declarative tool for defining and running multi-container Docker applications
- Single YAML file (`docker-compose.yml`) describes entire application stack
- Manages services, networks, volumes, and their relationships

What is Docker Compose?

- **Why Use Docker Compose?**

- Simplifies multi-container orchestration
- Reproducible environments: same file works across dev/staging/prod
- Version control: compose file in Git tracks infrastructure as code
- Service discovery: automatic DNS resolution between services
- Dependency management: start services in correct order

What is Docker Compose?

- **When to Use**

- Local development: run full stack (DB, API, frontend) with one command
- CI/CD: test multi-container applications
- Small deployments: simple production stacks without Kubernetes complexity
- Learning: understand container orchestration concepts

Docker Compose – Core Concepts

- **Services**

- Each service = one container (or multiple with `deploy.replicas`)
- Defined in `services:` section of YAML
- Service name becomes hostname for DNS resolution

- **Networks**

- Default: Compose creates isolated bridge network per project
- Services on same network can communicate by name
- User-defined networks: create custom network topologies

Docker Compose – Core Concepts

- **Volumes**

- Named volumes: managed by Docker, persist across container restarts
- Bind mounts: map host directories into containers
- Anonymous volumes: temporary storage (removed with down -v)

- **Project**

- Project name = directory name (or -p flag)
- All resources prefixed with project name (e.g., myapp_db_1)
- Isolates multiple compose stacks on same host

Docker Compose File Structure

```
services:  
  service-name:  
    image: image:tag  
    # or  
    build: ./path/to/dockerfile  
    ports:  
      - "8080:80"  
    environment:  
      - KEY=value  
    volumes:  
      - volume-name:/path  
    networks:  
      - network-name
```

```
volumes:  
  volume-name:
```

Service Configuration – Common Options

- **Image & Build**

- `image: nginx:1.27`: use pre-built image
- `build: ./app`: build from Dockerfile in ./app
- `build: { context: ., dockerfile: Dockerfile.prod }`: custom Dockerfile

- **Ports**

- `ports: ["8080:80"]`: map host:container ports
- `expose: ["80"]`: expose internally (no host mapping)
- `ports: ["3000-3005:3000-3005"]`: port ranges

Service Configuration – Common Options

- **Environment Variables**

- `environment: { KEY: value }`: inline definition
- `env_file: .env`: load from file

- **Volumes**

- `volumes: ["appdata:/var/lib/app"]`: named volume
- `volumes: ["/data:/data"]`: bind mount
- `volumes: ["/data"]`: anonymous volume

Dependencies & Startup Order

- **depends_on**
 - Ensures services start in order (but doesn't wait for readiness)
 - Example: backend: depends_on: [db] → db starts before backend
 - Limitation: only controls startup order, not health
- **Healthchecks & Condition-Based Dependencies**
 - Define healthcheck in service:

```
healthcheck:  
  test: ["CMD", "curl", "-f", "http://localhost:8080/health"]  
  interval: 10s  
  timeout: 5s  
  retries: 3
```
 - Wait for health: depends_on: { db: { condition: service_healthy } }

Networking in Docker Compose

- **Default Network**

- Compose creates one bridge network per project
- All services automatically join this network
- Services can reach each other by service name (DNS)

- **User-Defined Networks**

```
networks:
```

```
  frontend:
```

```
  backend:
```

```
services:
```

```
  web:
```

```
    networks: [frontend]
```

```
  api:
```

```
    networks: [frontend, backend]
```

```
  db:
```

```
    networks: [backend]
```

Volumes in Docker Compose

- **Named Volumes**

```
volumes:
```

```
  db-data:
```

```
services:
```

```
  db:
```

```
    volumes:
```

```
      - db-data:/var/lib/postgresql/data
```

- Managed by Docker, persist across down (unless -v flag)
- Location: /var/lib/docker/volumes/ (or Podman equivalent)

Volumes in Docker Compose

- **Bind Mounts**

```
services:
  app:
    volumes:
      - ./src:/app/src
      - ./config:/app/config:ro # read-only
```

- Map host directories into containers
- Useful for development (live code reload)

- **Anonymous Volumes**

- Temporary volumes, removed with `down -v`
- Use when you don't need persistence

Environment Variables & Secrets

- **Environment Variables**

- Inline: `environment: { DB_HOST: db }`
- From file: `env_file: .env`
- Variable substitution: `environment: { DB_PASS: ${DB_PASSWORD} }`
- Precedence: CLI `env` > `.env` file > compose defaults

- **Secrets (Swarm Mode)**

- For production: use Docker Swarm secrets or external secret managers
- In Compose: use `.env` file (exclude from Git!)
- Best practice: `.env.example` template, `.env` in `.gitignore`

- **Example `.env`**

```
DB_PASSWORD=secret123
```

```
API_KEY=abc123
```

Docker Compose Commands

- **Lifecycle**

- `podman compose up`: start services (foreground)
- `podman compose up -d`: start in detached mode
- `podman compose down`: stop and remove containers
- `podman compose down -v`: also remove volumes

- **Management**

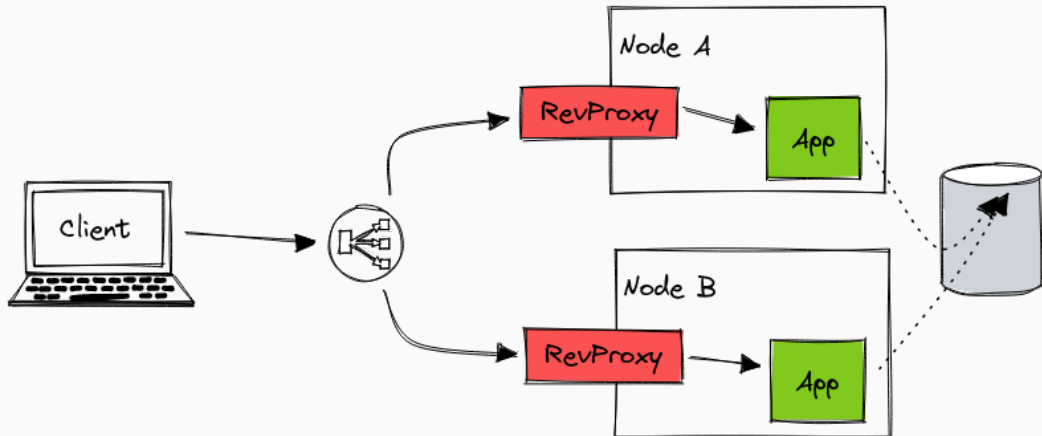
- `podman compose ps`: list running services
- `podman compose logs -f`: follow logs from all services
- `podman compose logs -f service-name`: logs from one service
- `podman compose restart service-name`: restart one service

- **Development**

- `podman compose up service-name`: start only one service (and dependencies)
- `podman compose exec service-name sh`: exec into running container
- `podman compose build`: rebuild images
- `podman compose pull`: pull latest images

Reverse Proxy, Load Balancer

- Typically one or more layers between application container and internet
- Common choices: HAProxy, F5, NGinx, Kong, Traefik



Exercise 10

Create a `docker-compose.yml` with:

- A PostgreSQL database service
- A simple web service (nginx) that depends on the database
- A custom network for isolation

Walk-through

```
services:
  db:
    image: postgres:16-alpine
    environment:
      POSTGRES_DB: myapp
      POSTGRES_USER: app
      POSTGRES_PASSWORD: secret
    networks:
      - backend
  web:
    image: nginx:1.27-alpine
    ports:
      - "8080:80"
    depends_on:
      - db
    networks:
      - backend
networks:
  backend:
```

Walk-through (cont.)

```
podman compose up -d  
podman compose ps  
podman compose logs db  
podman compose down
```

Exercise 11

podman compose Volume

Add volume persistence to the database service and verify data survives container removal.

Walk-through

```
services:
  db:
    image: postgres:16-alpine
    environment:
      POSTGRES_DB: myapp
      POSTGRES_USER: app
      POSTGRES_PASSWORD: secret
    volumes:
      - db-data:/var/lib/postgresql/data
  networks:
    - backend
web:
  image: nginx:1.27-alpine
  ports:
    - "8080:80"
  depends_on:
    - db
  networks:
    - backend
volumes:
```

Walk-through (cont.)

```
podman compose up -d
podman compose exec db psql -U app myapp -c "CREATE TABLE test (id serial);"
podman compose down      # removes containers, keeps volume
podman compose up -d
podman compose exec db psql -U app myapp -c "\dt"    # table still there
podman compose down -v   # now also removes volume
```

Fundamentals: Docker and Podman

What are Container Runtimes?

High-level runtimes:

- containerd (Docker's default)
- CRI-O (Kubernetes-native)

Low-level runtimes:

- runc (reference OCI implementation)
- crun (faster, C-based alternative)
- gVisor (sandboxed runtime)

- Industry standard high-level runtime
- Used by Docker, Kubernetes
- Manages container lifecycle
- Image distribution and storage
- Works with various low-level runtimes

- Reference implementation of OCI runtime spec
- Written in Go
- Creates and runs containers
- Used by containerd and CRI-O
- Lightweight and stable

- Fast, lightweight OCI runtime
- Written in C (vs. runc in Go)
- Lower memory footprint
- Faster startup times
- Podman's default on many distros

Performance benefit: ~30% faster container startup

- Sandboxed container runtime
- Additional security layer
- User-space kernel (prevents syscall abuse)
- Some performance overhead
- Good for untrusted workloads

Trade-off: Security vs. Performance

Runtime Comparison

Runtime	Language	Speed	Use Case
runc	Go	Fast	Standard workloads
crun	C	Faster	Performance-critical
gVisor	Go	Slower	High security needs

Build your own images

- simple text file containing instruction how to build an image
- comparable to packaging instructions (but simpler)
- one line for each layer
- typically:
 - copy files into the image
 - run commands

Build process

1. copy files in build folder to build context
2. find FROM in Dockerfile
3. start container from image referenced there (namespace, directory)
4. go through Dockerfile line by line and execute statement
5. save resulting file systems to create layers
6. combine layers and runtime information as container image

Example

Dockerfile:

```
FROM nginx:1.27
```

```
COPY index.html /usr/share/nginx/html/index.html
```

index.html:

```
<html>
```

```
  <head><title>DEMO</title></head>
```

```
  <body>Nothing to see here!</body>
```

```
</html>
```

first start:

```
podman compose up -d
```

```
curl localhost
```

rebuild:

```
podman compose build
```

```
podman compose up -d
```

docker-compose.yml:

```
---
```

services:

nginx:

```
  build: ./
```

```
  ports:
```

```
    - "80:80"
```

Exercise 12

Inspect built image

- Create files as just shown using a simple Dockerfile, index.html and the Docker Compose definition
- Find out the name of the created Docker image!
- Discuss how you could have created that image otherwise

Dockerfile:

```
FROM nginx:1.27
```

```
COPY index.html /usr/share/nginx/html/index.html
```

index.html:

```
<html>
```

```
  <head><title>DEMO</title></head>
```

```
  <body>Nothing to see here!</body>
```

```
</html>
```

docker-compose.yml:

```
---
```

services:

nginx:

```
    build: ./
```

```
    ports:
```

```
      - "80:80"
```

Walk-through

Build and start:

```
> podman compose up -d --build
```

Verification:

```
> podman images <DIRECTORY>_<SERVICENAME>
```

Or build manually:

```
> podman build -t NAME:TAG .
```

```
> podman images
```

Dockerfile contents

- FROM defines the parent image
 - RUN executes commands (batch mode)
 - ADD & COPY copies local files
-
- USER sets the current user
 - ENV defines variables
 - LABEL sets metainformation
 - EXPOSE exposes ports
 - ENTRYPOINT defines the central command
 - CMD defines the central command

Build context and .dockerignore

- Docker copies files from the build context to a cache directory
- .dockerignore allows to exclude files
- Syntax as with .gitignore
- Block- and allow-lists possible

```
# Ignore everything
```

```
*
```

```
# Allow little
```

```
!index.html
```

ADD vs. COPY

ADD/COPY <SRC> <DEST>

Best Practice: use COPY unless

- <SRC> is a URL
- <SRC> points to an archive that is to be expanded

RUN vs. ENTRYPOINT vs. CMD

- RUN creates a layer in the image
- ENTRYPOINT defines the initial command, rather fixed
- CMD defines the initial command, easy to overwrite

-
- SHELL mode: `CMD echo "Hello World" ⇒ sh -c echo "Hello World"`
 - EXEC mode: `CMD ["/bin/echo", "Hello", "World"] ⇒ /bin/echo Hello World`

ENTRYPOINT and CMD can be combined

ENTRYPOINT or CMD

ENTRYPOINT echo "Hello World"

podman run image	"Hello World"
podman run image Joe	"Hello World"

CMD echo "Hello World"

podman run image	"Hello World"
podman run image Joe	"... executable file not found ..."
podman run image echo "Hello Joe"	"Hello Joe"

Combining ENTRYPOINT and CMD

```
ENTRYPOINT ["/bin/echo", "Hello"]
```

```
CMD ["World"]
```

```
podman run image          "Hello World".
```

```
podman run image Joe     "Hello Joe".
```

Notice for use in Kubernetes

ENTRYPOINT is referred to as command, CMD as args.

containers:

- name: busybox

image: busybox

command: ["sleep"] # replaces ENTRYPOINT

args: ["3000"] # replaces CMD

Entrypoints, Services & Daemons

- Container is terminated after any exit signal
- Start daemons or server through ENTRYPOINT or CMD (rather than systemd or init)
- Example from php:7-apache:
`CMD ["apache2-foreground"]`
- Logs go to STDOUT
- Inherited through FROM

```
RUN apt-get update \  
    && apt-get install -y curl=7.61.* \  
    && rm -rf /var/lib/apt/lists/*
```

- Update package source, install and empty cache in one command
- “Cache busting” by explicitly stating version number (also with FROM)
- Avoid upgrade or dist-upgrade!

Multi-Stage Builds

```
FROM golang:1.7.3 as build
WORKDIR /go/src/github.com/alexellis/href-counter/
RUN go get -d -v golang.org/x/net/html
COPY app.go .
RUN CGO_ENABLED=0 GOOS=linux go build -a \
    -installsuffix cgo -o app .
```

```
FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=build \
    /go/src/github.com/alexellis/href-counter/app .
CMD ["/app"]
```

Example slightly modified from: docs.docker.com/develop/develop-images/multistage-build/

Best practices for images builds

- build small images for minimal transport size and attack surface
 - As few layers as possible
 - As small layers as possible
 - Use multi stage builds
 - Minimize the build context (use `.dockerignore`)
 - Avoid unnecessary content
- Prefer slim bases (scratch, distroless, alpine)
- Pin versions
- Sensible layer ordering
- Use unprivileged user with numeric user ID
- Allow users to inject options and parameters

Exercise 13

Create a multi-stage Dockerfile for a simple application

Create a main.go:

```
package main
```

```
import "fmt"
```

```
func main() {  
    fmt.Println("hello, world")  
}
```

Build stage:

```
FROM golang:1.24-alpine AS build  
WORKDIR /src  
COPY go.mod main.go .  
RUN go build -o /bin/hello .
```

Walk-through

```
FROM golang:1.24-alpine AS build
```

```
WORKDIR /src
```

```
COPY go.mod main.go .
```

```
RUN go build -o /bin/hello .
```

```
FROM scratch
```

```
COPY --from=build /bin/hello /bin/hello
```

```
CMD ["/bin/hello"]
```

Exercise 14

Push your app

Push your image to the local registry at localhost:5000.

The local registry is already running:

```
podman run -d --name local-registry -p 5000:5000 registry:2
```

Container image names consist of several parts:

```
[host[:port]/]name[:tag]
```

host (and port) tell Podman where to find the registry API.

```
> podman build -t localhost:5000/<IMAGE>:<TAG> .
```

```
# OR tag an existing image
```

```
> podman tag <IMAGE>:<TAG> localhost:5000/<IMAGE>:<TAG>
```

```
> podman push --tls-verify=false localhost:5000/<IMAGE>:<TAG>
```

Exercise 15

Registry API

Query the API to see which images are present and which tags are set. Hint:

`http://localhost:5000/v2/_catalog`

Full API documentation: docs.docker.com/registry/spec/api/

```
> curl http://localhost:5000/v2/_catalog  
{"repositories":["<IMAGE>"]}
```

```
> curl http://localhost:5000/v2/<IMAGE>/tags/list  
{"name":"<IMAGE>","tags":["latest"]}
```

Option 1: Distribution repository

Ubuntu/Debian

```
apt install docker.io
```

Warning: Often outdated versions!

Option 2: Docker's official repository (recommended)

Add Docker's GPG key and repository

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg \  
| sudo gpg --dearmor -o /usr/share/keyrings/docker.gpg
```

Install latest version

```
apt update && apt install docker-ce docker-ce-cli containerd.io
```

Features:

- GUI for container management
- Built-in Kubernetes
- Easy setup on Windows/Mac

License considerations:

- Free for small businesses, personal use, education
- Requires license for companies >250 employees or >\$10M revenue
- Linux alternatives: Podman Desktop, Rancher Desktop

Linux (most distros):

Fedora/RHEL

```
dnf install podman
```

Ubuntu 20.10+

```
apt install podman
```

Arch

```
pacman -S podman
```

daemon.json Configuration

Location: /etc/docker/daemon.json

Common settings:

```
{  
  "log-driver": "json-file",  
  "log-opts": {  
    "max-size": "10m",  
    "max-file": "3"  
  },  
  "storage-driver": "overlay2",  
  "insecure-registries": ["registry.local:5000"],  
  "registry-mirrors": ["https://mirror.gcr.io"]  
}
```

daemon.json: Storage Driver

```
{  
  "storage-driver": "overlay2",  
  "storage-opts": [  
    "overlay2.override_kernel_check=true"  
  ]  
}
```

Why overlay2?

- Best performance
- Efficient disk usage
- Copy-on-write support

daemon.json: Registry Configuration

```
{  
  "insecure-registries": [  
    "registry.internal.local:5000"  
  ],  
  "registry-mirrors": [  
    "https://mirror.example.com"  
  ]  
}
```

Use cases:

- Private registries without TLS
- Corporate proxy/mirror registries
- Faster image pulls

Podman Configuration

Location: /etc/containers/storage.conf and ~/.config/containers/storage.conf

```
[storage]
```

```
driver = "overlay"
```

```
[storage.options]
```

```
mount_program = "/usr/bin/fuse-overlayfs"
```

```
[storage.options.overlay]
```

```
mountopt = "nodev,metacopy=on"
```

Rootless Configuration

Podman (rootless by default):

```
podman run nginx
```

Docker (requires setup):

```
## Install rootless extras
```

```
dockerd-rootless-setuptool.sh install
```

```
## Configure user namespaces
```

```
echo "username:100000:65536" >> /etc/subuid
```

```
echo "username:100000:65536" >> /etc/subgid
```

Docker:

- Daemon-based (dockerd runs as root)
- Client-server model
- All operations go through daemon

Podman:

- Daemonless (direct fork-exec model)
- Each command runs as separate process
- No single point of failure

Docker:

- Daemon runs as root
- Root access required for most operations
- Security concerns with daemon socket

Podman:

- Rootless containers by default
- User namespaces for isolation
- No daemon = reduced attack surface

Good news: Podman is CLI-compatible!

```
alias docker=podman
```

```
## All these work identically
```

```
docker run nginx
```

```
docker ps
```

```
docker build -t myapp .
```

```
docker compose up ## via podman compose
```

Exception: Swarm commands

Docker:

- Containers managed by Docker daemon
- systemd integration via third-party solutions

Podman:

- Native systemd integration
- Generate systemd unit files (deprecated)

```
podman generate systemd --new --name mycontainer \  
  > /etc/systemd/system/mycontainer.service  
systemctl enable --now mycontainer
```

- Using Quadlets

Systemd Integration via Quadlets

- Files placed in `/etc/containers/systemd` or `~/.config/containers/systemd`
- File types: `name.container`, `name.volume`, `name.network`, `name.kube`, `name.image`, `name.build`, `name.pod`
- See `man 5 podman-systemd.unit`

[Container]

`Image=nginx:latest`

`PublishPort=8080:80`

`Volume=/var/www:/usr/share/nginx/html:Z`

[Service]

`Restart=always`

[Install]

`WantedBy=default.target`

Docker:

- No native pod concept
- Use docker compose for multi-container apps

Podman:

- Native Kubernetes-style pods
- Multiple containers sharing network/storage

```
podman pod create --name mypod -p 8080:80
```

```
podman run --pod mypod nginx
```

```
podman run --pod mypod redis
```

Choose Docker when:

- Docker Desktop features are important
- Team already familiar with Docker
- Need Docker Swarm orchestration

Choose Podman when:

- Rootless containers are priority
- Red Hat/Fedora environment
- Kubernetes pod compatibility needed
- No daemon dependency wanted

Moving from Docker to Podman:

1. Install Podman
2. Set alias: `alias docker=podman`
3. Test existing scripts
4. Update CI/CD pipelines if needed
5. Handle docker-compose → podman-compose, if still using the legacy compose

Most scripts work without modification!

Key Takeaways

- Docker: Mature, daemon-based, wide ecosystem
- Podman: Daemonless, rootless, Kubernetes-compatible
- Both use same container standards (OCI)
- Multiple runtime options (runc, crun, gVisor)
- Configuration via daemon.json or storage.conf

Linux-in-Linux: Users and permission

- Processes in Linux are assigned to a user id.
- Control user running container
 - in Dockerfile or
 - during podman run
- UID in container and host can differ (→ user namespaces)

Best practice

- create a non-privileged user in your image
- run container as unprivileged user

Exercise — User Privileges

- run a container
 - image `node:latest`
 - bind-mount the directory `/tmp` into the container to `/tmp`
 - as command create a file in the folder `/data`
- do this exercise in 4 scenarios:
 - without further settings
 - by setting `--user 1000`
 - using host namespaces `--userns host` for user 1000
 - using host namespaces `--userns host` without specifying the user

Can you observe the difference in the folder `tmp` of your host?

Walk-through

```
> alias d='docker run -v /tmp:/tmp'
> d --rm node:latest sh -c 'touch /tmp/$(whoami)-$(hostname)'
> d --user 1000 --rm node:latest sh -c 'touch /tmp/$(whoami)-$(hostname)'
> d --userns=host --rm node:latest sh -c 'touch /tmp/$(whoami)-$(hostname)'
> d --user 1000 --userns=host --rm node:latest sh -c 'touch /tmp/$(whoami)-$(hostname)'
> ls -ltr /tmp | tail -n 4
-rw-r--r-- 1 231072 231072      0 Apr 14 13:52 root-2528fdbdfc37
-rw-r--r-- 1 232072 232072      0 Apr 14 13:52 node-b23a6edeadc8
-rw-r--r-- 1 root   root      0 Apr 14 13:52 root-a5d7aadb31d8
-rw-r--r-- 1 knoppi knoppi    0 Apr 14 13:52 node-8d032cce2e56
```

- By default 1-1-mapping of UIDs between host and container
- Remapping allows to to a $i \rightarrow (i + x)$ mapping between host and container.
- Configure this in `/etc/docker/daemon.json`.
- No privilege escalation!
- By force: `podman run --userns host`

- clair
- trivy
- gype

- scans packages and compares them to CVEs
 - Alpine
 - RHEL / CentOS / OracleLinux
 - Debian / Ubuntu
- scans language specific packages
 - Ruby
 - Java
 - JavaScript
 - Python

Exercise — quick tutorial gype

1. install gype (<https://github.com/anchore/gype#installation>)
2. scan the images nextcloud:fpm (based on Debian) and nextcloud:fpm-alpine (based on Alpine)

Note: exact vulnerability counts change with every DB update — the comparison is what matters.

Walk-through

```
> gype nextcloud:fpm
```

- ✓ Vulnerability DB [no update available]
- ✓ Pulled image
- ✓ Cataloged packages [~300 packages]
- ✓ Scanned image [several hundred vulnerabilities]

NAME	INSTALLED	FIXED-IN	VULNERABILITY	SEVERITY
------	-----------	----------	---------------	----------

...

→ many packages, many findings (Debian base + PHP stack)

```
> gype nextcloud:fpm-alpine
```

- ✓ Vulnerability DB [no update available]
- ✓ Pulled image
- ✓ Cataloged packages [~80 packages]
- ✓ Scanned image [significantly fewer vulnerabilities]

→ smaller attack surface

Exercise — Scan base images (grype)

Scan the images `debian:12`, `alpine:3.21` and `gcr.io/distroless/static-debian12` using `grype`

Walk-through

```
> gype debian:12
```

- ✓ Cataloged packages [~90 packages]
- ✓ Scanned image [several dozen vulnerabilities]

NAME	INSTALLED	FIXED-IN	VULNERABILITY	SEVERITY
------	-----------	----------	---------------	----------

...

```
> gype alpine:3.21
```

- ✓ Cataloged packages [~15 packages]
- ✓ Scanned image [0-2 vulnerabilities]

```
# → minimal base, minimal attack surface
```

```
> gype gcr.io/distroless/static-debian12
```

- ✓ Cataloged packages [3 packages]
- ✓ Scanned image [0 vulnerabilities]

No vulnerabilities found

```
# → no shell, no package manager, nothing to exploit
```

- scans packages and compares them to CVEs
- can scan configurations (IaC, Dockerfiles)

Exercise — quick tutorial trivy

1. install trivy (aquasecurity.github.io/trivy/latest/getting-started/installation)
2. scan the images `nextcloud:fpm` (based on Debian) and `nextcloud:fpm-alpine` (based on Alpine)

Note: exact counts change with every DB update — the comparison is what matters.

Walk-through

```
> trivy image nextcloud:fpm
```

```
nextcloud:fpm (debian 12)
```

```
=====
```

```
Total: several hundred (LOW: many, MEDIUM: some, HIGH: some, CRITICAL: few)
```

```
# → Debian base + PHP + many packages = large attack surface
```

```
> trivy image --severity HIGH,CRITICAL nextcloud:fpm-alpine
```

```
nextcloud:fpm-alpine (alpine 3.x)
```

```
=====
```

```
Total: significantly fewer
```

```
# → Alpine base + same app, much smaller attack surface
```

Exercise — Scan base images (trivy)

Scan the images `debian:12`, `alpine:3.21` and `gcr.io/distroless/static-debian12`

Walk-through

```
> trivy image debian:12
```

```
debian:12 (debian 12)
```

```
=====
```

```
Total: several dozen (LOW: most, MEDIUM: some, HIGH: few)
```

```
> trivy image alpine:3.21
```

```
alpine:3.21 (alpine 3.21)
```

```
=====
```

```
Total: 0-2
```

```
# → minimal base OS
```

```
> trivy image gcr.io/distroless/static-debian12
```

```
gcr.io/distroless/static-debian12 (debian 12)
```

```
=====
```

```
Total: 0
```

```
No vulnerabilities found
```

Exercise — Log4Shell

Scan the image `spaceinvaderone/log4shell-testing-vulnerable` with both `grype` and `trivy` for the `log4shell` vulnerability

Walk-through

```
> trivy image spaceinvaderone/log4shell-testing-vulnerable | grep -C 5 CVE-2021-44228
| com.fasterxml.jackson.core:jackson-databind | CVE-2020-36518 | HIGH      | 2.13.0 | 2.12.6.
| (spring-boot-application.jar)                |                |                |                | ...
|                |                |                |                | ...
+-----+-----+-----+-----+
--+-----...
| org.apache.logging.log4j:log4j-core          | CVE-2021-44228 | CRITICAL | 2.14.1 | 2.12.2, 2
| (spring-boot-application.jar)                |                |                |                | ...
|                |                |                |                | ...
|                |                |                |                | ...
+                +-----+                +                +-
-----...
```

```
> gype spaceinvaderone/log4shell-testing-vulnerable | grep -C 4 CVE-2021-44228
```

```
log4j-api          2.14.1          java-archive CVE-2021-44832      Medium
```

Exercise — scan a Dockerfile using trivy

1. scan a Dockerfile from somewhere

Walk-through

```
> cat repo/Dockerfile
FROM scratch
ADD rootfs_20.04.tar.gz /
ADD libs/* /tmp
RUN dpkg -i /tmp/*.deb
```

```
> ./trivy config repo
2021-08-24T17:20:45.193+0200      INFO      Detected config files: 1
```

Dockerfile (dockerfile)

=====

Tests: 23 (SUCCESSES: 21, FAILURES: 2, EXCEPTIONS: 0)

Failures: 2 (UNKNOWN: 0, LOW: 1, MEDIUM: 0, HIGH: 1, CRITICAL: 0)

```
+-----+-----+-----+-----+
-----
```

Best Practices Image Scanning

- Scan 4 times:
 - Image scanning during build
 - Vulnerability scanning in registry
 - Scan during admission (e.g. in k8s)
 - Regularly check running containers
- Pin versions
- Prefer slim bases

Include scan in your pipeline

Scanning in a CI Pipeline

Even without running a CI system today — the pattern:

```
git push
  → trivy config .           # scan Containerfile before building
  → podman build             # build image
  → trivy image <image>      # scan built image
  → podman push              # push only if scans pass
```

Reference pipeline (GitHub Actions style):

```
- name: Scan Containerfile
  run: trivy config --exit-code 1 .

- name: Build image
  run: podman build -t localhost:5000/myapp:latest .

- name: Scan image
  run: trivy image --severity HIGH,CRITICAL --exit-code 1 localhost:5000/myapp:latest

- name: Push image
```

Exercise — Scan the VisitCounter image

1. Build the VisitCounter: `podman build -t visitcounter:latest listings/visitcounter/app/`
2. Scan the final image: `trivy image visitcounter:latest`
3. Compare to the builder image: `trivy image golang:1.24-alpine`
4. What is the attack surface difference between the two?

Items to discuss

- mono repo / multiple repos
- deployment
- control
- GitOps

Day 3 — Declarative Deployment

Today:

- Cloud Native Patterns
- Podman & Systemd
- Quadlets: declarative container units
- Rootless Deployment
- Final Project

Goal of the Day:

The app from Day 2 gets deployed declaratively — as a systemd service, rootless, with automatic restart.

Cloud Native Patterns

Cloud native technologies empower organizations to build and run **scalable applications in modern, dynamic environments** — public, private, and hybrid clouds.

Key characteristics:

- **Loosely coupled** — services are independent, replaceable
- **Resilient** — failures are expected and handled gracefully
- **Observable** — logs, metrics, traces are first-class citizens
- **Automatable** — humans define desired state, systems converge to it

Source: Cloud Native Computing Foundation (CNCF)

12-Factor App

Methodology for building portable, resilient services — maps directly to containers:

Factor	Principle	Container connection
Config	Store config in environment	<code>--env / env_file</code>
Backing services	Treat DB, cache as attached resources	service names via DNS
Processes	Stateless, share nothing	ephemeral containers
Logs	Treat as event streams	<code>stdout</code> → <code>podman logs</code>
Disposability	Fast startup, graceful shutdown	<code>SIGTERM</code> handling

Full reference: 12factor.net

Container Design Principles

One process per container

- Easier to scale, replace, and debug
- Logs and signals go to the right place
- Compose / Quadlets handle multi-process coordination

Immutable images

- Never patch a running container — rebuild the image
- Tag by content (digest or version), not latest in production
- Rollback = restart the old image

Fail fast, recover fast

- Healthchecks tell the runtime when to restart
- Restart=always in Quadlets — the system recovers, not the operator

What This Looks Like in Practice

Our VisitCounter stack follows these principles:

- app reads all config from environment variables (Factor: Config)
- db is an attached backing service, reachable by name (Factor: Backing services)
- app is stateless — only db has a volume (Factor: Processes)
- All output goes to stdout, readable via `podman logs` (Factor: Logs)
- Quadlets restart containers automatically on failure (Factor: Disposability)

From Code to Container

The workflow — manual today, automated in CI/CD:

code change

- `podman build` (build new image)
- `trivy image` (scan for vulnerabilities)
- `podman push` (push to registry)
- `systemctl restart` (Quadlet picks up new image)

CI/CD automates steps 2–4. The concepts are the same.

Rootless Builds

Tool	How	When
podman build	Uses Buildah internally, rootless	Local dev, simple pipelines
buildah	Fine-grained layer control	Advanced scripting
kaniko	Runs in userspace, no daemon	CI without Podman available

No daemon, no root required for any of these.

Local Registry

For development and training — run a registry locally:

```
podman run -d \  
  --name local-registry \  
  -p 5000:5000 \  
  registry:2
```

Verify it's running:

```
curl http://localhost:5000/v2/_catalog  
# {"repositories":[]}
```

No TLS config needed for localhost — Podman allows insecure access by default there.

Building and Pushing an Image

Build

```
podman build -t localhost:5000/visitcounter:latest \  
  listings/visitcounter/app/
```

Check sizes – spot the difference between builder and final image

```
podman images | grep -E 'golang|visitcounter'
```

Push

```
podman push --tls-verify=false localhost:5000/visitcounter:latest
```

Verify

```
curl http://localhost:5000/v2/_catalog
```

```
curl http://localhost:5000/v2/visitcounter/tags/list
```

Start a local registry and push the VisitCounter

1. Start a local registry container on port 5000
2. Build the VisitCounter image from `listings/visitcounter/app/`
3. Tag it as `localhost:5000/visitcounter:latest`
4. Push it to the local registry
5. Verify via the registry API: does the image appear?

Walk-through

Start registry

```
podman run -d --name local-registry -p 5000:5000 registry:2
```

Build

```
cd listings/visitcounter/app
```

```
podman build -t localhost:5000/visitcounter:latest .
```

Push

```
podman push --tls-verify=false localhost:5000/visitcounter:latest
```

Verify

```
curl http://localhost:5000/v2/_catalog
```

```
# {"repositories":["visitcounter"]}
```

```
curl http://localhost:5000/v2/visitcounter/tags/list
```

```
# {"name":"visitcounter","tags":["latest"]}
```

Update Workflow with Quadlets

After a code change — this is what CI/CD would trigger automatically:

1. Rebuild

```
podman build -t localhost:5000/visitcounter:latest .
```

2. Push

```
podman push --tls-verify=false localhost:5000/visitcounter:latest
```

3. Restart the Quadlet service

```
systemctl --user restart visitcounter-app.service
```

4. Verify

```
curl localhost:8080
```

systemd and Quadlets handle restart and recovery — the image update is just a push + restart.

Credentials should never be passed as plain `-e` variables in production.

Podman Secrets:

```
echo "my_secret" | podman secret create db_pass -  
podman run --secret db_pass postgres
```

Note: For Kubernetes, use native Secret objects — never store secrets in images or compose files.

Skopeo: Images Without Running Containers

skopeo works with images directly — without pulling or running them:

Inspect metadata from registry (no pull needed)

```
skopeo inspect docker://localhost:5000/visitcounter:latest
```

Copy between registries

```
skopeo copy \  
  docker://localhost:5000/visitcounter:latest \  
  docker://other-registry.example.com/visitcounter:latest
```

Copy to a local directory (air-gapped environments)

```
skopeo copy \  
  docker://localhost:5000/visitcounter:latest \  
  dir:/tmp/visitcounter-export
```

Exercise 17

Full update cycle

Simulate what CI/CD does — manually:

1. Change the page title in `listings/visitcounter/app/main.go` (replace `VisitCounter` in the `<h1>` tag with something else)
2. Rebuild: `podman build -t localhost:5000/visitcounter:latest .`
3. Push: `podman push --tls-verify=false localhost:5000/visitcounter:latest`
4. Restart: `systemctl --user restart visitcounter-app.service`
5. Verify: `curl localhost:8080` — does the new title appear?

Walk-through

Edit

```
sed -i 's|<h1>VisitCounter</h1>|<h1>My Counter</h1>|' \
  listings/visitcounter/app/main.go
```

Rebuild

```
cd listings/visitcounter/app
podman build -t localhost:5000/visitcounter:latest .
```

Push

```
podman push --tls-verify=false localhost:5000/visitcounter:latest
```

Restart

```
systemctl --user restart visitcounter-app.service
```

Verify

```
curl localhost:8080
```

After verifying, revert the change — the VisitCounter should stay clean for the final project.

Podman: Quadlets & Declarative Deployment

Why systemd integration?

- Containers need to survive reboots, recover from crashes, start in the right order
- Docker: the daemon handles this — but the daemon is a single point of failure
- Podman: no daemon → integrate directly with **systemd**, the init system already present

Two approaches (old and new):

Approach	Status	Mechanism
podman generate systemd	Deprecated since Podman 4.4	Generates unit files from running containers
Quadlets	Current standard	Declarative .container files, systemd generates units

What are Quadlets?

Quadlets are small declarative files that describe a container (or volume, network, pod) as a systemd unit. systemd reads them and manages the container lifecycle.

File locations:

- Rootless: `~/.config/containers/systemd/`
- Root: `/etc/containers/systemd/`

File types: `.container .volume .network .pod .image .build .kube`

Reference: `man 5 podman-systemd.unit`

Quadlet: minimal example

~/.config/containers/systemd/webdemo.container:

[Container]

Image=nginx:1.27-alpine

PublishPort=8080:80

[Service]

Restart=always

[Install]

WantedBy=default.target

Activate:

```
systemctl --user daemon-reload
```

```
systemctl --user start webdemo.service
```

```
systemctl --user status webdemo.service
```

```
curl localhost:8080
```

Quadlet: volumes and networks

Volumes and networks can be declared as their own unit files — systemd ensures they exist before the container starts.

```
app-data.volume:
```

```
[Volume]
```

```
app-net.network:
```

```
[Network]
```

Reference them in .container files by name (without extension):

```
Volume=app-data.volume:/data
```

```
Network=app-net.network
```

Quadlet: VisitCounter

Three files, same directory — systemd resolves dependencies automatically.

visitcounter-db.container:

[Container]

```
Image=postgres:16-alpine
Environment=POSTGRES_DB=visitcount
Environment=POSTGRES_USER=app
Environment=POSTGRES_PASSWORD=secret
Volume=visitcounter-db.volume:/var/lib/postgresql/data
Network=visitcounter-app.network
```

[Service]

```
Restart=always
```

[Install]

```
WantedBy=default.target
```

Quadlet: VisitCounter (continued)

visitcounter-app.container:

[Container]

Image=localhost:5000/visitcounter:latest

Environment=DB_HOST=systemd-%N

Environment=DB_NAME=visitcount

Environment=DB_USER=app

Environment=DB_PASSWORD=secret

Network=visitcounter-app.network

[Unit]

After=visitcounter-db.service

[Service]

Restart=always

[Install]

WantedBy=default.target

visitcounter-proxy.container:

Declarative Deployment

Imperative (what we did so far):

```
podman run -d ...
```

```
podman run -d ...
```

```
podman run -d ...
```

State lives in your head (or shell history).

Declarative:

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

```
└─ app.container
```

```
└─ db.container
```

```
└─ app-net.network
```

```
└─ db-data.volume
```

State lives in files — readable, diffable, versionable.

Declarative Deployment: update workflow

New image version is available in the registry:

Pull new image

```
podman pull localhost:5000/visitcounter:latest
```

Restart the affected unit

```
systemctl --user restart visitcounter-app.service
```

systemd pulls the new image automatically on next start

Or automated via pipeline trigger:

In GitLab CI after push:

```
ssh deploy@trainee-host "systemctl --user restart visitcounter-app.service"
```

Declarative Deployment vs. Kubernetes

	Quadlets	Kubernetes
Scope	Single host	Cluster
State store	Files on disk	etcd
Reconciliation	systemd	control plane
Format	INI-style unit files	YAML manifests
Migration	podman kube generate	—

Same philosophy: describe desired state, let the system converge. Quadlets are a good on-ramp to understanding Kubernetes.

Deploy VisitCounter via Quadlets

1. Copy the nginx config: `mkdir -p ~/visitcounter/nginx && cp listings/visitcounter/nginx/default.conf ~/visitcounter/nginx/`
2. Create the three .container files and the .volume / .network files in `~/.config/containers/systemd/`
3. Run `systemctl --user daemon-reload`
4. Start `visitcounter-proxy.service` — systemd resolves dependencies automatically
5. Verify with `systemctl --user status visitcounter-*.service`

Walk-through

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

```
# copy unit files (see listings/visitcounter/quadlets/)
```

```
cp listings/visitcounter/quadlets/* ~/.config/containers/systemd/
```

```
systemctl --user daemon-reload
```

```
# start only the proxy – systemd starts db and app first (After=)
```

```
systemctl --user start visitcounter-proxy.service
```

```
systemctl --user status visitcounter-db.service
```

```
systemctl --user status visitcounter-app.service
```

```
systemctl --user status visitcounter-proxy.service
```

```
curl localhost:8080
```

```
# verify persistence after reboot
```

```
sudo reboot
```

```
# after login:
```

```
curl localhost:8080
```

Why wouldn't I want to use Docker?

- Design flaws (handles too much)
- Hardening docker hosts is difficult.
- Everything depends on a daemon.
 - Building images does not require a privileged daemon.
 - podman: containers do not depend on centralized daemon.

Generate Kubernetes pod YAML from a container or pod

```
> podman generate kube --help
```

Play a pod based on Kubernetes YAML

```
> podman play kube --help
```

Thoughts about runtime change

- You can use the same registries you use with docker.
- In most cases, you can switch back thanks to standardization and the common image format.

Other alternatives

Runtime:

- kata Containers
- gVisor
- containerd

Build engine / image manipulation:

- kaniko
- buildah
- skopeo
- packer
- orca-build
- source-to-image

k8s cluster → CRI → containerd / cri-o

CLI → docker / podman, buildah, skopeo / rkt

Why Container Security?

- Containers share the host kernel — a breakout is more critical than with VMs
- Default settings are a compromise between usability and security
- Principle of least privilege: a container should only be able to do what it needs

-
- Already covered: run as non-root user (USER in Dockerfile, --user)
 - This section: what else can we restrict?

Linux Capabilities

- Root in Linux is not a single privilege — it is split into ~40 **capabilities**
- Each capability can be granted or revoked independently

Capability	What it allows
NET_BIND_SERVICE	Bind to ports < 1024
SYS_PTRACE	Trace processes (strace)
CHOWN	Change file ownership
SETUID	Change process UID
NET_ADMIN	Configure network interfaces
DAC_OVERRIDE	Bypass file permission checks

- Containers run with a reduced but still generous default set

Full reference: `man 7 capabilities` — man7.org/linux/man-pages/man7/capabilities.7.html

Dropping Capabilities

Drop all capabilities, re-add only what is needed:

```
podman run --cap-drop ALL --cap-add NET_BIND_SERVICE nginx
```

Check what a container currently uses:

```
podman inspect <container> \
  --format '{{.HostConfig.CapAdd}} / {{.HostConfig.CapDrop}}'
```

Rule of thumb

Start with --cap-drop ALL, add back what breaks.

Read-only Filesystem

Prevent the container from writing to its own filesystem:

```
podman run --read-only nginx
```

Processes that need to write (temp files, sockets) get explicit tmpfs mounts:

```
podman run --read-only \  
  --tmpfs /tmp \  
  --tmpfs /var/cache/nginx \  
  --tmpfs /var/run \  
nginx
```

- Reduces attack surface: no persistent malware, no log tampering
- Breaks nothing in a well-written image

No New Privileges

Prevent processes inside the container from gaining new privileges via setuid/setgid binaries:

```
podman run --security-opt no-new-privileges nginx
```

- Blocks privilege escalation even if a setuid binary exists in the image
- Should be the default for all production containers
- Rootless Podman enables this by default

SELinux: Volume Labels

On SELinux-enabled systems (RHEL, Fedora), bind mounts need a label:

```
# :z – shared between multiple containers  
podman run -v /data/app:/app:z myimage
```

```
# :Z – private to this container only  
podman run -v /data/app:/app:Z myimage
```

Without the label the container cannot read the mounted files — even as root.

Check the current label:

```
ls -Z /data/app
```

Seccomp Profiles

- Seccomp filters which **system calls** a container may use
- Podman ships a default profile that blocks ~40 dangerous syscalls
- Custom profiles for stricter hardening

Use the default profile (active by default)

```
podman run --security-opt seccomp=default nginx
```

Disable for debugging only – never in production

```
podman run --security-opt seccomp=unconfined nginx
```

Custom profile

```
podman run --security-opt seccomp=./my-profile.json nginx
```

Rootless as a Security Feature

Podman's default: rootless containers via user namespaces

Container "root" maps to your user ID on the host

```
podman run --rm busybox id
```

```
# uid=0(root) gid=0(root)
```

On the host: just your own UID

- A breakout gives the attacker only your user's privileges — not root
- No daemon running as root
- Contrast: Docker daemon runs as root by default

Always:

- Run as non-root (USER in Dockerfile)
- `--security-opt no-new-privileges`
- Pin image versions (FROM nginx:1.27)
- Scan images (trivy, gype)

Where possible:

- `--read-only` + tmpfs for writable paths
- `--cap-drop ALL` + re-add as needed
- Use rootless Podman
- SELinux labels on volumes (:Z)

Exercise 19

Harden a container

Start with this insecure baseline:

```
podman run -d --name insecure \  
  -p 8080:80 \  
  nginx:1.27
```

Now harden it step by step:

1. Add `--read-only` — which paths need `--tmpfs`?
2. Add `--security-opt no-new-privileges`
3. Add `--cap-drop ALL` — does nginx still start?
4. Add back the one capability nginx needs

Verify nginx still serves pages: `curl localhost:8080`

Walk-through

```
podman run -d --name secure \  
-p 8080:80 \  
--read-only \  
--tmpfs /var/cache/nginx \  
--tmpfs /var/run \  
--security-opt no-new-privileges \  
--cap-drop ALL \  
--cap-add CHOWN \  
--cap-add SETUID \  
--cap-add SETGID \  
--cap-add NET_BIND_SERVICE \  
nginx:1.27
```

```
curl localhost:8080
```

```
# → 200 OK
```

```
podman inspect secure \  
[
```

Tooling: Inspect, Explore, Manage

Excursion: podman-tui

TUI for Podman — inspect running containers, pods, images, volumes and networks live in the terminal.

```
# Fedora/RHEL
```

```
dnf install podman-tui
```

```
# Other distros – binary release
```

```
curl -LO https://github.com/containers/podman-tui/releases/latest/download/podman-tui_linux_amd64.tar.gz
```

```
tar xf podman-tui_linux_amd64.tar.gz && ./podman-tui
```

What you get:

- Live CPU / memory / network stats per container
- Log streaming with scroll
- Exec shell directly from the TUI
- Manage images, volumes, networks — no separate commands needed
- Speaks directly to Podman — no daemon, no Docker socket

Install and launch podman-tui

Binary release (all distros)

```
curl -LO https://github.com/containers/podman-tui/releases/latest/download/podman-tui_linux_amd64.tar.gz  
tar xf podman-tui_linux_amd64.tar.gz && sudo mv podman-tui /usr/local/bin/
```

podman-tui

Navigate with arrow keys — explore containers, images, volumes and networks live.

Excursion: podman-tui — Demo

With the VisitCounter stack running:

```
podman-tui
```

Navigate with arrow keys — explore:

- Container list → select visitcounter-app → logs tab
- Stats tab — watch memory while curling the app
- Exec tab — open a shell inside the running container
- Images → spot the size difference between builder and final image

Excursion: dive

Explore image layers interactively — see exactly what each Containerfile instruction adds or changes.

Install

```
curl -LO https://github.com/wagoodman/dive/releases/latest/download/dive_linux_amd64.tar.gz  
tar xf dive_linux_amd64.tar.gz && sudo mv dive /usr/local/bin/
```

Inspect images

```
dive golang:1.24-alpine  
dive localhost:5000/visitcounter:latest
```

What to look for:

- **Layer efficiency score** — how much wasted space across layers
- Which RUN instruction adds the most weight
- Files added / modified / removed per layer (left panel)
- Why FROM scratch results in a single tiny layer

Excursion: dive — Demo

```
# Builder image – lots of layers, lots of weight  
dive golang:1.24-alpine
```

```
# Final image – one layer, one binary  
dive localhost:5000/visitcounter:latest
```

The difference makes multi-stage builds tangible: the builder is ~300 MB, the final image ~10 MB — dive shows you exactly why.

Podman Desktop

GUI for container management — Linux, macOS, Windows. Free and open source.

Features:

- Start / stop / inspect containers
- Build images from Containerfile
- Manage pods, volumes, networks
- Compose support
- Kubernetes / Kind integration
- Extensions ecosystem

Why it matters:

- Rootless by default — same as Podman CLI
- Free for everyone (no license tiers)
- Alternative to Docker Desktop on macOS/Windows
- Good onramp for developers who prefer a GUI

`podman-desktop.io`

Conclusions and Outlook

What You Can Do Now

- Run and inspect containers with Podman
- Build multi-stage images — small, secure, reproducible
- Manage storage, networking, and permissions
- Orchestrate multi-container apps with Compose
- Deploy declaratively with Quadlets + systemd
- Scan images for vulnerabilities with trivy / gype

Where to Go From Here

Deepen what you learned:

- `man podman-systemd.unit` — full Quadlet reference
- Podman Desktop — GUI for rootless containers
- Buildah — advanced image building without Dockerfile
- Skopeo — image inspection and copying

Natural next step:

- Kubernetes — same ideas, cluster scale
- `podman generate kube` — export what you built today
- OpenShift — Kubernetes + enterprise features
- Helm — package manager for K8s workloads

Key Principles to Take Home

- **Immutable:** Rebuild, don't reconfigure
- **Rootless:** Least privilege by default
- **Declarative:** State in files, not in your head
- **OCI:** Standards = tool choice, no vendor lock-in

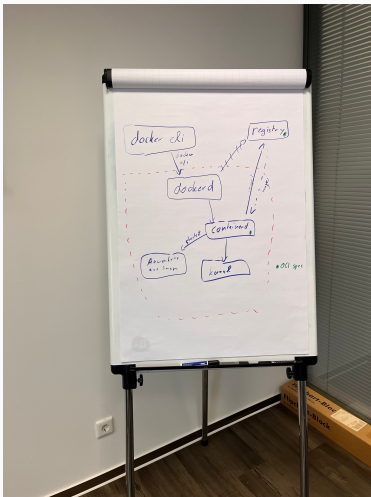
Summary

- Miniaturize your services!
- Care for security!
- Automate everything!

Thank you for your attention!

- Did you like this training?
- What did you learn?
- Leave your feedback here: <https://atix.de/trainings/feedback-training/>

Flipchart



Flipchart 2

