Containers: Appendix

Deep dives and backup slides for lectures 1–3

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

ESIR 2 SI — Architectures Logicielles
Université de Rennes

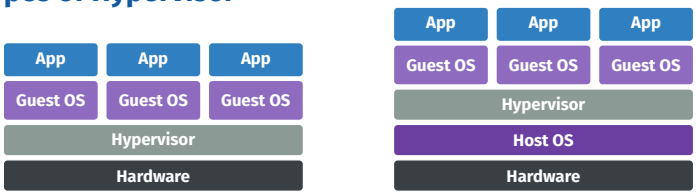
Appendix to Lecture 1: DevOps and Why Containers

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

Deep dives and backup slides, in this order:

- **Hypervisors**

Types of Hypervisor



Type 1: bare metal
Xen, VMware ESXi, Hyper-V, KVM

Type 2: hosted
VirtualBox, VMware Workstation, Parallels, UTM

KVM: Linux itself becomes the hypervisor (usually counted as type 1; QEMU provides the emulated devices).
Either way, every VM boots its own guest OS kernel.

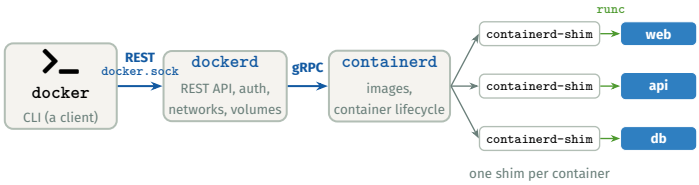
Appendix to Lecture 2: Containers in Practice

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

Deep dives and backup slides, in this order:

- **Container Engines**
- **Running Containers**
- **Registries and Images**
- **Rootless Storage in Detail**
- **Networking in Detail**
- **Building Images**
- **Supply Chain**
- **Compose**

Docker Engine Internal Architecture



- `runc`: sets up namespaces + cgroups, starts the process, then **exits**
- the shim keeps the container alive if `dockerd` / `containerd` restart
- `dockerd` runs as root by default (rootless mode exists, opt-in): group `docker` ≈ root
- Podman: `podman` → `conmon` → `crun`, no daemon

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

Container Engines: Different Levels

Orchestrator:	Kubernetes (many hosts): talks CRI (Container Runtime Interface) to each node's engine
Engines:	`docker` → `dockerd` → `containerd`; `podman` → `conmon` (no daemon, no containerd); CRI: `containerd`, `CRI-O`
Low-level OCI runtimes:	`runc`, `crun`, `youki`; gVisor (a user-space kernel), Kata (one micro-VM each)
Linux kernel:	namespaces, cgroups, capabilities, seccomp, OverlayFS

- Kubernetes dropped `dockershim` (v1.24, 2022): Docker-built images still run (OCI)
- Lab Module 1 went all the way down: `unshare`, `mount -t overlay`, `/sys/fs/cgroup`

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

One Concern per Container

Each container should do one thing and do it well

- one main process (PID 1), which may have children (nginx workers, postgres backends)
- scale APIs and front-ends differently than databases
- version and update each part in isolation
- swap one part: local dev database container, managed production database
- several services in one container: who restarts them? who collects their logs? → one container each, wired by Compose

Running Several Commands

```
$ podman run --rm docker.io/library/alpine:3.20 \
      sh -c "apk add --no-cache figlet && figlet ESIR"
$ podman run --rm docker.io/library/postgres:16-alpine \
      sh -c 'echo $PGDATA'          # expanded inside the container
/var/lib/postgresql/data
$ podman cp CONTAINER:/etc/os-release .  # out or in, even stopped
```

- `run`/`exec` start one executable; `&&`, pipes, `>`, `$VAR` are shell syntax: use `sh -c "..."`
- quote it, or your host shell expands it first (single quotes: `$PGDATA` reaches the container as is)
- packages to keep: a `RUN` line in a Containerfile

Placing Resource Limits

```
$ podman run -d --name capped \
      --memory 256m --memory-swap 256m --cpus 1.5 --pids-limit 200 \
      docker.io/library/alpine:3.20 sleep infinity
$ podman stats --no-stream capped        # live usage vs limits
NAME     CPU %   MEM USAGE / LIMIT   PIDS   ...  (trimmed)
capped   0.02%   1.2MB / 268.4MB     1
```

- default: no limit; one container can eat all host RAM and CPU
- `--memory-swap` = `--memory`: no swap; `--pids-limit` stops fork bombs
- over `--memory`: killed, `Exited (137)`, `podman inspect`: `OOMKilled: true`

These flags write cgroup files (`memory.max`, `pids.max`…): lecture 3.

Registries

A registry stores and serves images (OCI Distribution API, HTTPS): an app store for images

- push / pull by name or digest; only missing layers are transferred
- the name of an image says where it lives and is pushed

Public registries

- `docker.io`: Docker Hub (`docker.io/library/*` = "official images")
- `quay.io` (Red Hat), `ghcr.io` (GitHub), `registry.fedoraproject.org`

Private registries

- self-hosted (Harbor, `distribution`), GitLab/GitHub projects, cloud providers

`podman login / push / pull`: lecture, *Build, Run, Push, Pull.*

`https://github.com/opencontainers/distribution-spec`

Auditing an Image (1): Layers and History

```
$ podman pull docker.io/library/postgres:16-alpine
$ podman image tree docker.io/library/postgres:16-alpine
+-- ID: 63ca1fbb43ae Size: 8.6MB                (trimmed)
`-- ID: 9e8d7c6b5a4f Size: 5.1kB Top Layer of: [...]
$ podman history --no-trunc docker.io/library/postgres:16-alpine
SIZE     CREATED BY                    (newest first, trimmed)
0B       CMD ["postgres"]
0B       VOLUME [/var/lib/postgresql/data]
250MB    RUN ... wget postgresql.tar.bz2 ... make install ...
3.6kB    RUN ... addgroup -g 70 -S postgres; adduser -u 70 ...
8.6MB    ADD alpine-minirootfs-3.24.3-x86_64.tar.gz /
```

- layer = read-only directory in `~/.local/share/containers/storage/overlay/`; `0B` lines only change the config
- user `postgres` = UID 70 (see rootless storage)

Lab: Module 2, Ex. 2.1

Auditing an Image (2): Read Its Config

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

```
{ "ExposedPorts": {"5432/tcp": {}},  "StopSignal": "SIGINT",
  "Entrypoint": ["docker-entrypoint.sh"],  "Cmd": ["postgres"],
  "Env": ["PATH=...", "PG_MAJOR=16", "PGDATA=/var/lib/..."],
  "Volumes": {"/var/lib/postgresql/data": {}} }
```

- `--format`: a Go template; `jq .` pretty-prints (or `python3 -m json.tool`)
- no `User`: starts as root; the entrypoint switches to `postgres` (UID 70)
- required variables (`POSTGRES_PASSWORD`) are not in the config: read the docs or logs
- `go-httpbin`: `"User": "0"`, port 8080, only a `Cmd` (lecture: the fields table)

Lab: Module 2, Ex. 2.1

SELinux and Bind Mounts: `:z` and `:Z`

```
$ ls -dZ ~/site
unconfined_u:object_r:user_home_t:s0 /home/me/site
$ podman run -d -v ~/site:/usr/share/nginx/html:Z \
      docker.io/library/nginx:1.27
$ ls -dZ ~/site
system_u:object_r:container_file_t:s0:c123,c456 /home/me/site
```

- Fedora/RHEL: SELinux (MAC) lets `container_t` processes touch only `container_file_t` files
- `:z`: label shared by all containers; `:Z`: private label (this container's categories)
- never `:Z` a system or home directory itself
- without SELinux (Ubuntu: AppArmor) these flags are no-ops

Lab: Module 3, Ex. 3.1

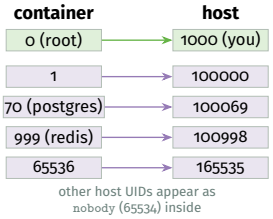
Rootless Storage: Who Owns the Files?

Your user owns a range of UIDs

`/etc/subuid`: `me:100000:65536` (`+ subgid`)

```
$ podman unshare cat /proc/self/uid_map
         0       1000      1
         1     100000  65536
```

- container `root` (0) → you (1000)
- container UID $n \geq 1$ → host UID $100000 + n - 1$



Example: on the host, `ls -ln` shows `postgres` files (UID 70) as 100069

Kernel mechanism (user namespaces): lecture 3.

Lab: Module 3, Conceptual foundation

The Rootless Bind-Mount Trap

```
$ mkdir -p ~/lab-data/redis-bind
$ ls -ldn ~/lab-data/redis-bind
drwxr-xr-x. 2 1000 1000 40 Nov  3 10:00 /home/me/lab-data/redis-bind
$ podman run -d --name redis-broken --user 999:1000 \
      -v ~/lab-data/redis-bind:/data:Z \
      docker.io/library/redis:7-alpine --save 60 1
$ podman exec redis-broken redis-cli bgsave
$ podman logs redis-broken
# Failed opening the temp RDB file temp-11.rdb (in server root
  dir /data) for saving: Permission denied
```

`:Z` settled SELinux; this time classic UNIX permissions say no.

Lab: Module 3, Ex. 3.1

Why Does It Fail?

```
$ podman exec redis-broken ls -ldn /data
drwxr-xr-x    2 0        0              40 Nov  3 10:00 /data
```

	Inside the container	On the host (kernel's view)
owner of `/data`	0 (root)	1000 (you)
`redis-server` runs as	UID 999	UID 100998
applicable bits	"others": `r-x`	"others": `r-x`

- kernel DAC check: UID 100998 writing into a 0755 dir owned by 1000 ⇒ denied
- `chmod 777` "works" but opens the directory to every user
- a root entrypoint can `chown` first (official redis, postgres); a non-root image cannot

Lab: Module 3, Ex. 3.1

Fix 1: The `:U` Volume Option

```
$ podman rm -f redis-broken
$ podman run -d --name redis-fixed --user 999:1000 \
      -v ~/lab-data/redis-bind:/data:Z,U \
      docker.io/library/redis:7-alpine --save 60 1
$ podman exec redis-fixed redis-cli save
$ ls -ln ~/lab-data/redis-bind/
-rw-------. 1 100998 100999 104 Nov  3 10:05 dump.rdb
```

- `:U`: Podman recursively `chown`s the source to the container user's mapped UID/GID
- host owner: 100998 (= container UID 999)
- options combine: `:Z,U` (`:ro,Z,...`)

Lab: Module 3, Ex. 3.1

Fix 2: `podman unshare` — Fix 3: Named Volumes

`podman unshare` runs a command *inside* your user namespace

```
$ podman unshare chown 999:1000 ~/lab-data/redis-bind
$ ls -ldn ~/lab-data/redis-bind     # host: 100998 100999
$ podman unshare rm -r ~/lab-data/redis-bind   # clean up
```

Named volume: `-v redis-data:/data`

- filled from the image with the right owner: nothing to fix
- the best choice when you don't need to edit the files from the host

After fix 1 or 2, your host user cannot edit those files directly any more.

Lab: Module 3, Ex. 3.1

Podman's Network Stack, Rootless

- netavark (replaced CNI in Podman 4): bridges, `veth` pairs, firewall, port forwarding
- aardvark-dns: one DNS server per user-defined network: container names resolve

Docker: built-in `libnetwork`, DNS at `127.0.0.11`.

Rootless: no host bridge

(creating one needs root)

- pasta (default since Podman 5, was `slirp4netns`): packets become sockets of your user
- user-defined networks: bridge in a private *rootless network namespace*, linked out by pasta
- container IPs unreachable from the host: use `-p`

```
>_ podman info --format '{{.Host.NetworkBackend}}'   → netavark
```

Lab: Module 3, Ex. 3.2

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

The Default Bridge

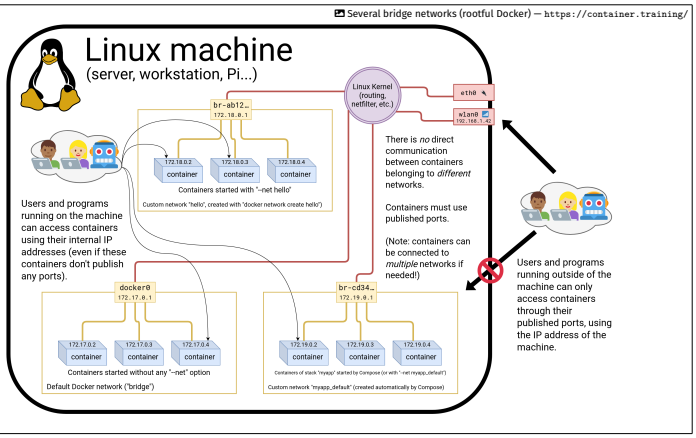
A virtual switch on the host (rootful model)

- a Linux bridge: Docker `docker0` (`172.17.0.0/16`), Podman `podman0` (`10.88.0.0/16`)
- each container: its own network namespace with an `eth0` = one end of a `veth` pair, the other end plugged into the bridge
- the bridge's IP is the containers' gateway
- outgoing traffic: NAT (masquerade); `-p`: DNAT rules

On the default network (`podman`, Docker's `bridge`), containers cannot resolve each other by name.

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

Several bridge networks (rootful Docker) — https://container.training/



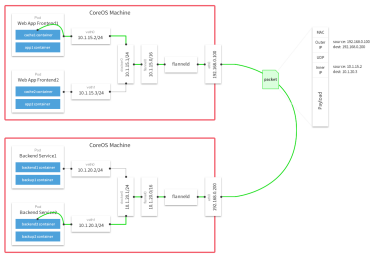
Network Modes

`>_ podman run --network MODE ...` (same for `docker run`)

`bridge`	default when rootful: private subnet, NAT, `-p` to publish
`NAME`	a user-defined bridge network: DNS by container name
`none`	only `lo`: no network at all
`host`	the host's network stack: no isolation, native speed
`container:ID`	share another container's network namespace (talk over `localhost`, like a Podman *pod*)
`pasta`	default when rootless: user-space networking, ports are ordinary sockets of your user

Multi-Host Networking

- a bridge spans one host
- across hosts: an overlay (VXLAN: packets in UDP)
- Kubernetes: CNI plugins Flannel, Calico, Cilium
- Weave Net: discontinued



Flannel overlay between two hosts

A First Containerfile

Recreate an application **anywhere** a container engine runs:

```
# Pinned, fully-qualified base image
FROM docker.io/library/debian:13-slim
# Update + install + clean up in ONE layer
RUN apt-get update \
 && apt-get install -y --no-install-recommends memcached \
 && rm -rf /var/lib/apt/lists/*
# Do not run as root; document the port
USER memcache
EXPOSE 11211
CMD ["memcached"]
```

- `-y`: builds are non-interactive (nobody can answer "Continue? [Y/n]")
- `#` is a comment only at the start of a line

Containerfile for a Node.js Application

```
FROM docker.io/library/node:22-alpine

WORKDIR /my/app
COPY package.json package-lock.json ./
RUN npm ci --omit=dev
COPY . .

USER node
ENV NODE_ENV=production
EXPOSE 3000
CMD ["node", "web.js"]
```

← Base image

← Run while building the image

← Runtime settings

← Run to start the app

Why copy `package.json` *before* the source code? → lecture, "The Build Cache".

Main Instructions

Building the filesystem

`FROM`	base image, **first**; pin it!
`RUN`	command at build time ⇒ layer
`COPY`	files from context/stage ⇒ layer
`ADD`	+ tarballs, URLs (prefer `COPY`)
`WORKDIR`	set (and create) current dir.
`ARG`	build-time variable; gone after
`ENV`	env. variable stored in the image

Metadata for the runtime (config only)

`USER`	user for next steps (numeric)
`EXPOSE`	app port: metadata only
`VOLUME`	anonymous volume at run time
`ENTRYPOINT`	executable started as **PID 1**
`CMD`	default command or *arguments*
`LABEL`	key/value metadata
`HEALTHCHECK`	app healthy?*

⚠ No secrets in `ARG`/`ENV`: `podman history` and `image inspect` show them.

Prefer the **exec form** `CMD ["node", "web.js"]` (next slides).

* Not in OCI images: `podman build` drops it unless `--format docker`. Or at run time: Compose `healthcheck:`, Quadlet `HealthCmd=`.

Shell Form vs Exec Form

Shell form (plain string)

```
RUN apk add figlet
CMD figlet hello $USER
```

- run as `/bin/sh -c "..."`
- expands `$VAR`, `&&`, pipes
- needs a shell in the image
- `CMD`: the shell is PID 1, not your app

Exec form (JSON array)

```
RUN ["apk", "add", "figlet"]
CMD ["figlet", "hello"]
```

- `execve()` directly, no parsing
- no extra process
- no shell needed (distroless)
- your app is **PID 1**, gets `SIGTERM`

Rule of thumb: shell form for `RUN`, **exec form** for `CMD` and `ENTRYPOINT`.

ENTRYPOINT and CMD

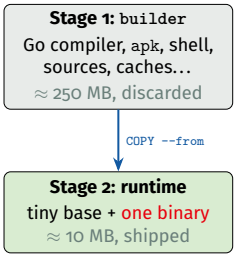
```
FROM docker.io/library/alpine:3.20
RUN apk add --no-cache figlet
ENTRYPOINT ["figlet", "-f", "script"]
CMD ["hello world"]
```

`podman run ...`	**Runs**
`myfiglet`	`figlet -f script hello world`
`myfiglet hola`	`figlet -f script hola` (replaces `CMD`)
`-it --entrypoint sh myfiglet`	`sh` (replaces both)

- `ENTRYPOINT`: the base command; `CMD`: its default arguments
- `ENTRYPOINT` suits "containerized binaries" (`podman run tool --help`)

Multi-Stage Builds

- Each `FROM` starts a new **stage**
- Name it: `FROM ... AS builder`
- Later stages pick files from it: `COPY --from=builder`
- Only the last stage becomes the tagged image
- Compilers, sources, caches stay behind



Lab: Module 4, Ex. 4.1

Lab: A Tiny Go Web Service

`main.go` (abridged)

```
package main

import ("fmt"; "net/http"; "os")

func main() {
  port := os.Getenv("PORT")
  if port == "" { port = "3000" }
  http.HandleFunc("/", hello)
  fmt.Printf("Listening on :%s\n", port)
  http.ListenAndServe(":"+port, nil)
}
// hello() prints the container hostname
```

- standard library only
- port from the `PORT` **environment variable** (12-factor config)
- prints the container **hostname** (UTS namespace)
- Go builds a single static binary: ideal for tiny images

Lab: Module 4, Ex. 4.1

Build Secrets: Not in `ARG` or `ENV`

```
# BAD: build-argument values are recorded in the image history
ARG API_TOKEN
RUN curl -H "Authorization: Bearer $API_TOKEN" -o x.tgz $URL
# GOOD: a secret FILE mounted for this one RUN, written to no layer
RUN --mount=type=secret,id=token \
    curl -H "Authorization: Bearer $(cat /run/secrets/token)" \
         -o x.tgz $URL
```

```
$ podman build --secret id=token,src=token.txt -t app:1.0 .
$ podman history --no-trunc app:1.0      # BAD version: token shown
```

- same for `ENV`, and for a `COPY`'d file deleted later (still in its layer)
- same syntax with `docker build` (BuildKit); SSH keys: `--mount=type=ssh`

Supply Chain: Scan, List, Sign

```
$ trivy image docker.io/library/nginx:1.27      # known CVEs
$ grype docker.io/library/nginx:1.27            # same idea
$ syft docker.io/library/nginx:1.27 -o spdx-json=sbom.json
$ podman build --sbom=syft --sbom-output=sbom.json -t app:1.0 .
$ cosign sign --key cosign.key quay.io/esir/web@sha256:...
$ cosign verify --key cosign.pub quay.io/esir/web:1.0
```

- **Scan** packages against CVE databases; CI fails on critical; rebuild often (base images get fixes)
- **SBOM** (software bill of materials): packages and versions → "are we affected?" at the next CVE
- **Sign**: who built this digest? Podman checks on pull when `/etc/containers/policy.json` requires `sigstoreSigned`

🔗 `https://docs.sigstore.dev/`

Compose File Structure

`compose.yaml` (preferred name; `docker-compose.yml` still accepted)

- `services` (mandatory): one entry per container
  - each one has an `image`, a `build` directory, or both
- `networks` (optional): networks to create and attach (default: one private network per project)
- `volumes` (optional): **named volumes** shared or kept across runs
- ~~`version: "3"`~~: obsolete, ignored (Compose v2 warns about it)

The **project name** (default: the directory name) prefixes everything: `selfhosted-stack_pgdata`, `selfhosted-stack_frontend-net`…

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

Service Keys Are `podman run` Options

Compose key	`podman run` option	**Meaning**
`image` / `build`	`IMAGE` / `podman build`	what to run
`container_name`	`--name`	fixed container name
`ports`	`-p`	publish a port
`volumes`	`-v`	volume or bind mount
`environment`	`-e`	environment variables
`networks`	`--network`	attach to networks
`command`	trailing args	overrides `CMD`
`restart`	`--restart`	restart policy
`mem_limit`	`--memory`	cgroup `memory.max`
`healthcheck`	`--health-cmd ...`	health test
`depends_on`	—	start order (Compose only)

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

Service Discovery: Who Is `postgres`?

```
postgres://USER:PASSWORD@postgres/DBNAME?sslmode=disable
```

- `postgres` is the **service name**: the network's DNS (aardvark-dns) resolves it to the container's IP
  - only on networks both containers share (`backend-net`)
  - default port 5432, never published on the host
- IPs change at every `up`; names do not
- `sslmode=disable`: OK, traffic stays on a private network inside one host
- Quote `"1"`: YAML would otherwise read `1` as a number

🧪 **Lab: Module 4, Ex. 4.2**

Life After Compose

(Or: when do we need something else?)

- Compose is not a cluster orchestrator
  - one host, no scheduling across nodes: if the host dies, the stack dies
- Great for **development**, CI, **small single-host deployments**
  - with rootless Podman, the stack lives in your login session: what happens at logout? at reboot?

→ Lecture 3: systemd, lingering, Quadlets. Beyond one host: appendix, lecture 3 part.

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

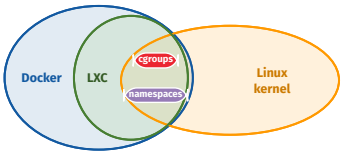
Appendix to Lecture 3: Under the Hood and Production

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

Deep dives and backup slides, in this order:

History	Namespaces	Control Groups	Security in Detail
OverlayFS	Quadlets, Pods, Kubernetes		

From LXC to runc



The original Docker stack (2013): Docker drove LXC, which used kernel features

- **2008, LXC**: cgroups + namespaces, driven by *external tooling*
- **2013, Docker** drives LXC, adds images (layers), Dockerfile, registry
- **2014**: Docker replaces LXC with its own `libcontainer`
- **2015**: `libcontainer` → `runc`, given to the OCI
- **Today**: `runc`, `crun`; LXC / Incus remain for "system containers"

Manipulating Namespaces

Syscall	Effect	Used by
`clone(2)`	a *new child* starts in new namespaces	runtimes: the container's PID 1
`unshare(2)`	moves the *caller* into new namespaces	`unshare(1)`
`setns(2)`	*joins* existing namespaces	`nsenter(1)`, `podman exec`

- Pseudo-files `/proc/<pid>/ns/*`: same inode = same namespace
- Destroyed when the last process exits (unless bind-mounted)

(full comparison: next slide)

```
$ ls -l /proc/self/ns
cgroup -> cgroup:[4026531835]
ipc    -> ipc:[4026531839]
mnt    -> mnt:[4026531832]
net    -> net:[4026531833]
pid    -> pid:[4026531836]
...
```

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

Entering Namespaces: `clone` vs `unshare` vs `setns`

	`clone(2)`	`unshare(2)`	`setns(2)`
Isolates	The *new child*	The *caller* itself	The *caller* itself
New process?	Yes	no	no
Namespaces	Creates new ones	Creates new ones	Joins *existing* ones (`/proc/<pid>/ns/*`)
Typical user	Container runtimes (the container's PID 1)	`unshare(1)`, runc's `nsexec`	`nsenter(1)`, `podman exec`
In practice	Spawn a program that *starts* isolated; the parent can write its `uid_map` from outside	Isolate *yourself*, or step by step (user namespace first, then the others)	Enter a running container

⚠ PID (and time) namespaces only apply to *future children* ⇒ `unshare --pid` needs `--fork`.

The Eight Namespaces

Namespace	Flag	Isolates
mnt	CLONE_NEWNS	mount table: root filesystem, /tmp…
uts	CLONE_NEWUTS	hostname, NIS domain name
ipc	CLONE_NEWIPC	System V IPC, POSIX queues
pid	CLONE_NEWPID	process IDs
net	CLONE_NEWNET	interfaces, routes, firewall, ports
user	CLONE_NEWUSER	UIDs/GIDs and capabilities
cgroup	CLONE_NEWCGROUP	view of the cgroup tree
time	CLONE_NEWTIME	monotonic and boot-time clock offsets

- Each process belongs to exactly one namespace of each type
- **Always active**: even without containers, one namespace of each type holds all processes

Mnt Namespace and pivot_root

- Processes get their own mount table and their own root filesystem
- Containers: mount the OverlayFS root (OverlayFS, lecture 3), then pivot_root(2) into it
  (safer than chroot: the old root is detached, not just hidden)
- "Private" mounts: isolate /tmp, mask parts of /proc and /sys, show secrets only to some processes

```
$ unshare --user --map-root-user --mount bash
# mount -t tmpfs none /tmp
# touch /tmp/only-here; ls /tmp
only-here
# exit                the mount vanishes with the namespace
```


UTS Namespace

- gethostname / sethostname
- A custom hostname per container (podman run --hostname web)
- That's (mostly) it!
- UTS = *UNIX Time-Sharing*, from struct utsname (uname(2))

```
$ unshare --user --map-root-user \
      --uts bash
# hostname
mylaptop
# hostname tupperware
# hostname
tupperware
# exit
$ hostname
mylaptop
```

No sudo needed: creating a *user* namespace first gives us root powers over the new namespaces.

Net Namespace

- Its own private network stack: interfaces (its own lo), routes, firewall rules, sockets
- So two containers can both listen on port 80
- The engine plugs it in: veth + bridge (rootful), pasta (rootless), or --network=none / host / container:web (next slide)

```
$ unshare --user --map-root-user --net bash
# ip link
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN ...
# ping -c1 1.1.1.1
ping: connect: Network is unreachable
```


Net Namespaces in Practice

Typical use

- One network namespace per container
- **Rootful**: a veth pair (virtual cable): eth0 in the container, the other end on a host bridge (podman0, docker0) + NAT
- **Rootless**: no host interfaces → pasta turns the traffic into sockets of your user
- Move an interface: ip link set dev eth0 netns <PID>

Engine options

- --network=none: only lo
- --network=host: no network namespace, no isolation
- --network=container:web: join another container's (pods do this)

Details: *Networking in detail* (this appendix), lab Module 3.

PID Namespace

- Processes only "see" processes of their PID namespace
- Each has its own numbering, starting at 1
- Nested: one PID per level (PID 1 in the container, 24817 on the host)
- PID 1 exits ⇒ the whole namespace is killed
  (PID 1 lifecycle, zombies, --init: lecture 2)
- ps reads /proc: a new PID namespace needs a new /proc mount
- unshare --pid only affects future *children* ⇒ --fork (next slides)

PID Namespace in Action

```
$ unshare --user --map-root-user --pid --fork bash
# echo $$
1
# ps aux | wc -l            why all the host processes?!
312
# kill 4242
bash: kill: (4242) - No such process
```

- ps has no special syscall: it reads /proc, still the host's
- Host processes visible, but out of reach
- Fix: a new /proc in a new mount namespace: --mount-proc

```
$ unshare --user --map-root-user --pid --fork --mount-proc bash
# ps aux    → only bash (PID 1) and ps (PID 2)
```


Why Do We Need --fork?

- unshare(1) = unshare(2), then exec the program
- The caller moves into the new namespaces…
  except PID: its PID cannot change from *X* to 1
- Only its *children* enter the new PID namespace; the first becomes PID 1
- Without --fork: bash's first command becomes PID 1, exits ⇒ namespace dead
  (next fork: "Cannot allocate memory")
- With --fork: unshare forks; the child (PID 1) execs bash

Runtimes use clone(CLONE_NEWPID) directly. See man 2 unshare, man 7 pid_namespaces.

IPC, Time and cgroup Namespaces

IPC

- System V semaphores, message queues and shared memory, without conflicts between containers (old PostgreSQL versions cared)

Time (Linux 5.6)

- Offsets for CLOCK_MONOTONIC and CLOCK_BOOTTIME: useful to checkpoint and restore a container on another machine (CRIU)
- The wall clock (CLOCK_REALTIME) is not namespaced

Cgroup

- Virtualizes /proc/self/cgroup and the root of /sys/fs/cgroup
- In a container, cat /proc/self/cgroup shows 0::/: that is why Ex. 1.3 reads /sys/fs/cgroup/memory.max directly

When unshare --user Fails (Ubuntu ≥ 23.10)

- Ubuntu restricts unprivileged user namespaces with AppArmor (23.10; on by default since 24.04): a hardening against kernel exploits
- The lab's unshare --user --map-root-user may then fail; **Podman still works** (its AppArmor profile allows user namespaces)

```
$ unshare --user --map-root-user bash
unshare: write failed /proc/self/uid_map: Operation not permitted
$ sysctl kernel.apparmor_restrict_unprivileged_userns
kernel.apparmor_restrict_unprivileged_userns = 1
```

What to do (check with the VM admin first): explore from Podman's own user namespace,

```
podman unshare unshare --pid --mount --uts --fork bash
```

or (admin only, lowers the protection):

```
sudo sysctl kernel.apparmor_restrict_unprivileged_userns=0
```

History: cgroups v1 vs v2

cgroups v1 (2008; what Docker was built on)
- One tree per controller: `/sys/fs/cgroup/memory/`, `.../cpu/...`
- A process could sit at different places in each tree
- Unsafe to delegate to unprivileged users

cgroups v2 (Linux 4.5, 2016): a rewrite
- Single unified hierarchy; processes only in leaves
- Safe delegation of a subtree to a user → rootless limits
- New: `memory.high`, pressure stall info (PSI), `cgroup.kill`
- Default: Fedora 31+, Debian 11+, Ubuntu 21.10+, RHEL 9

Old tutorials use v1 names: `memory.limit_in_bytes`, `cpu.cfs_quota_us`...

Example: The cgroup Tree of a Workstation

```
$ tree -d /sys/fs/cgroup                          (trimmed)
/sys/fs/cgroup/
|-- init.scope                                    systemd, PID 1
|-- system.slice
|   |-- sshd.service
|   `-- docker-de3ee38bc8d9...scope               a rootful container
`-- user.slice
    `-- user-1000.slice
        |-- session-1.scope                       your login shell
        `-- user@1000.service                     your systemd --user
            `-- user.slice
                `-- libpod-3077775c9852...scope   rootless
```

Rootless containers live inside your user's slice: they share your user's limits, and die with your session unless "lingering" is enabled (lecture 3, lab Module 5).

Memory cgroup: Accounting and Limits

Accounting
- `memory.current`, `memory.peak`: bytes used
- `memory.stat`: breakdown (anon = heap/stack, file = page cache, kernel, sockets...)
- Each page is "charged" to *one* group (shared pages: first user pays)

Limits and reservations (v2)

`memory.min`	hard reservation: never reclaimed
`memory.low`	soft reservation: reclaimed only as a last resort
`memory.high`	throttle + aggressive reclaim, *no* OOM kill
`memory.max`	hard limit, then the OOM killer
`memory.swap.max`	swap the group may use

v1: only a soft and a hard limit (soft ones were hard to use).

Hard Limits and the OOM Killer

- A cgroup never exceeds `memory.max`; when it tries to:
  1. the kernel reclaims from the group (page cache, swap if allowed)
  2. else the OOM killer, *scoped to this cgroup*, `SIGKILL`s its biggest process
- `memory.oom.group = 1`: kill the whole group
- `memory.events`: counters `high`, `max`, `oom`, `oom_kill`

Avoiding the OOM killer (databases, stateful services)
- `memory.high` below `memory.max`: slow down instead of dying
- monitor pressure (`memory.pressure`, PSI); v1's "OOM notifier" is gone

Setting a Memory Limit by Hand

It is "just files". As root, on the host:

```
$ sudo mkdir /sys/fs/cgroup/demo
$ echo 100M | sudo tee /sys/fs/cgroup/demo/memory.max
$ echo $$   | sudo tee /sys/fs/cgroup/demo/cgroup.procs
$ cat /sys/fs/cgroup/demo/memory.current
```

- The current shell *and all its future children* are now limited to 100 MiB
- To clean up: move the processes out, then `rmdir demo` (`rmdir`, not `rm -r`)
- This is exactly what Podman and Docker do for each container

Engine Flags Are Just cgroup Files

Podman / Docker flag	cgroup v2 file	Value written
`--memory=16m`	`memory.max`	16777216
`--memory=16m --memory-swap=32m`	`memory.swap.max`	16777216 (RAM + swap = 32m)
`--memory-reservation=8m`	`memory.low`	8388608
`--cpus=1.5`	`cpu.max`	150000 100000 (quota, period: µs)
`--cpu-shares=512`	`cpu.weight`	relative share (default 100)
`--cpuset-cpus=0,1`	`cpuset.cpus`	0-1
`--pids-limit=100`	`pids.max`	100 (Podman default: 2048)
`--device-read-bps`	`io.max`	8:0 rbps=1048576

- `pids.max` stops fork bombs; `/dev` access: an eBPF filter (Devices, below)
- Same knobs: Compose `mem_limit`, Quadlet `PidsLimit=`, systemd `MemoryMax=`
- Rootless: only *delegated* controllers (`cpu io memory pids`)

CPU and cpuset Controllers

CPU
- Accounting: `cpu.stat` (`usage_usec`, `nr_throttled`...)
- `cpu.weight`: relative share when the CPU is busy (default 100)
- `cpu.max`: "*quota period*" in µs: `50000 100000` = 50 % of one core; `max` = unlimited

```
$ podman run --rm --cpus=0.5 docker.io/library/alpine:3.20 \
      cat /sys/fs/cgroup/cpu.max
50000 100000
```

Cpuset
- Pin a group to CPUs (`cpuset.cpus`) and NUMA memory nodes (`cpuset.mems`)
- Use cases: dedicated cores (no cache flushes), NUMA performance

Block I/O, Network and PIDs

I/O (`io`, formerly `blkio`)
- `io.stat`: bytes and operations read/written, per device
- `io.max`: per-device hard limits (`rbps`, `wbps`, `riops`, `wiops`); `io.weight`
- v2 also accounts buffered writes (in v1 they went through the page cache unthrottled)

Network
- v1 `net_cls` / `net_prio` are gone in v2: use eBPF or nftables matching on the cgroup path

PIDs
- `pids.max` / `pids.current`: protects against fork bombs `:(){ :|:& };:`
- `podman run --pids-limit=100` (Podman's default limit is 2048)

Devices

- Containers must not open disks directly (`/dev/sda*`, `/dev/nvme*`...)
- But some devices must always work: `/dev/null`, `/dev/zero`, `/dev/random`, `/dev/tty`
- **v1**: a `devices` controller, e.g. `echo 'c 1:3 mr' > devices.allow`
- **v2**: no file! The runtime attaches an eBPF program (`BPF_PROG_TYPE_CGROUP_DEVICE`); inspect with `sudo bpftool cgroup tree`

Interesting nodes to pass with `--device`
- `/dev/net/tun` (VPNs), `/dev/fuse` (user-space filesystems), `/dev/kvm` (VMs in containers), `/dev/dri` (GPU), `/dev/ttyUSB*` (serial)

Capabilities

Root's power, split
- ≈ 40 flags: `CAP_CHOWN`, `CAP_NET_BIND_SERVICE` (ports < 1024), `CAP_SYS_ADMIN` ("the new root")...
- Podman keeps 11 by default

Least privilege
- `--cap-drop=all`, then `--cap-add` only what is needed (also in Compose)
- Never `--privileged`: all capabilities, all devices

```
$ podman run --rm docker.io/library/alpine:3.20 \
      grep CapEff /proc/self/status
CapEff: 00000000800405fb                capsh --decode it
$ podman run --rm --cap-drop=all docker.io/library/alpine:3.20 \
      chown nobody /tmp
chown: /tmp: Operation not permitted    root, but without CAP_CHOWN
```

Seccomp and `no_new_privs`

Seccomp-bpf (*secure computing*)
- A BPF filter on system calls (by number and arguments)
- Engines ship a default profile blocking rare, risky calls (`kexec_load`, `init_module`, `reboot`… → EPERM)

`no_new_privs`
- `execve` can never gain privileges (setuid bits, file capabilities ignored)
- `--security-opt=no-new-privileges`; Quadlet `NoNewPrivileges=true`

```
$ podman run --rm --security-opt=no-new-privileges \
      docker.io/library/alpine:3.20 \
      grep -E 'NoNewPrivs|Seccomp:' /proc/self/status
NoNewPrivs: 1
Seccomp:    2                       2 = filter mode
```


Linux Security Modules (LSM)

- Mandatory access control: checks *every* operation, even root's
- **SELinux** (Fedora, RHEL): sVirt labels each container
  - *Type enforcement*: `container_t` processes only touch `container_file_t` files
  - *MCS separation*: a random category pair per container (`s0:c123,c456`) → A cannot read B's files
  - See it: `ps -eZ | grep container_t`
  - Hence `:z` / `:Z` on bind mounts (lecture 2)
- **AppArmor** (Debian, Ubuntu, SUSE): path-based profiles
- Engines set this up. Do not disable it to "fix" a permission error

Lab: Building Image Layers by Hand

Ex. 1.1 sandbox (root of your *own* user + mount namespaces): `mount` without `sudo`.

```
# mkdir -p /tmp/overlay-lab && mount -t tmpfs tmpfs /tmp/overlay-lab
# cd /tmp/overlay-lab && mkdir lower1 lower2 upper work merged
# echo "Ubuntu 24.04 Base Layer"           > lower1/os-release.txt
# echo "Config v1 (from base layer)"       > lower1/app.conf
# echo "Secret token to be deleted"        > lower1/secret.txt
# echo "Config v2 (overridden by layer 2)" > lower2/app.conf
# echo "App binary v1.0"                   > lower2/app.bin
# mount -t overlay overlay \
    -o lowerdir=./lower2:./lower1,upperdir=./upper,workdir=./work \
    ./merged
```

`lower1` = base OS layer; `lower2` = app layer, on top (leftmost)

Lab: Module 1, Ex. 1.2 (steps 1–3)

Where Are the Layers?

```
$ podman info --format '{{.Store.GraphDriverName}} {{.Store.GraphRoot}}'
overlay /home/alice/.local/share/containers/storage
$ podman inspect --format '{{.GraphDriver.Data.UpperDir}}' sleeper
/home/alice/.local/share/containers/storage/overlay/39fb6.../diff
$ podman inspect --format '{{.GraphDriver.Data.LowerDir}}' sleeper
```

- Rootless: `~/.local/share/containers/storage/overlay/<layer>/diff` rootful Podman: `/var/lib/containers/storage`; Docker: `/var/lib/docker/overlay2`
- Kernel OverlayFS inside the user namespace (Linux ≥ 5.13)
- Files belong to your sub-UIDs: use `podman unshare ls -l`

Lab: Module 1, Ex. 1.2

`[Container]`: `podman run` Options, Declaratively

`Image=`	pinned FQIN (required by `AutoUpdate`)
`ContainerName=`	`--name`
`Network=....network`	`--network` + **start the network first**
`PublishPort=`	`-p`, here on **loopback only**
`PidsLimit=128`	cgroup `pids.max`: stops a fork bomb
`ReadOnly=true`	read-only root filesystem: no webshell or backdoor can be written
`NoNewPrivileges=true`	`setuid` binaries (`sudo`…) cannot gain privileges
`AutoUpdate=registry`	opt in to `podman auto-update`

Defense in depth: even after an RCE, the attacker can neither write, escalate, nor exhaust the host.

Lab: Module 5, Ex. 5.2

Automatic Updates

```
podman auto-update --dry-run      # what would be updated?
systemctl --user enable --now podman-auto-update.timer   # daily
```

For each unit with `AutoUpdate=registry`:
1. compare the local image **digest** with the registry's for the same **tag**
2. if changed: `pull`, then `systemctl restart` the unit
3. if the new container fails to start: **roll back** to the previous image (a failing *health check* counts only with `HealthCmd=` + `Notify=healthy`)

- Needs a fully-qualified image name
- It follows a tag: an exact version (`:v2.15.0`) rarely moves, `:latest` may break you

Lab: Module 5, Ex. 5.3

Pods: Containers Sharing a Network Namespace

- A **pod** = containers sharing one network namespace: same `localhost`
- An **infra container** holds the namespaces; members restart independently
- Ports are published by the pod (as in Kubernetes, where the pod is the unit)

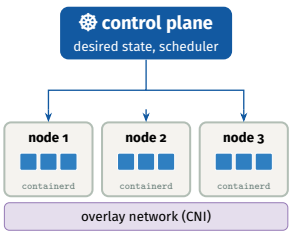
```
podman pod create --name web \
    -p 127.0.0.1:8080:8080
podman run -d --pod web --name app \
    docker.io/mccutchen/go-httpbin:v2.15.0
podman run --rm --pod web \
    docker.io/library/alpine:3.20 \
    wget -qO- http://localhost:8080/ip
podman ps -a --pod    # + infra container
```

Quadlet: `web.pod` + `Pod=`

```
# web.pod
[Pod]
PublishPort=127.0.0.1:8080:8080
# app.container
[Container]
Image=docker.io/mccutchen/go-httpbin:v2.15.0
Pod=web.pod
```


Beyond One Host

An **orchestrator** runs containers on a cluster:
- **scheduling**: which node runs what
- scaling, rolling updates, node failures
- service discovery, load balancing
- **multi-host networking**: overlay (VXLAN), CNI plugins (Flannel, Calico, Cilium)



Kubernetes: you declare the desired state ("3 replicas of this image"); controllers reconcile it continuously: systemd units for a whole cluster.
Also: Docker Swarm, Nomad; k3s, kind. From Podman: `podman kube` (next slide).

From Podman to Kubernetes

Podman speaks Kubernetes **YAML**:

```
podman kube generate --type deployment quadlet-httpbin \
    > httpbin.yaml                  # export
podman kube play httpbin.yaml       # run it
podman kube play --down httpbin.yaml    # stop it
```

`httpbin.kube` (Quadlet)

```
[Kube]
Yaml=httpbin.yaml
[Install]
WantedBy=default.target
```

- default output: a bare `Pod`; a `Deployment` is what a cluster restarts and scales
- the same YAML runs under systemd on one host…
- …and with `kubectl apply` on a cluster