DIHSÜD
Digital Innovation Hub

DIGITALISIERUNG FÜR KMU MÖGLICH MACHEN

DER DIGITAL INNOVATION HUB SÜD ALS KOSTENLOSES SERVICE FÜR KMU

Der DIH SÜD
wird unterstützt von:

FFG Forschung wirkt.
Das Land Steiermark
LAND KÄRNTEN

Über mich

Manfred Pamsl

Technische Mathematik / Informations- und Datenverarbeitung an der TU Graz

Softwareentwicklung und IT Lösungen in der Telekommunikations-Branche

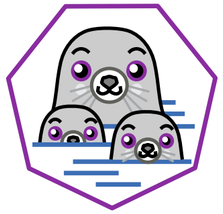
Lehr- und Forschungsfokus

- Cloud Computing
- Server-Security

DIHSÜD
Digital Innovation Hub

FH | JOANNEUM
University of Applied Sciences

Application Container Technologies



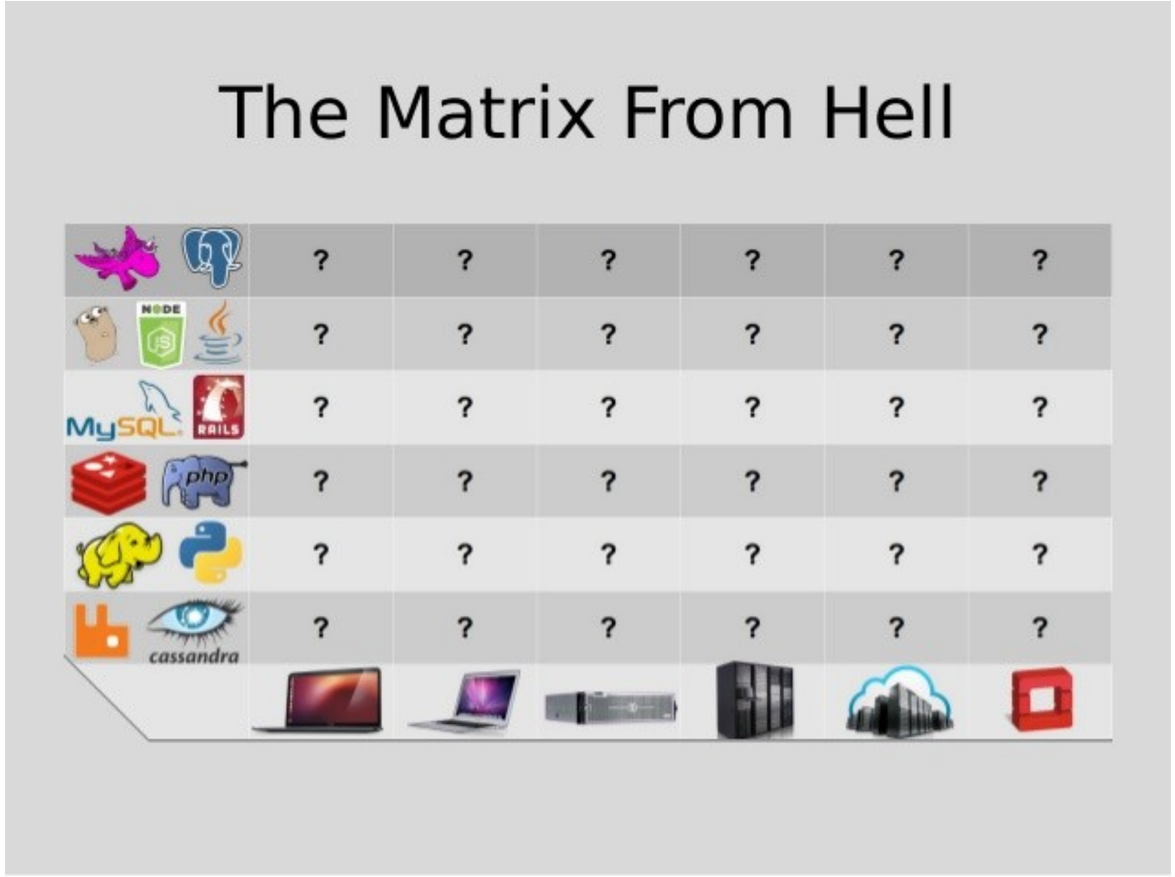
podman

Motivation

Software **deployment** and **installation** challenge

- Multiplicity of **software stacks**: static website, user database, web frontend, API endpoints, …
- Multiplicity of **hardware environments**: development VM, production cluster, QA server, customer data center, …

Each application must run in different hardware and software environments



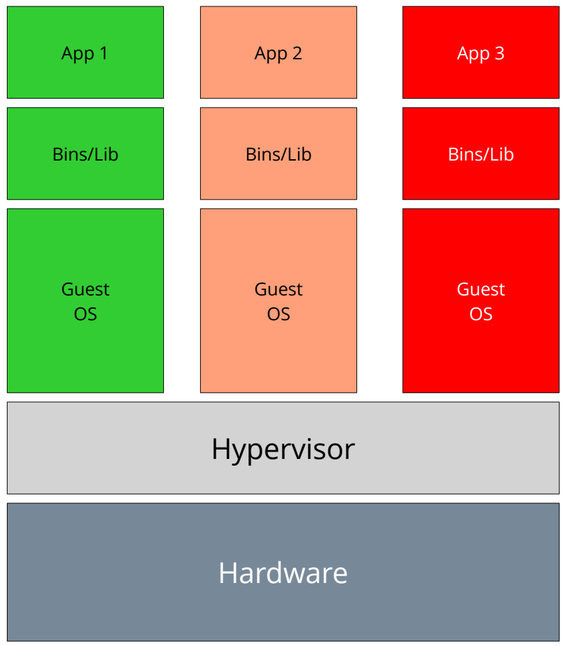
Source: https://www.slideshare.net/Docker/docker-lpc-2014cristian

“Matrix from Hell” Solution Approach

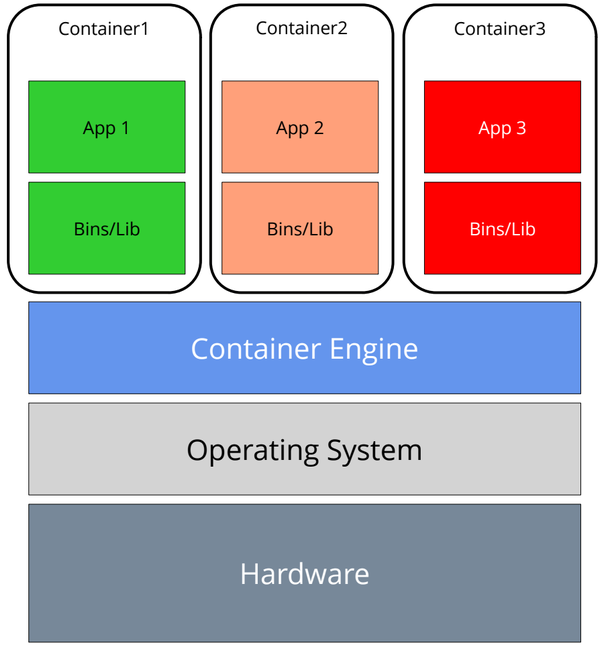
Build a **shipping container for application code**

- Provides a **portable run-time environment** for the application
  - Combines **code**, **libraries**, **data**, **package manager**
  - Avoids conflicting or missing libraries
- Provides **common management capabilities** for applications
  - Start, stop, migrate, logging, monitoring, network configuration, …

Containers versus Virtual Machines

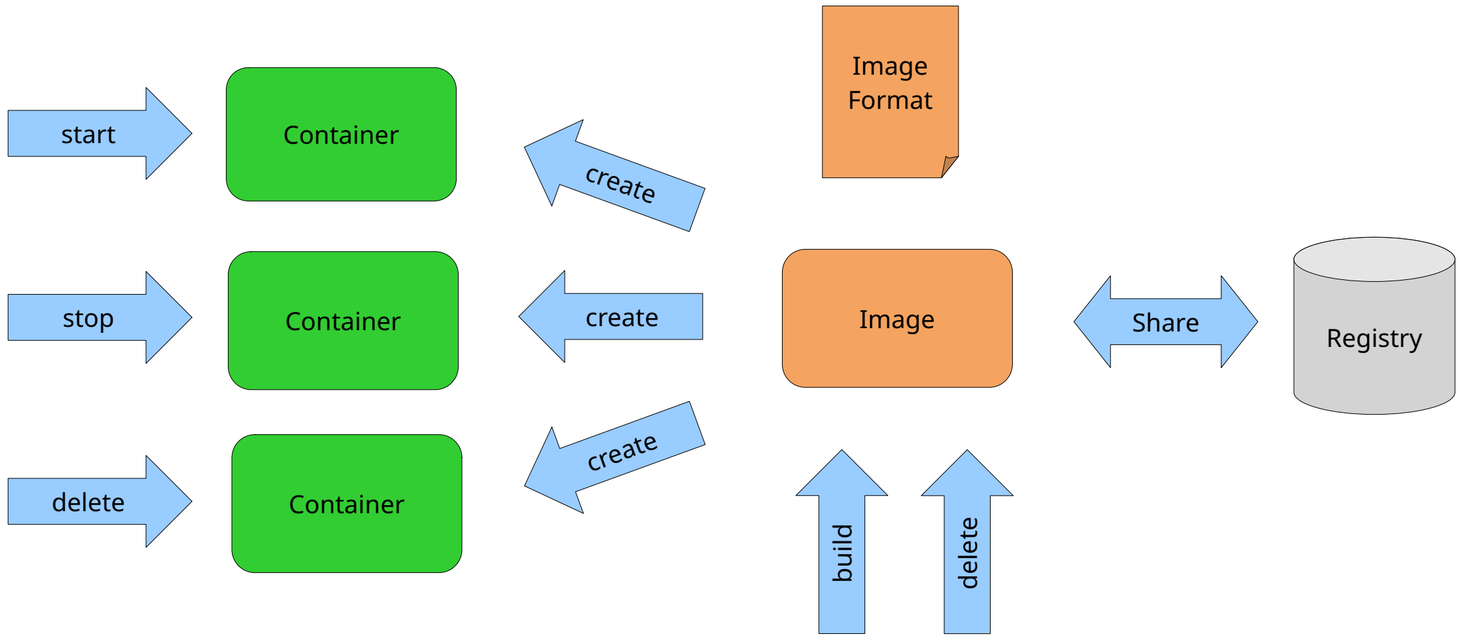


Virtual Machines



Containers

Container Concepts Overview



Container Terms

Container

- Processes that are executed in an isolated environment of an operating system

Container Image

- Ready-to-run software package, containing everything needed to run the processes of a container

Container Engine

- Software accepting user requests to manage images and containers

Container Runtime

- Lower level component to instantiate containers, typically used by a container engine

Container Host

- System on which containers are executed

Container Terms

Registry

- A service that stores and distributes container images

Repository

- A collection of container images in a registry that have the same name, but different tags
- Repository names can also include namespaces that are separated by forward slash characters

Tag

- The tag for an image specifies its version

Image Address: [loginServerUrl]/[repository][:tag]

- Example: docker.io/busybox:latest

Linux OS Level Virtualization Features

Control groups (cgroups) for resource utilization

- CPU, memory, disk I/O, network bandwidth
- https://man7.org/linux/man-pages/man7/cgroups.7.html

Kernel namespaces

- Processes, network, mount points, user and group IDs, IPC, hostname
- https://man7.org/linux/man-pages/man7/namespaces.7.html

May be managed either directly, through `libvirt` or `systemd-nspawn`

Engine versus Runtime Responsibilities

Engine responsibilities

- Handling input from user or orchestration API
- Pulling images from a registry server
- Preparing a container mount point by expanding the image
- Preparing the metadata which will be passed to the container runtime
- Calling the container runtime

Runtime responsibilities

- Consuming the container mount point provided by the container engine
- Consuming the container metadata provided by the container engine
- Setting up control groups and namespaces
- Setting up SELinux policy or App Armor rules
- Starting the containerized processes

Open Container Initiative (OCI)

Runtime specification

- How to run a “filesystem bundle” that is unpacked on a disk

Image specification

- Open specification for container images
- Enables the creation of interoperable tools for building, transporting, and preparing a container image to run

OCI compatible container runtimes

- runc, crun, youki, ...

https://www.opencontainers.org/

Container Engines Overview

Docker Engine and Containerd

- Widespread, available on Linux and Windows Server
- Docker Engine uses Containerd as interface to an OCI-based container runtime
- Containerd also implements the Kubernetes Container Runtime Interface (CRI) API

Podman

- Daemon-less container engine, used mainly on Redhat-based Linux systems
- Only for direct user interaction, does not support the Kubernetes CRI API
- Containers may be created by the “root” user (“rootful”) or by a unprivileged user (“rootless”)

CRI-O

- OCI-based implementation of the Kubernetes CRI API
- Usually not directly used from the command line

Container Image Registries

Registries are storing image repositories

Used to **share** and **collaborate** on images

- Images can be **pushed** to or **pulled** from a registry
- https://specs.opencontainers.org/distribution-spec/

Available as a commercial hosted services or as deployment on-premise, e.g.

- https://hub.docker.com/

Image and Container Layers

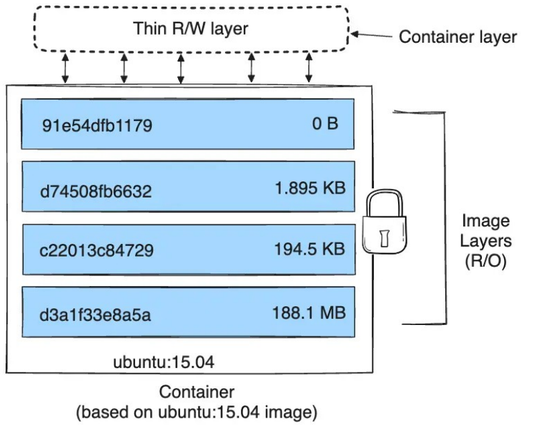
Container images are built from a series of **read-only layers** identified by a UUID

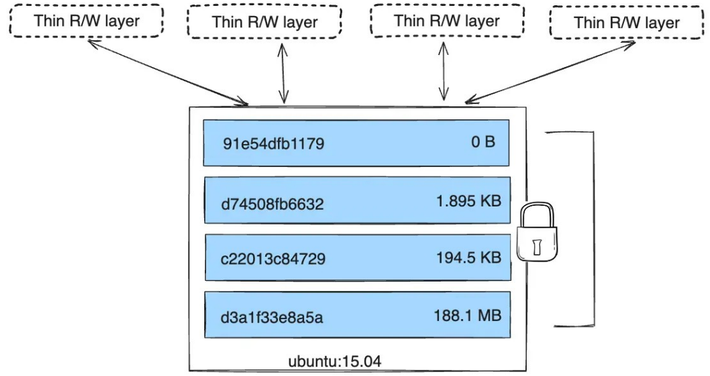
- Layers are **stacked** on top of each other and presented as a single uniform view
- Changes to the image are added as an additional layer

Containers are built from an image with an additional thin **writeable layer**

Different storage implementations are available

- OverlayFS, device mapper, BTRFS, ...





Source: https://docs.docker.com/storage/storagedriver/

Build Images

Images may be built by instructions from a “**Dockerfile**” and a “**context**” (local directory or Git URL)

- Alternatively, an image may be exported from an existing container

Files in the context are available for the build process (e.g. binaries and libraries)

The Dockerfile specifies how to build the image:

- Base image
- Files to add from the context
- Commands to run during build process (e.g. `apt install ...`)
- The executable to run in the resulting container

Basic Dockerfile Syntax

Full reference: https://docs.docker.com/engine/reference/builder/

```
FROM <image>
```

- Defines the base image, use “scratch” if starting from scratch

```
COPY ["<src>", … "<dest>"]
```

- Copy local files, local directories from the context to the image

```
RUN  ["<executable>", "<param1>", … ]
```

- Run command in a new layer on top of the image and commit the results

```
CMD  ["<executable>", "<param1>", … ]
```

- Specify default command when executing the container, may be overwritten on the command line

```
EXPOSE <port> [<port> …]
```

- Informs the container engine that the container will listen on the specified ports at runtime

Dockerfile / Build Example

```
# Dockerfile

FROM busybox:latest

RUN ["mkdir", "/html"]

COPY ["index.html", "/html"]

CMD ["/bin/httpd", "-f", \

     "-h", "/html", "-v"]

EXPOSE 80
```

```
# docker build -f Dockerfile -t myhttpd:latest .
Sending build context to Docker daemon  3.072kB
Step 1/5 : FROM busybox:latest
 ---> 2fb6fc2d97e1
Step 2/5 : RUN ["mkdir", "/html"]
 ---> Running in 02882efce428
Removing intermediate container 02882efce428
 ---> 60f80a108bd2
Step 3/5 : COPY ["index.html", "/html"]
 ---> 1365e68bf281
Step 4/5 : CMD ["/bin/httpd", "-f", "-h", "/html", "-v"]
 ---> Running in 4d05490be56e
Removing intermediate container 4d05490be56e
 ---> 5cd60a0bf140
Step 5/5 : EXPOSE 80
 ---> Running in 17cdd7997e78
Removing intermediate container 17cdd7997e78
 ---> 2ed4f903fee1
Successfully built 2ed4f903fee1
Successfully tagged myhttpd:latest
```

Multi-stage Image Builds

Allows creating a smaller final images

- E.g. by using a separate build and deployment image

Uses multiple FROM statements in a Dockerfile

- Each FROM instruction can use a different base image and begins a new stage of the build
- Artifacts may be copied from one stage to another

```
# syntax=docker/dockerfile:1
FROM golang:1.21
WORKDIR /src
COPY <<EOF ./main.go
package main

import "fmt"

func main() {
  fmt.Println("hello, world")
}
EOF
RUN go build -o /bin/hello ./main.go

FROM scratch
COPY --from=0 /bin/hello /bin/hello
CMD ["/bin/hello"]
```

Source: https://docs.docker.com/build/building/multi-stage/

Basic Container Engine Commands

Get help on the docker command

```
# docker --help
# docker <sub-command> \
  --help
```

Command syntax example

```
# docker image ls
```

https://docs.docker.com/engine/reference/commandline/cli/

Get help on the podman command

```
# podman --help
# podman <sub-command> \
  --help
# man podman
# man podman-<subcommand>
```

Command syntax example

```
# podman image ls
```

https://docs.podman.io/en/latest/Commands.html

Default Container Networks

bridge

- Virtual bridge shared between containers and the host
- Egress (outgoing) NAT and ingress (incoming) port forwarding from the host interfaces to the container (port “publishing”)

none

- Only the loopback interface (“`lo`”) is available, containers are isolated from the external networks

host

- Same network namespace as the host, all network interfaces of the host are available

Custom Container Networks

macvlan

- Container has access to a physical interface of the host via a sub-interface with its own MAC and IP address
- Exposes a single host interface to the container (requires promiscuous mode on the “parent” host interface)

ipvlan

- Like macvlan, but all sub-interfaces share the same MAC address
- Useful in case the number of MAC addresses on the external switch port is limited, or in case traffic filtering by the host `netfilter` chains in the default namespace is needed

slirp4netns

- Default network setup for “rootless” containers created by Podman, since unprivileged users are not allowed to create network interfaces
- Connects the container’s network namespace to a usermode TCP/IP stack

Managing Application Data

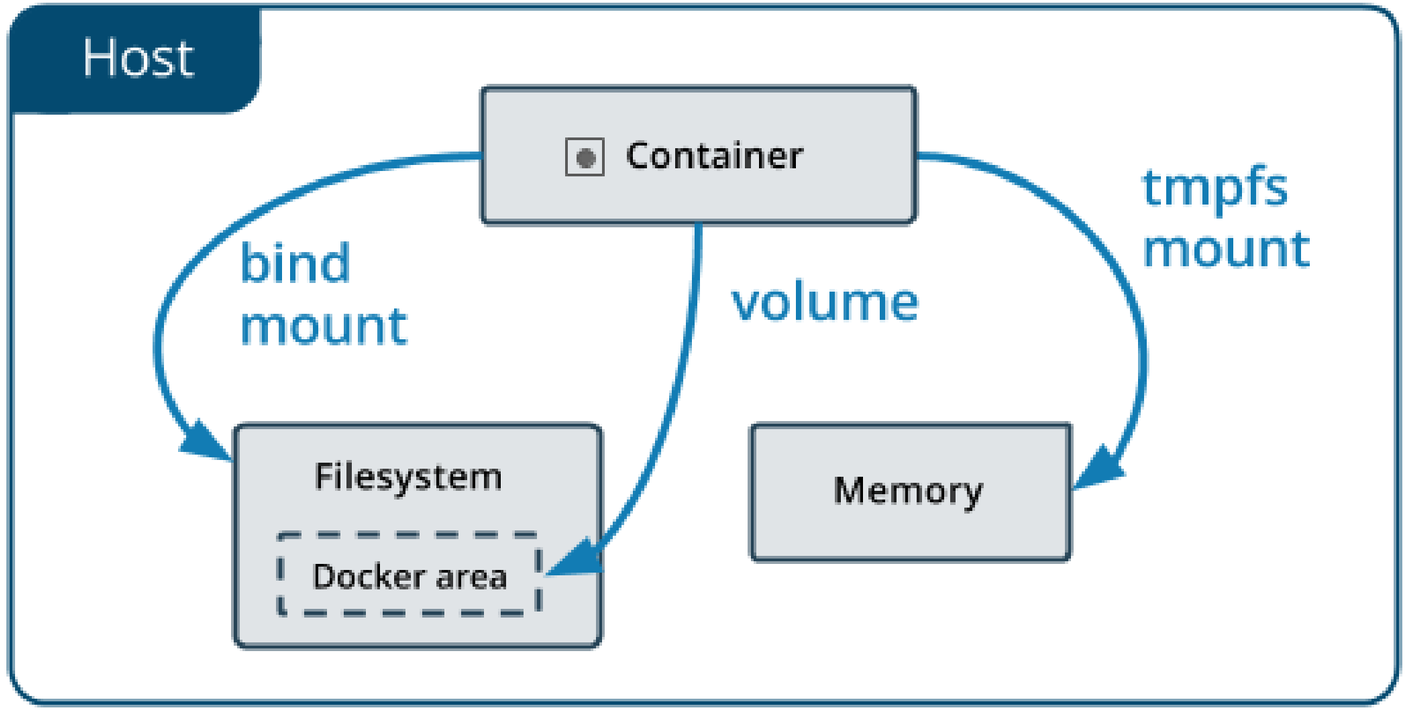
Drawbacks of storing application data in the writeable layer of a container

- Data can not easily be accessed from outside the container (e.g. for backup reasons)
- Data is lost, if the container is destroyed (Data is not lost, if the container is only stopped)

Container engines provide several ways to mount data from the host filesystem into the container

- Separates the lifetime of the application data from the container lifetime

Mount Types Overview



Source: https://docs.docker.com/engine/admin/volumes/

Volumes

Volumes are created and managed by the container engine

- Stored in a part of the host file system managed by the container engine (e.g. `/var/lib/containers/storage/volumes`)
- Volumes can be mounted into multiple containers simultaneously
- Volumes are available, even if no container is using the volume

“Named volumes” are explicitly created and given a name by the user

“Anonymous volumes” implicitly created and given a name by the container engine

Bind Mounts

A file or directory on the host is mounted into a container

- Non-empty target directory is obscured by the bind mount
- Can also be used to replace/add single files (instead of directories)

The file or directory is referenced by its full or relative path on the host machine

- The file or directory within the container does not need to exist, it is created on demand

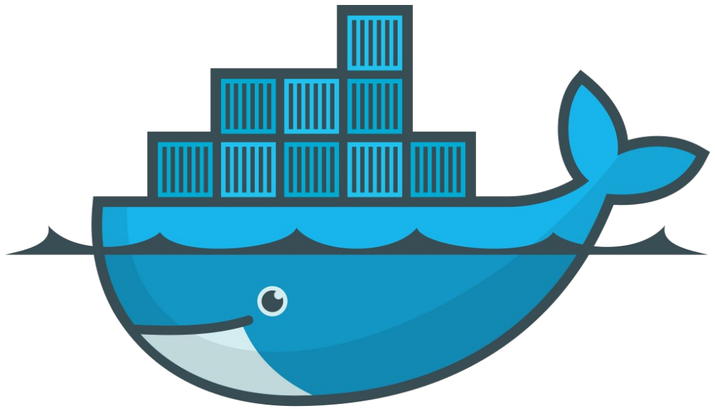
TMPFS Mount

Data is stored in the memory of the host machine, not in a persistent storage location

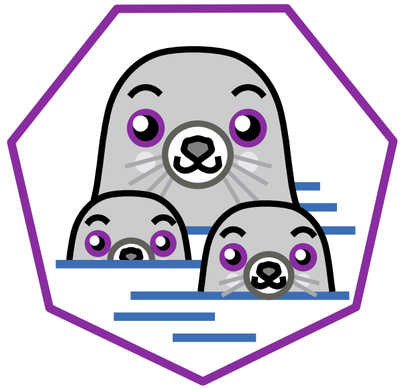
- If a container is stopped, the mount is removed and the data is lost
- If an image is built from the container, the data in the TMPFS mount is not included in the image

TMPFS mounts can not be shared among containers

Any Questions So Far ?



docker



podman

Further Reading

https://podman.io

https://www.docker.com

https://developers.redhat.com/blog/2018/02/22/container-terminology-practical-introduction

https://docs.microsoft.com/en-us/azure/container-registry/container-registry-concepts

https://www.capitalone.com/tech/cloud/container-runtime/

Basic Podman Commands (1)

Get help on the podman command

```
# podman --help
# podman <sub-command> --help
# man podman
# man podman-<subcommand>
```

Display system information

```
# podman info
```

Search configured registry services for images matching the term “busybox”

```
# podman search busybox
```

Pull the image “busybox” with tag “latest” in namespace “library” from the “docker.io” registry service

```
# podman pull \
docker.io/library/busybox:latest
```

Push “myimage” to a registry service

```
# podman push myreg:5000/myimage
```

List local images

```
# podman image ls
```

Remove a locally stored image “myimage” (but not from the registry)

```
# podman image rm myimage
```

Basic Podman Commands (2)

Create an interactive (`-i`, keeps STDIN open) container named “fed1” from image “fedora” to run command “`bash`”, allocate a Pseudo-TTY (`-t`)

```
# podman container create -i -t --name fed1 fedora bash
```

Start container with name “fed1”

```
# podman container start fed1
```

Attach your terminal to a running container “fed1” (detach with `<ctrl>-p <ctrl-q>`)

```
# podman container attach fed1
```

Stop container with name “fed1” by sending the `SIGTERM` signal to the processes within the container

```
# podman container stop fed1
```

Send the `SIGKILL` signal to a container with name “fed1”

```
# podman container kill fed1
```

Remove container “fed1” from local repository

```
# podman container rm fed1
```

Basic Podman Commands (3)

Create and run a detached (`-d`) container from image “httpd” and publish (`-p`, forward) port 1080 from host interface 10.0.0.1 to port 80 in the container

```
# podman container run -d -p 10.0.0.1:1080:80 --name web1 httpd
```

Same as above, but publish all exposed ports (`-P`) to random ports on the host interfaces

```
# podman container run -d -P httpd
```

List all (`-a`) container, even if they are not running

```
# podman container ls -a
```

Execute command `/bin/bash` in a running container (interactive with pseudo-TTY)

```
# podman container exec -it web1 /bin/bash
```

Create a new image “httpd” with tag “2.6” from the changes in container “web1”

```
# podman container commit web1 httpd:2.6
```

Copy file `/etc/httpd.conf` from container web1 to the host directory `/tmp`

```
# podman container cp web1:/etc/httpd.conf /tmp
```

Volume Management

Create a named volume

```
# podman volume create myvol
```

List volumes

```
# podman volume ls
```

Inspect a volume

```
# podman volume inspect myvol
```

Remove a volume

```
# podman volume rm myvol
```

Remove volumes not used by a running container

```
# podman volume prune
```

Start a Container with a Volume

Option “-v” or “--mount” are used to start a container with a volume

```
# podman run -d  -it --name devtest \
  --mount source=myvol2,target=/app \
  nginx:latest
# podman run -d -it --name devtest \
  -v myvol2:/app \
  nginx:latest
```

If a container is started with a volume that does not already exist, the volume is automatically created