Red Hat
Training and
Certification

DO180 Supplmentary Activities

Andrew Blum

Version 1.0

Table of Contents

DO180 supplementary activities are designed to expand on topics presented in the standard commercial course. These activities can be completed within the delivery of a DO180 course in addition to the typical hands-on activities.

The content topics are based on learner feedback and questions asked during various DO180 VT deliveries.

Chapter 1. Chapter 1: Introducing Container Technology

1.1. What types of container formats exist?

- lxd, used by lxc runtime https://linuxcontainers.org/
- aci from appc https://github.com/appc/spec/blob/master/spec/aci.md#app-container-image used by rkt from coreos
- docker (v1) deprecated 2/28/2017 https://github.com/moby/moby/blob/master/image/spec/v1.md
- docker (2v1) overly complicated b/c of backwards compat with docker v1 https://github.com/docker/distribution/blob/master/docs/spec/manifest-v2-1.md
- docker (2v2) https://github.com/docker/distribution/blob/master/docs/spec/manifest-v2-2.md
- oci (originally based on docker 2v2) https://github.com/opencontainers/image-spec/blob/master/spec.md

Listing 1. Inspecting the format type of an image

```
$ skopeo inspect docker://k8s.gcr.io/pause
$ podman pull k8s.gcr.io/pause:latest
$ podman inspect k8s.gcr.io/pause:latest | grep Manifest
        "ManifestType": "application/vnd.docker.distribution.manifest.v2+json",
```

1.2. What types of container formats are supported by the public cloud vendors?

- Google cloud: https://cloud.google.com/container-registry/docs/image-formats
- Azure: https://docs.microsoft.com/en-us/azure/container-registry/container-registry-image-formats
- aws: https://docs.aws.amazon.com/AmazonECR/latest/userguide/image-manifest-formats.html

1.3. What are namespaces?

Namespaces are responsible for resource isolation. Think of the view-master which can alter what you see when you view through it.

From `man 7 namespaces`:

```
A  namespace  wraps  a global system resource in an abstraction that makes it appear
to the processes within the namespace that they have their own
isolated instance of the global resource.  Changes to the global resource are visible
to other processes that are members of the namespace, but are
invisible to other processes.  One use of namespaces is to implement containers.
```

Namespace	Flag	Page	Isolates
Cgroup	CLONE_NEWCGROUP	cgroup_namespaces(7)	Cgroup root directory
IPC	CLONE_NEWIPC	ipc_namespaces(7)	System V IPC and POSIX message queues
Network	CLONE_NEWNET	network_namespaces(7)	Network devices, stacks, ports, etc.
Mount	CLONE_NEWNS	mount_namespaces(7)	Mount points
PID	CLONE_NEWPID	pid_namespaces(7)	Process IDs
Time	CLONE_NEWTIME	time_namespaces(7)	Boot and monotonic clocks
User	CLONE_NEWUSER	user_namespaces(7)	User and group IDs
UTS	CLONE_NEWUTS	uts_namespaces(7)	Hostname and NIS domain name

Listing 2. Source Running a process in a unique pid namespace

```
[student@workstation ~]$ unshare -Urpf --mount-proc
[root@workstation ~]# sleep 9000 &
[1] 32
[root@workstation ~]# ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 11:48 pts/1    00:00:00 -bash
root          32       1  0 11:49 pts/1    00:00:00 sleep 9000
root          33       1  0 11:49 pts/1    00:00:00 ps -ef
```

Now, open a different terminal and run:

Listing 3. Source Inspecting a process run in a unique pid namespace

```
[student@workstation ~]$ ps -ef | grep 9000
student     4497     4466  0 11:49 pts/1    00:00:00 sleep 9000
student     4583     4414  0 11:51 pts/2    00:00:00 grep --color=auto 9000
```

Notice the different pid for this same sleep process as seen from the global pid namespace (ie pid 4497 vs 32)

1.4. What are control groups?

Control groups limit what resources a process group can consume. Allows processes to be organized into hierarchical groups so that usage of particular resources can be limited and monitored.

There are different cgroup versions supported. In more recent RHEL releases (RHEL9), cgroupv2 is used. A quick way to determine which cgroup version your system is using is by inspecting the `mount` output.

Systems using cgroup v1 will mount the different cgroup hierarchies like:

```
[student@workstation ~]$ mount | grep cgroup
cgroup on /sys/fs/cgroup/systemd type cgroup
(rw,nosuid,nodev,noexec,relatime,seclabel,xattr,release_agent=/usr/lib/systemd/systemd-
cgroups-agent,name=systemd)
cgroup on /sys/fs/cgroup/cpu,cpuacct type cgroup
(rw,nosuid,nodev,noexec,relatime,seclabel,cpu,cpuacct)
cgroup on /sys/fs/cgroup/freezer type cgroup
(rw,nosuid,nodev,noexec,relatime,seclabel,freezer)
cgroup on /sys/fs/cgroup/memory type cgroup
(rw,nosuid,nodev,noexec,relatime,seclabel,memory)
cgroup on /sys/fs/cgroup/hugetlb type cgroup
(rw,nosuid,nodev,noexec,relatime,seclabel,hugetlb)
cgroup on /sys/fs/cgroup/rdma type cgroup (rw,nosuid,nodev,noexec,relatime,seclabel,rdma)
cgroup on /sys/fs/cgroup/net_cls,net_prio type cgroup
(rw,nosuid,nodev,noexec,relatime,seclabel,net_cls,net_prio)
cgroup on /sys/fs/cgroup/cpuset type cgroup
(rw,nosuid,nodev,noexec,relatime,seclabel,cpuset)
cgroup on /sys/fs/cgroup/devices type cgroup
(rw,nosuid,nodev,noexec,relatime,seclabel,devices)
cgroup on /sys/fs/cgroup/blkio type cgroup
```

```
(rw,nosuid,nodev,noexec,relatime,seclabel,blkio)
cgroup on /sys/fs/cgroup/perf_event type cgroup
(rw,nosuid,nodev,noexec,relatime,seclabel,perf_event)
cgroup on /sys/fs/cgroup/pids type cgroup (rw,nosuid,nodev,noexec,relatime,seclabel,pids)
```

Newer systems will use cgroup v2 by default which includes a single mount point providing a single single control group hierarchy against which all resource controllers are mounted.

```
[user1@rhel9 ~]$ mount | grep cgroup
cgroup2 on /sys/fs/cgroup type cgroup2
(rw,nosuid,nodev,noexec,relatime,seclabel,nsdelegate,memory_recursiveprot)
```

Note free doesn't report cgroup memory accounting

Memory accounting reported in `/proc/meminfo` is NOT namespaced. So, a container's view of memory from tools like free/top will show the system accounting vs /sys/fs/cgroup/memory/memory.usage_in_bytes https://ops.tips/blog/why-top-inside-container-wrong-memory/

1.5. What is seccomp?

Seccomp limits what system calls a process can make… even if running as root ! Seccomp means “secure computing mode” From `man 2 seccomp`:

> Limits what system calls by either read(), write(), _exit(), and sigreturn() or by a list of allowed calls given as “filters” The Seccomp field of the /proc/[pid]/status file provides a method of viewing the seccomp mode of a process

Typically the seccomp mode for an uncontainerized process is disabled:

```
[student@workstation ~]$ grep -i seccomp /proc/3421/status
Seccomp:    0
```

0 means SECCOMP_MODE_DISABLED; 1 means SECCOMP_MODE_STRICT; 2 means SECCOMP_MODE_FILTER

In SECCOMP_MODE_STRICT, it cannot use any system calls except exit(), sigreturn(), read() and write().

In SECCOMP_MODE_FILTER, since linux 3.5, it is possible to define advanced custom filters based on the BPF (Berkley Packet Filters) to limit what system calls and their arguments can be used by the process.

Compare containers running unconfined seccomp policy with the default seccomp policy.

```
[student@workstation ~]$ podman run -d --security-opt=seccomp=unconfined ubi8:latest
sleep 9000
84c0cf6071e0e7370f1b125c419f508f170843abfc97594e6560dbb81f8c0cff
[student@workstation ~]$ ps -ef | grep 9000
student     2199     2188  0 14:58 ?        00:00:00 /usr/bin/coreutils --coreutils-prog
-shebang=sleep /usr/bin/sleep 9000
student     2210     1886  0 14:58 pts/0    00:00:00 grep --color=auto 9000
[student@workstation ~]$ grep -i seccomp /proc/2199/status
Seccomp:    0
[student@workstation ~]$ podman run -d ubi8:latest sleep 8000
e42d14a6074419cae413301c8aa485efdd509cc08cd0ddeef067f5fea6496a81
[student@workstation ~]$ ps -ef | grep 8000
student     2238     2227  0 14:59 ?        00:00:00 /usr/bin/coreutils --coreutils-prog
-shebang=sleep /usr/bin/sleep 8000
student     2258     1886  0 14:59 pts/0    00:00:00 grep --color=auto 8000
[student@workstation ~]$ grep -i seccomp /proc/2238/status
Seccomp:    2
```

Containers use `/usr/share/containers/seccomp.json` as the default seccomp profile.

1.6. What are capabilites?

From `man 7 capabilities`

Capabilities allow a process to become privileged bypassing normal permssion checks.

> Privileged processes bypass all kernel permission checks… Linux divides the privileges traditionally associated with superuser into distinct units, known as capabilities, which can be independently enabled and disabled. Capabilities are a per-thread attribute.

A list and description for the various capabilities is found in `man 7 capabilities`. It's possible to set and list the capabilities available to a process (Bounding Capabilties) as well as those being used by the process (Effective Capabilities). Consider those on this uncontainerized sshd process:

```
[student@workstation ~]$ ps -ef | grep sshd
root        1049        1  0 08:11 ?        00:00:00 /usr/sbin/sshd -D
[student@workstation ~]$ grep Cap /proc/1049/status
CapInh:    0000000000000000
CapPrm:    000001ffffffffff
CapEff:    000001ffffffffff
CapBnd:    000001ffffffffff
```

```
CapAmb:    0000000000000000
[student@workstation ~]$ capsh --decode=000001ffffffffff
0x000001ffffffffff=cap_chown,cap_dac_override,cap_dac_read_search,cap_fowner,cap_fsetid,c
ap_kill,cap_setgid,cap_setuid,cap_setpcap,cap_linux_immutable,cap_net_bind_service,cap_ne
t_broadcast,cap_net_admin,cap_net_raw,cap_ipc_lock,cap_ipc_owner,cap_sys_module,cap_sys_r
awio,cap_sys_chroot,cap_sys_ptrace,cap_sys_pacct,cap_sys_admin,cap_sys_boot,cap_sys_nice,
cap_sys_resource,cap_sys_time,cap_sys_tty_config,cap_mknod,cap_lease,cap_audit_write,cap_
audit_control,cap_setfcap,cap_mac_override,cap_mac_admin,cap_syslog,cap_wake_alarm,cap_bl
ock_suspend,cap_audit_read,cap_perfmon,cap_bpf,cap_checkpoint_restore
```

- CapPrm = Permitted Capabilities
- CapBnd = Bounding Capabilities
- CapEff = Effective Capabilities

Capabilities can also be assigned to a file binary (getcap, setcap):

```
[student@workstation sbin]$ getcap /sbin/arping
/sbin/arping cap_net_raw=p
```

- e: Effective. This means the capability is “activated”.
- p: Permitted. This means the capability can be used/is allowed.
- i: Inherited. The capability is kept by child/subprocesses upon execve() for example

1.7. What container specific selinux labels are used ?

SELinux policy protects files and other processes on a system using a labeling system. It possible to view the labels on a process by running `ps -eZ`. The default labels used for containerized processes and files is defined in `/usr/share/containers/selinux/contexts`:

```
[student@workstation ~]$ cat /usr/share/containers/selinux/contexts
process = "system_u:system_r:container_t:s0"
file = "system_u:object_r:container_file_t:s0"
ro_file="system_u:object_r:container_ro_file_t:s0"
kvm_process = "system_u:system_r:container_kvm_t:s0"
init_process = "system_u:system_r:container_init_t:s0"
engine_process = "system_u:system_r:container_engine_t:s0"
```

1.8. What happended to the SELinux label svirt_sandbox_t?

This legacy label is still valid to use for containerized file access, but it is now an SELinux *alias*.

```
[root@workstation ~]# mkdir /testdir
[root@workstation ~]# chcon -t svirt_sandbox_file_t /testdir
[root@workstation ~]# podman run -it -v /testdir:/data rhel7 /bin/bash
[root@a8b47cb39617 /]# touch /data/test1
```

To determine the aliases used for with the container_file_t label use the `seinfo` command from the `setools-console` package:

```
[student@workstation ~]$ rpm -q selinux-policy
selinux-policy-3.14.3-95.el8.noarch
[student@workstation ~]$ sudo yum install setools-console -y
[student@workstation ~]$ seinfo -t container_file_t -x
Types: 1
   type container_file_t alias { svirt_sandbox_file_t svirt_lxc_file_t }, SNIP
```

1.9. What other selinux target contexts are allowed by selinux policy ?

SELinux policy does allow a process with scontext of `container_t` to access a tcontext of `container_file_t`. There are a number of other target contexts allowed as well.

To get a list of those install the selinux-policy-doc package and consult the `container_selinux` man page:

```
[student@workstation ~]$ sudo yum install selinux-policy-doc -y
[student@workstation ~]$ man container_selinux
MANAGED FILES
       The SELinux process type container_t can manage files labeled with the following
file types.

cephfs_t
cifs_t
container_file_t
fusefs_t
hugetlbfs_t
nfs_t
onload_fs_t
```

1.10. How to install podman and container-tools in RHEL ?

For RHEL7 systems,

```
[root@rhel7 ~]# subscription-manager repos --enable rhel-7-server-extras-rpms
[root@rhel7 ~]# yum install podman
```

For RHEL8 systems,

```
[root@workstation ~]# subscription-manager repos --list-enabled
This system has no repositories available through subscriptions.

[root@workstation ~]# yum repolist
Updating Subscription Management repositories.
Unable to read consumer identity
This system is not registered to Red Hat Subscription Management. You can use
subscription-manager to register.
repo id                                                                  repo name
rhel-8.2-for-x86_64-appstream-rpms                                       Red Hat
Enterprise Linux 8.2 AppStream (dvd)
rhel-8.2-for-x86_64-baseos-rpms                                          Red Hat
Enterprise Linux 8.2 BaseOS (dvd)

[root@workstation ~]# yum module list container-tools
Updating Subscription Management repositories.
Last metadata expiration check: 0:00:48 ago on Mon 19 Aug 2019 10:51:59 AM EDT.
Red Hat Enterprise Linux 8 for x86_64 - AppStream (RPMs)
Name                           Stream                        Profiles
Summary
container-tools                1.0                           common [d]
Common tools and dependencies for container runtimes
container-tools                rhel8 [d][e]                  common [d] [i]
Common tools and dependencies for container runtimes

Hint: [d]efault, [e]nabled, [x]disabled, [i]nstalled

[root@workstation ~]# yum module info container-tools

                 : buildah-0:1.11.6-7.module+el8.2.0+5856+b8046c6d.x86_64
                 : conmon-2:2.0.6-1.module+el8.2.0+5182+3136e5d4.x86_64
                 : podman-0:1.6.4-10.module+el8.2.0+6063+e761893a.src
                 : runc-0:1.0.0-65.rc10.module+el8.2.0+5762+aaee29fb.src
                 : skopeo-1:0.1.40-10.module+el8.2.0+5955+6cd70ceb.src
                 : slirp4netns-0:0.4.2-3.git21fdece.module+el8.2.0+5658+9a15711d.src
```

```
[root@workstation ~]# yum module install container-tools
```

1.11. Is there a module or group that can assist installation in Fedora ?

```
[root@badger ~]# cat /etc/redhat-release
Fedora release 34 (Thirty Four)

[root@badger ~]# dnf repolist
repo id                                                           repo name
fedora                                                            Fedora 34 - x86_64
fedora-cisco-openh264                                             Fedora 34 openh264 (From
Cisco) - x86_64
fedora-modular                                                    Fedora Modular 34 -
x86_64
google-chrome                                                     google-chrome
rpmfusion-free                                                    RPM Fusion for Fedora 34
- Free
rpmfusion-free-updates                                            RPM Fusion for Fedora 34
- Free - Updates
updates                                                           Fedora 34 - x86_64 -
Updates
updates-modular                                                   Fedora Modular 34 -
x86_64 - Updates

[root@badger ~]# dnf install -y @container-tools
Last metadata expiration check: 1:17:32 ago on Tue 21 Sep 2021 05:29:15 AM CDT.
Module or Group 'container-tools' is not available.
Error: Nothing to do.

Although there is no container-tools group or module in Fedora, there is a container-
management group:

[root@badger ~]# dnf groupinfo "Container Management"
Last metadata expiration check: 1:16:42 ago on Tue 21 Sep 2021 05:29:15 AM CDT.
Group: Container Management
 Description: Tools for managing Linux containers
 Default Packages:
   podman
 Optional Packages:
   buildah
   flatpak
   flatpak-builder
   origin-clients
```

```
[root@badger ~]# dnf install -y @container-management
```

Chapter 2. Chapter 2: Creating Containerized Services

2.1. How can you determine the available tags within a repository ?

The `podman search` command has an option `--list-tags` that will list out a number of different image tags available in a given repository.

```
[student@workstation ~]$ podman search registry.access.redhat.com/rhel7 --list-tags
NAME                                TAG
registry.access.redhat.com/rhel7  7.9-333
registry.access.redhat.com/rhel7  7.3-89
registry.access.redhat.com/rhel7  7.5-245.1527091554
registry.access.redhat.com/rhel7  7.9-401-source
registry.access.redhat.com/rhel7  7.3-82
...SNIP...
```

This command will list out the first 25 tags given by the registry API. Alternatively, communication with a registry API is possible using a tool like `curl`. Consider:

```
[student@workstation ~]$ curl  -L https://registry.access.redhat.com/v2/rhel7/tags/list
{"name": "rhel7", "tags": ["7.3-74", "7.4-120", "7.2-56", "7.3-89", "7.3-66", "7.5-424",
"7.5-245.1527091554", "7.4-129", "7.1-12", "7.6-122", "7.3-82", "7.7-384.1575996163",
"7.5-409.1533127727", "7.2-75", "7.2-38", "7.6", "7.7-348", "7.4", ...SNIP....]}
```

There is also a tool called `skopeo` that can be used to obtain details about a particular repository:

```
[student@workstation ~]$ skopeo inspect docker://registry.access.redhat.com/rhel7
{
    "Name": "registry.access.redhat.com/rhel7",
    "Digest": "sha256:17ef5565008d71f89c8444a8c1141aafd25bd0d6a1ccaecb5f34c517509f9036",
    "RepoTags": [
        "7.9-333",
        "7.3-89",
        "7.5-245.1527091554",
        "7.9-401-source",
```

Working directly with registries including different registry APIs is covered in greater detail in Chapter 4.

Now that we can identify an image within a repository that we want, let's pull a local copy using the **latest** tag:

```
[student@workstation ~]$  podman pull registry.access.redhat.com/rhel7:latest
9a3387c8f6bc9b63b119dc61ddbaed6bb20795a7b187908ca1b5ecabc5c19aac
```

How does that compare to:

```
[student@workstation ~]$  podman pull registry.access.redhat.com/rhel7
9a3387c8f6bc9b63b119dc61ddbaed6bb20795a7b187908ca1b5ecabc5c19aac
```

Notice they are identified by the same image id.

Your specific image IDs may differ than the examples given above. As the content of images change over time, so will their IDs.

What about a different version of rhel7?

```
[student@workstation ~]$  podman pull registry.access.redhat.com/rhel7:7.7
6682529ce3faf028687cef4fc6ffb30f51a1eb805b3709d31cb92a54caeb3daf
```

2.2. How to pull using a short-name alias ?

Consider:

```
[student@workstation ~]$ head /etc/containers/registries.conf.d/001-rhel-shortnames.conf
[aliases]
"3scale-amp2/3scale-rhel7-operator-metadata" = "registry.redhat.io/3scale-amp2/3scale-
rhel7-operator-metadata"
"3scale-amp2/3scale-rhel7-operator" = "registry.redhat.io/3scale-amp2/3scale-rhel7-
operator"
"3scale-amp24/wildcard-router" = "registry.redhat.io/3scale-amp24/wildcard-router

[student@workstation ~]$ grep rhel7 /etc/containers/registries.conf.d/001-rhel-
shortnames.conf
"rhel7/open-vm-tools" = "registry.access.redhat.com/rhel7/open-vm-tools"
"rhel7" = "registry.access.redhat.com/rhel7"
"rhel7/rhel-atomic" = "registry.access.redhat.com/rhel7/rhel-atomic"
"rhel7/rhel" = "registry.access.redhat.com/rhel7/rhel"

[student@workstation ~]$ podman pull rhel7:latest
Trying to pull registry.access.redhat.com/rhel7:latest...
```

```
Getting image source signatures

[student@workstation ~]$  podman images
REPOSITORY                         TAG       IMAGE ID       CREATED         SIZE
registry.access.redhat.com/rhel7  latest      2664aa19856f  2 weeks ago    216 MB
registry.access.redhat.com/rhel7  7.7         6682529ce3fa  22 months ago  215 MB
```

The shortname alias is available as a convenience.

2.3. How to run "My First Container"?

Two essential details needed to run a containerized process are:

1. The container image
2. The command to execute inside the container

```
[student@workstation ~]$  podman run --help
NAME:
   podman run - Run a command in a new container

USAGE:
   podman run [command options] IMAGE [COMMAND [ARG...]]
```

Here are some different ways to run the command `cat /etc/redhat-release`. Notice the different ways the container image is identified in each of the `podman run` commands:

```
[student@workstation ~]$ podman run registry.access.redhat.com/rhel7:latest cat
/etc/redhat-release
Red Hat Enterprise Linux Server release 7.9 (Maipo)

[student@workstation ~]$ podman run registry.access.redhat.com/rhel7:7.5 cat /etc/redhat-
release
Red Hat Enterprise Linux Server release 7.5 (Maipo)

[student@workstation ~]$ podman run registry.access.redhat.com/rhel7 cat /etc/redhat-
release
Red Hat Enterprise Linux Server release 7.9 (Maipo)

[student@workstation ~]$ podman run rhel7:7.7 cat /etc/redhat-release
Red Hat Enterprise Linux Server release 7.7 (Maipo)

[student@workstation ~]$ podman images
REPOSITORY                                        TAG          IMAGE ID      CREATED
SIZE
```

```
registry.access.redhat.com/rhel7                 latest      d19b7e812477   5 days ago
218 MB

[student@workstation ~]$ podman run d19b7e812477 cat /etc/redhat-release
Red Hat Enterprise Linux Server release 7.9 (Maipo)
```

What about running a container from an image that has not been *pulled* yet ?

```
[student@workstation ~]$ podman run rhel7:7.8 cat /etc/redhat-release
Resolved "rhel7" as an alias (/etc/containers/registries.conf.d/001-rhel-shortnames.conf)
Trying to pull registry.access.redhat.com/rhel7:7.8...
Getting image source signatures
Checking if image destination supports signatures
Copying blob b13ffc206103 done
Copying blob 872582724f33 done
Copying config 9da37a6819 done
Writing manifest to image destination
Storing signatures
Red Hat Enterprise Linux Server release 7.8 (Maipo)
```

Notice several images have been pulled and are available from your local container storage. List them with `podman images` like:

```
[student@workstation ~]$ podman images
REPOSITORY                          TAG         IMAGE ID      CREATED      SIZE
registry.access.redhat.com/rhel7  latest      c7344c9fb18c  3 weeks ago  216 MB
registry.access.redhat.com/rhel7  7.8         9da37a681956  2 years ago  215 MB
registry.access.redhat.com/rhel   7.7         6682529ce3fa  2 years ago  215 MB
registry.access.redhat.com/rhel7  7.7         6682529ce3fa  2 years ago  215 MB
```

Let’s take a closer look at the `cat /etc/redhat-release` commands that were executed using `podman run`.

Can we find those in ps output?

```
[student@workstation ~]$  ps -ef | grep cat
gdm       4068  3897  0 16:15 ?        00:00:00 /usr/libexec/gsd-print-notifications
root      7313  6645  0 18:30 pts/0    00:00:00 grep --color=auto ca
```

They are not running anymore. To get some extra information about all containers that are running or stopped, Use `podman ps -a` instead:

```
[student@workstation ~]$ podman ps
```

```
[student@workstation ~]$ podman ps -a
CONTAINER ID   IMAGE                                        COMMAND
CREATED                STATUS                             PORTS    NAMES                IS
INFRA
27a3d872ff92   registry.access.redhat.com/rhel:7.4        cat /etc/redhat-rel...   About
a minute ago   Exited (0) About a minute ago           confident_dubinsky   false

6905b9e93f0d   registry.access.redhat.com/rhel7:7.5-404   cat /etc/redhat-rel...   6
minutes ago        Exited (0) 6 minutes ago                dreamy_panini        false

de3fa91850ee   registry.access.redhat.com/rhel7:latest    cat /etc/redhat-rel...   6
minutes ago        Exited (0) 6 minutes ago                competent_bell       false
```

Some images have prebuilt commands that will be executed if the COMMAND is missing from the `podman run`:

```
[student@workstation ~]$  podman inspect registry.access.redhat.com/rhel7:latest | less
...SNIP...
"Cmd": [
     "/bin/bash"
],
...SNIP...
```

So what happens when we run this without giving any command to execute:

```
[student@workstation ~]$  podman run registry.access.redhat.com/rhel7:latest
[student@workstation ~]$
```

What just happened?... Did it fail? ps -a (no, just non-interactive)

```
[student@workstation ~]$  podman ps -a
CONTAINER ID   IMAGE                                        COMMAND
CREATED         STATUS                        PORTS    NAMES                IS INFRA
04e62d0683ff   registry.access.redhat.com/rhel7:latest    /bin/bash                5
seconds ago    Exited (0) 4 seconds ago           youthful_benz        false
```

To make the bash command interactive run:

```
[student@workstation ~]$  podman run -i registry.access.redhat.com/rhel7:latest
asdljflkasdjf
/bin/bash: line 3: asdljflkasdjf: command not found

echo hello world
```

```
hello world
whoami
root
tty
not a tty
exit
```

What about a terminal prompt? Let's add the -t flag to allocate a TTY

```
[student@workstation ~]$  podman run -it registry.access.redhat.com/rhel7:latest
[root@66f4d93191b7 /]# tty
/dev/pts/0
[root@66f4d93191b7 /]# exit
```

That looks much better.

Another useful option when running a container is the -d or detach, for example:

```
[student@workstation ~]$  podman run -d registry.access.redhat.com/rhel7:latest sleep
5000
829e8264c3f722e047002ebf9bf55b38fcc9b2be3b6f0a2afdfb4088d01a3a7f
[student@workstation ~]$
[student@workstation ~]$  podman ps
CONTAINER ID   IMAGE                                  COMMAND      CREATED          STATUS
PORTS   NAMES
829e8264c3f7   registry.access.redhat.com/rhel7:7.5   sleep 5000   4 seconds ago    Up 3
seconds ago          loving_montalcini
```

It would not make sense to run a container with **BOTH** -it (interactive+tty) and -d (detach). Think about it, you cannot be interactive and detached at the same time.

Sometimes we will want to set environment variables for use by the process in the container. Do this with `-e`:

```
[student@workstation ~]$  podman run registry.access.redhat.com/rhel7:latest env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
TERM=xterm
HOSTNAME=497c3c41f75f
container=oci
HOME=/root
[student@workstation ~]$ podman run -e FOO="hello world"
registry.access.redhat.com/rhel7:latest env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
TERM=xterm
```

```
HOSTNAME=c730b9772e9a
container=oci
FOO=hello world
HOME=/root
```

If you'd like to name a container use `--name` like:

```
[student@workstation ~]$ podman run -d --name mycontainer
registry.access.redhat.com/rhel7:latest sleep 5000
09b5adcfbcc0f894ef2d1782ebe5a28ba78e2bc67901814726b8d3b63f9545b5

[student@workstation ~]$  podman ps
CONTAINER ID   IMAGE                                                      COMMAND
CREATED         STATUS              PORTS   NAMES
09b5adcfbcc0   registry.access.redhat.com/rhel7:7.5                       sleep 5000
3 seconds ago   Up 2 seconds ago            mycontainer
```

To run a new command inside a running container use `podman-exec`:

```
[student@workstation ~]$  podman exec -it mycontainer /bin/bash
[root@db380c01c168 /]# ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 18:58 ?        00:00:00 sleep 5000
root         5     0  3 19:00 pts/0    00:00:00 /bin/bash
root        17     5  0 19:00 pts/0    00:00:00 ps -ef
[root@db380c01c168 /]#
[root@db380c01c168 /]# cat /proc/1/cgroup
```

What capabilities and seccomp mode are being used for pid 1 (the sleep 5000 process)?

```
[root@07d1eca25e39 /]# grep Cap /proc/1/status
CapInh:    0000000000000000
CapPrm:    00000000800425fb
CapEff:    00000000800425fb
CapBnd:    00000000800425fb
CapAmb:    0000000000000000
[root@07d1eca25e39 /]# grep -i seccomp /proc/1/status
Seccomp:    2
```

What SELinux label is being used on that containerized sleep process?

```
[root@db380c01c168 /]# ps -efZ
LABEL                            UID        PID  PPID  C STIME TTY          TIME CMD
```

```
system_u:system_r:container_t:s0:c478,c651 root 1  0  0 18:58 ?        00:00:00 sleep
5000

[root@db380c01c168 /]# ipcs -a
[root@db380c01c168 /]# exit
[student@workstation ~]$
```

2.4. How to change the hostname or otherwise identify which container you are running in ?

The hostname will be set to the container id by default:

```
[student@workstation ~]$  podman run -it rhel:latest /bin/bash
[root@560cf16fe847 /]# uname -n
560cf16fe847

[student@workstation ~]$  podman ps -a
CONTAINER ID   IMAGE                                    COMMAND     CREATED
STATUS             PORTS    NAMES
560cf16fe847   registry.access.redhat.com/rhel:latest   /bin/bash   2 minutes ago   Up 2
minutes ago           kind_elion
```

The existence of `/run/.containerenv` tells us that we are in a container:

```
[root@560cf16fe847 /]# ls -l /run/.containerenv
-rw-r--r--. 1 root root 0 Nov 12 13:44 /run/.containerenv
```

Env tells us what kind:

```
[root@560cf16fe847 /]# env
container=oci
```

Hostname could be set to something however:

```
[student@workstation ~]$  podman run --hostname foo --name foo -it rhel:latest /bin/bash
[root@foo /]# uname -n
foo
[root@foo /]# env | grep HOST
HOSTNAME=foo
```

2.5. What is that conmon process used for?

First, try running a containerized sleep process:

```
[student@workstation ~]$ podman run -d registry.access.redhat.com/rhel7:latest sleep 2200

[student@workstation ~]$ ps -ef | grep 2200
student     2171     2160  0 08:52 ?        00:00:00 sleep 2200
student     2182     2005  0 08:52 pts/0    00:00:00 grep --color=auto 2200
```

Notice here the `sleep 2200` process's parent PID is **2160**. What process is that?

```
[student@workstation ~]$ ps -ef | grep 2160
student     2160        1  0 08:52 ?        00:00:00 /usr/bin/conmon --api-version 1 -c
458eb55a5488ba85df6f6573133d3f2fb1d8143ff6cefa17e2278cdaf5caac58 -u
458eb55a5488ba85df6f6573133d3f2fb1d8143ff6cefa17e2278cdaf5caac58 -r /usr/bin/runc -b
/home/student/.local/share/containers/storage/overlay-
containers/458eb55a5488ba85df6f6573133d3f2fb1d8143ff6cefa17e2278cdaf5caac58/userdata -p
/run/user/1000/containers/overlay-
containers/458eb55a5488ba85df6f6573133d3f2fb1d8143ff6cefa17e2278cdaf5caac58/userdata/pidf
ile -n fervent_rhodes --exit-dir /run/user/1000/libpod/tmp/exits --full-attach -l k8s-
file:/home/student/.local/share/containers/storage/overlay-
containers/458eb55a5488ba85df6f6573133d3f2fb1d8143ff6cefa17e2278cdaf5caac58/userdata/ctr.
log --log-level warning --runtime-arg --log-format=json --runtime-arg --log --runtime-
arg=/run/user/1000/containers/overlay-
containers/458eb55a5488ba85df6f6573133d3f2fb1d8143ff6cefa17e2278cdaf5caac58/userdata/oci-
log --conmon-pidfile /run/user/1000/containers/overlay-
containers/458eb55a5488ba85df6f6573133d3f2fb1d8143ff6cefa17e2278cdaf5caac58/userdata/conm
on.pid --exit-command /usr/bin/podman --exit-command-arg --root --exit-command-arg
/home/student/.local/share/containers/storage --exit-command-arg --runroot --exit-command
-arg /run/user/1000/containers --exit-command-arg --log-level --exit-command-arg warning
--exit-command-arg --cgroup-manager --exit-command-arg cgroupfs --exit-command-arg
--tmpdir --exit-command-arg /run/user/1000/libpod/tmp --exit-command-arg --network-config
-dir --exit-command-arg  --exit-command-arg --network-backend --exit-command-arg cni
--exit-command-arg --runtime --exit-command-arg runc --exit-command-arg --storage-driver
--exit-command-arg overlay --exit-command-arg --events-backend --exit-command-arg file
--exit-command-arg container --exit-command-arg cleanup --exit-command-arg
458eb55a5488ba85df6f6573133d3f2fb1d8143ff6cefa17e2278cdaf5caac58
student     2171     2160  0 08:52 ?        00:00:00 sleep 2200
```

Another way to look at this is with pstree:

```
[student@workstation ~]$ pstree -a | less
```

```
|-conmon --api-version 1 -c
458eb55a5488ba85df6f6573133d3f2fb1d8143ff6cefa17e2278cdaf5caac58
-u458eb55a5488ba85df6f6573133d3
|   |-sleep 2200
|    -{conmon}

[student@workstation ~]$ /usr/bin/conmon --help
Usage:
  conmon [OPTION…] - conmon utility
```

`conmon` is the parent process for the `sleep 2200` process. From https://github.com/containers/conmon

> Conmon is a monitoring program and communication tool between a container manager (like Podman or CRI-O) and an OCI runtime (like runc or crun) for a single container.
>
> Upon being launched, conmon (usually) double-forks to daemonize and detach from the parent that launched it. It then launches the runtime as its child. This allows managing processes to die in the foreground, but still be able to watch over and connect to the child process (the container).
>
> While the container runs, conmon does two things:
>
> 1. Provides a socket for attaching to the container, holding open the container's standard streams and forwarding them over the socket.
> 2. Writes the contents of the container's streams to a log file (or to the systemd journal) so they can be read after the container's death.
>
> Finally, upon the containers death, conmon will record its exit time and code to be read by the managing programs.

2.6. Can you run an image built from architectures different than the container host ?

No, but you can pull them. See: https://www.redhat.com/sysadmin/specify-architecture-pulling-podman-images

```
[student@workstation ~]$ podman pull --arch=arm64 registry.access.redhat.com/ubi8
[student@workstation ~]$ podman pull registry.access.redhat.com/ubi8

[student@workstation ~]$ podman images
```

```
REPOSITORY                             TAG         IMAGE ID      CREATED        SIZE
registry.access.redhat.com/rhel7/rhel  latest      e2c37c467077  2 weeks ago    216 MB
registry.access.redhat.com/rhel7       7.9         e2c37c467077  2 weeks ago    216 MB
registry.access.redhat.com/rhel7       latest      e2c37c467077  2 weeks ago    216 MB
registry.access.redhat.com/ubi8        latest      d5c70d09f361  3 weeks ago    246 MB
<none>                                 <none>      2fd9e1478809  3 weeks ago    225 MB
```

Trying to run them will cause some errors:

```
[student@workstation ~]$ podman run 2fd9e1478809 uname -a
Linux f1b40336fff3 4.18.0-348.2.1.el8_5.x86_64 #1 SMP Mon Nov 8 13:30:15 EST 2021 x86_64
x86_64 x86_64 GNU/Linux
[student@workstation ~]$ podman run d5c70d09f361 uname -a
standard_init_linux.go:228: exec user process caused: exec format error
```

It is possible to use a qemu process to emulate different architectures if running multi-architectures is required. See https://github.com/multiarch/qemu-user-static

2.7. What are the disadvantages of running rootless containers ?

SEE https://github.com/containers/podman/blob/master/rootless.md

Some subcommands will not work, esp ones that depend on features like cgroups:

```
[student@workstation ~]$ podman pause 382
Error: pause is not supported for rootless containers

[student@workstation ~]$ podman stats 382
Error: stats is not supported in rootless mode without cgroups v2
```

Also, networking is handled differently for rootless as a non-root user has limitations on what it can do to the host's network:

```
[student@workstation ~]$ podman run -d -p 808:8080 --name myhttpd
registry.access.redhat.com/rhscl/httpd-24-rhel7
Error: error from slirp4netns while setting up port redirection: map[desc:bad request:
add_hostfwd: slirp_add_hostfwd failed]
```

SEE also https://opensource.com/article/19/2/how-does-rootless-podman-work

2.8. Understanding user namespaces

Root = not different from the host

Rootless = maps user and group IDs to appear to be running under a different ID.

```
[student@workstation ~]$ id
uid=1000(student) gid=1000(student) groups=1000(student),10(wheel) context
=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023

[student@workstation ~]$ podman run -it rhel7
[root@5367563cc886 /]# whoami
root
[root@5367563cc886 /]# id
uid=0(root) gid=0(root) groups=0(root)
```

student has uid=1000 in the global user namespace but is root with uid=0 inside a container.

From `man 7 user_namespaces`:

> In particular, a process can have a normal unprivileged user ID outside a user namespace while at the same time having a user ID of 0 inside the namespace; in other words, the process has full privileges for operations inside the user namespace, but is unprivileged for operations outside the namespace.
>
> ```
>    User and group ID mappings: uid_map and gid_map
> When  a  user  namespace  is  created,  it  starts  out  without  a  mapping  of
> user  IDs (group IDs) to the parent user namespace.  The /proc/[pid]/uid_map and
> /proc/[pid]/gid_map files (available since Linux 3.5) expose the mappings for user
> and group IDs inside  the  user namespace  for the process pid.
> ```
>
> Each line in the uid_map file specifies a 1-to-1 mapping of a range of contiguous user IDs between two user namespaces. The first two numbers specify the starting user ID in each of the two user namespaces. The third number specifies the length of the mapped range.

You can inspect this mapping file inside a container by running:

```
[root@5367563cc886 ~]$ cat /proc/self/uid_map
      (start of range)  (parent ns)      (range)
         0           1000          1
```

```
         1          100000       65536
```

So, the **root** user inside this namespace maps to the user with uid=1000 in the parent namespace (ie the “student” user).

A user with uid=1 in the child namespace would have a uid of 100000 in the parent namespace and increment up from there in the respective namespaces:

```
[root@c0ec71b5ed9e /]# cat /etc/passwd
[root@c0ec71b5ed9e /]# id 1
uid=1(bin) gid=1(bin) groups=1(bin)
[root@c0ec71b5ed9e /]# id 2
uid=2(daemon) gid=2(daemon) groups=2(daemon)
[root@c0ec71b5ed9e /]# id 3
uid=3(adm) gid=4(adm) groups=4(adm)
```

These users would map to 100000, 100001, and 100002 respectively. The child uid → parent uid mapping for the student@workstation user can thus be summarized by:

uid (child)	uid (parent)
0	1000
1	100000
2	100001
3	100002
4	100003
n	100000+(n-1)

2.9. What happens when a new user is created inside this container ?

Try it out with:

```
[root@c0ec71b5ed9e /]# useradd foo
[root@c0ec71b5ed9e /]# id foo
uid=1000(foo) gid=1000(foo) groups=1000(foo)
```

Within this container user_namespace, the **foo** user has a uid=1000. What would be that user\'s id outside the container ?

Use our mapping algorithm: uid=n → uid=100000+(n-1)

The foo user with uid=1000 inside the container would thus have a uid of 100000+(1000-1) or **100999**.

We can inspect the ownership of the files in the home directory for the upperdir (ephemeral storage) to prove it:

```
[student@workstation ~]$ podman images
REPOSITORY                       TAG         IMAGE ID      CREATED     SIZE
registry.access.redhat.com/rhel7  latest      d19b7e812477  6 days ago  218 MB

[student@workstation ~]$ podman run registry.access.redhat.com/rhel7 useradd foo

[student@workstation ~]$ podman ps -a
CONTAINER ID  IMAGE                                    COMMAND      CREATED
STATUS                        PORTS       NAMES
be70eea7d19f  registry.access.redhat.com/rhel7:latest  useradd foo  11 seconds ago
Exited (0) 10 seconds ago                 happy_herschel

[student@workstation ~]$ podman inspect be70eea7d19f | grep UpperDir
                    "UpperDir":
"/home/student/.local/share/containers/storage/overlay/2571e0f1edcf4b24bf9a54003f25ff8437
7ef027b1b034c757eb16c73af4139f/diff",

[student@workstation ~]$ ls -ld
/home/student/.local/share/containers/storage/overlay/2571e0f1edcf4b24bf9a54003f25ff84377
ef027b1b034c757eb16c73af4139f/diff/home/foo
drwx------. 2 100999 100999 62 Sep 21 10:45
/home/student/.local/share/containers/storage/overlay/2571e0f1edcf4b24bf9a54003f25ff84377
ef027b1b034c757eb16c73af4139f/diff/home/foo
```

2.10. Understanding rootless networking

Root = virtual ethernet device

Rootless = Slirp, tap device

Container networking normally uses CNI plugins to configure a bridge, but that would require root. For rootless, podman will execute `/usr/bin/slirp4netns` which provides userspace based networking. This command will create a tap device that is injected inside the new networking namespace of the containerized process.

```
[student@workstation ~]$ podman run -d registry.access.redhat.com/rhel7:latest sleep 5500
2a7d8b364faf2c8e825eb0bbadbfab7d113486514b74f78952c0ac29d19d1a95
[student@workstation ~]$ ps -ef | grep slirp
student     3568       1  0 11:00 pts/0    00:00:00 /usr/bin/slirp4netns --disable-host
-loopback --mtu=65520 --enable-sandbox --enable-seccomp --enable-ipv6 -c -e 3 -r 4
```

```
--netns-type=path /run/user/1000/netns/netns-629e7aae-b8f9-d99a-efd5-4fcbb37183ee tap0
```

Also, ping might not work depending on the version of `systemd` and `podman` installed:

```
[student@workstation ~]$ podman run -it ubi8 /bin/bash
[root@840855c79201 /]# yum install iputils
[root@ff226094dfd3 /]# ping google.com
PING google.com (172.217.1.238) 56(84) bytes of data.
^C
--- google.com ping statistics ---
57 packets transmitted, 0 received, 100% packet loss, time 57365ms
```

This has been fixed per https://bugzilla.redhat.com/show_bug.cgi?id=2037807

```
[student@workstation ~]$ rpm -q systemd
systemd-239-58.el8.x86_64

[student@workstation ~]$ rpm -q --changelog systemd
* Mon Feb 07 2022 systemd maintenance team <systemd-maint@redhat.com> - 239-57
- hash-funcs: introduce macro to create typesafe hash_ops (#2037807)
- hash-func: add destructors for key and value (#2037807)
- util: define free_func_t (#2037807)
- hash-funcs: make basic hash_ops typesafe (#2037807)
- test: add tests for destructors of hashmap or set (#2037807)
- man: document the new sysctl.d/ - prefix (#2037807)
- sysctl: if options are prefixed with "-" ignore write errors (#2037807)
- sysctl: fix segfault (#2037807)

[student@workstation ~]$ sysctl -a | grep ping
net.ipv4.ping_group_range = 0    2147483647

[student@workstation ~]$ podman run -it ubi8 /bin/bash
[root@34cb445d6819 /]# yum install iputils -y

[root@34cb445d6819 /]# ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=255 time=4.21 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=255 time=1.70 ms

[root@34cb445d6819 /]# exit
```

Thus, `ping` inside your rootless containers will be working in the latest DO180 classroom environments.

2.11. Understanding rootless storage

Root = native overlay2 which is a kernel module

Rootless = native overlay2 currently though fuse-overlayfs was used in the past

From https://www.redhat.com/sysadmin/podman-rootless-overlay

> The fuse-overlay has been great. However, it is a user-space file system, which means it needs to do almost twice as much work as the kernel. Every read/write has to be interpreted by the fuse-overlay before being passed onto the host kernel. For heavy workloads that hammer the file system, the performance of fuse-overlay suffers. You could see the fuse-overlayfs pegging out the CPU. Bottom line, we should see better performance with native overlayfs, especially for heavy read/write containers in rootless mode. For example, podman build . performance should improve significantly. Note that when writing to volumes, the fuse-overlayfs is seldom used, so performance will not be affected

The change to use the native overlay driver was first introduced in RHEL8.5 https://www.redhat.com/en/blog/whats-new-red-hat-enterprise-linux-85-container-tools See “Better Performance with Native OverlayFS“

To see the mount you’ll have to look inside the mount namespace for a running container:

```
[student@workstation ~]$ lsns -t mnt
        NS TYPE NPROCS   PID USER    COMMAND
4026531840 mnt       5  1967 student /usr/lib/systemd/systemd --user
4026532257 mnt       2  2078 student catatonit -P
4026532324 mnt       1  3582 student sleep 5500
```

From the `lsns` output the `sleep 5500` has PID **3582**. Check to confirm the **overlay** driver is used on the / mount for this containerized process:

```
[student@workstation ~]$ cat /proc/3582/mounts | grep overlay
overlay / overlay rw,context=
"system_u:object_r:container_file_t:s0:c315,c748",relatime,lowerdir=/home/student/.local/
share/containers/storage/overlay/l/FXJT5GEBB4CCTSNB6VPCWLO2OE:/home/student/.local/share/
containers/storage/overlay/l/CQ5X6VLN4VJJICYQTRKJMGREJY,upperdir=/home/student/.local/sha
re/containers/storage/overlay/6005fbe3808822c8e8e02cf1e8cfd5aadd7fd2333384ca6d72acf489b3b
6ede1/diff,workdir=/home/student/.local/share/containers/storage/overlay/6005fbe3808822c8
```

```
e8e02cf1e8cfd5aadd7fd2333384ca6d72acf489b3b6ede1/work 0 0
```

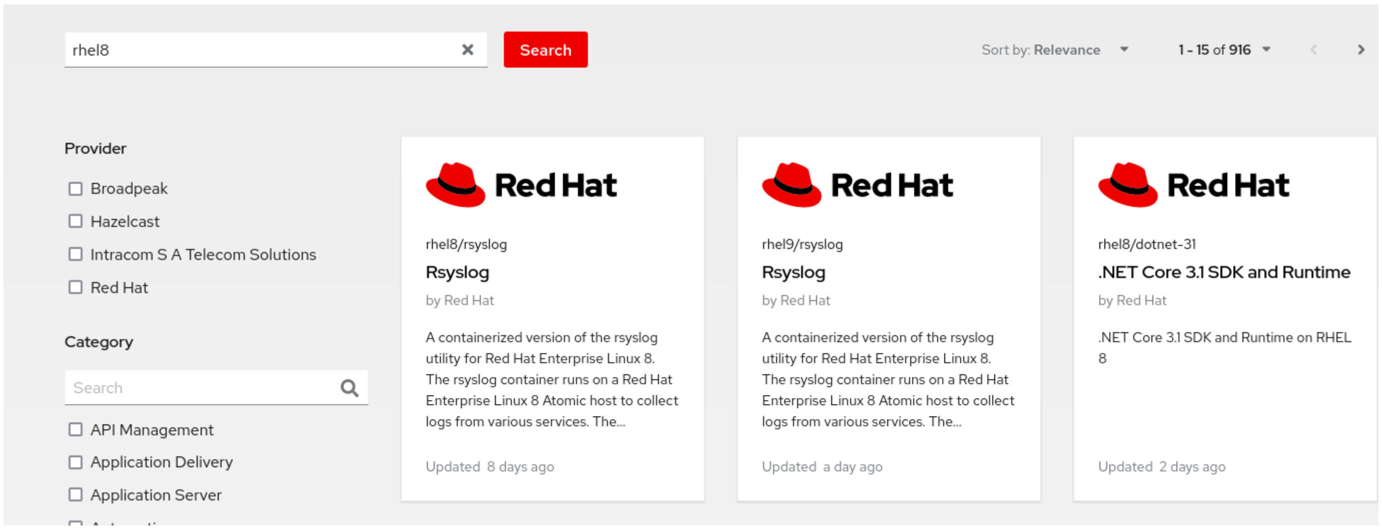
Notice also, there is no userspace fuse-overlayfs process running like in older versions of podman:

```
[student@workstation ~]$ ps -ef | grep fuse | grep -v grep
[student@workstation ~]$
```

Chapter 3. Chapter 3: Managing Containers

3.1. There is a rhel7 container image but not a rhel8 image. Where is the rhel8 container image?

Try to search and find the RHEL8 base image by searching https://catalog.redhat.com/software/containers/explore/ using the "rhel8" search string.



rhel8/rsyslog
Rsyslog
by Red Hat
A containerized version of the rsyslog utility for Red Hat Enterprise Linux 8. The rsyslog container runs on a Red Hat Enterprise Linux 8 Atomic host to collect logs from various services. The...
Updated 8 days ago

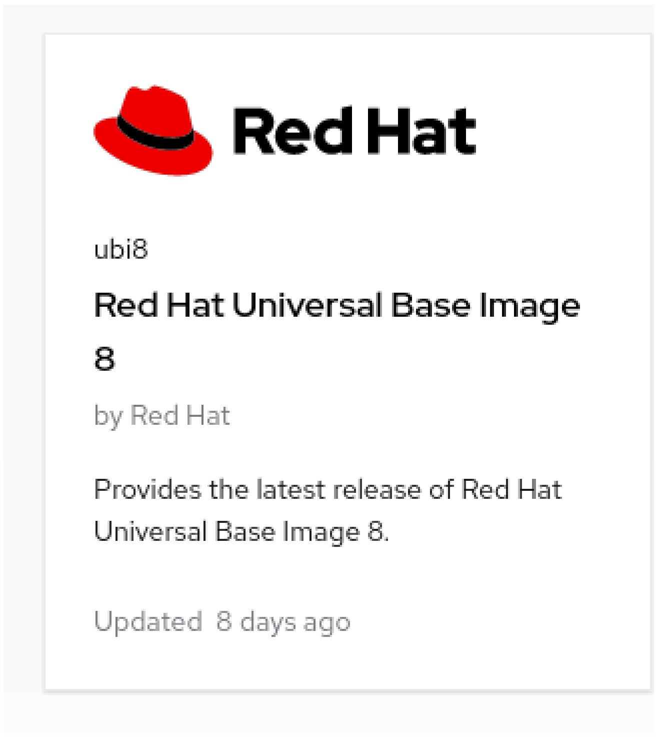
rhel9/rsyslog
Rsyslog
by Red Hat
A containerized version of the rsyslog utility for Red Hat Enterprise Linux 8. The rsyslog container runs on a Red Hat Enterprise Linux 8 Atomic host to collect logs from various services. The...
Updated a day ago

rhel8/dotnet-31
.NET Core 3.1 SDK and Runtime
by Red Hat
.NET Core 3.1 SDK and Runtime on RHEL 8
Updated 2 days ago

Figure 1. rhel8 search results

Of the search results returned, try to find a rhel8 **Scratch** image, one that includes only base RHEL8 software. You will not find one. Instead the images available include additional software or configuration The **rhel8/dotnet-31** image, for example, contains the .NET SDK and Runtime as well as standard RHEL8 packages. This may not be what you are looking for, especially, if you want a "vanilla" or **Scratch** RHEL8 image.

Now, repeat your search using **ubi8**


ubi8

Red Hat Universal Base Image 8

by Red Hat

Provides the latest release of Red Hat Universal Base Image 8.

Updated 8 days ago

Figure 2. ubi8 container image

Starting with RHEL8, Red Hat creates and distributes RHEL container images as **Universal Base Images** or **ubi** images. If you are looking for a "scratch" image that includes only the standard RHEL8 packages, then look for **ubi** instead of **rhel**.

3.2. How can you install additional packages using a ubi image?

Inspect first the `ubi.repo` file in the ubi8 image:

```
[student@workstation ~]$ podman pull registry.access.redhat.com/ubi8
Trying to pull registry.access.redhat.com/ubi8...

[student@workstation ~]$ podman run registry.access.redhat.com/ubi8 cat /etc/redhat-
release
Red Hat Enterprise Linux release 8.6 (Ootpa)

[student@workstation ~]$ podman run registry.access.redhat.com/ubi8 cat
/etc/yum.repos.d/ubi.repo
[ubi-8-baseos-rpms]
name = Red Hat Universal Base Image 8 (RPMs) - BaseOS
baseurl = https://cdn-ubi.redhat.com/content/public/ubi/dist/ubi8/8/$basearch/baseos/os
enabled = 1
gpgkey = file:///etc/pki/rpm-gpg/RPM-GPG-KEY-redhat-release
gpgcheck = 1
```

One interesting and useful feature of ubi images is the ability to install additional software via **yum** without a Red Hat subscription.

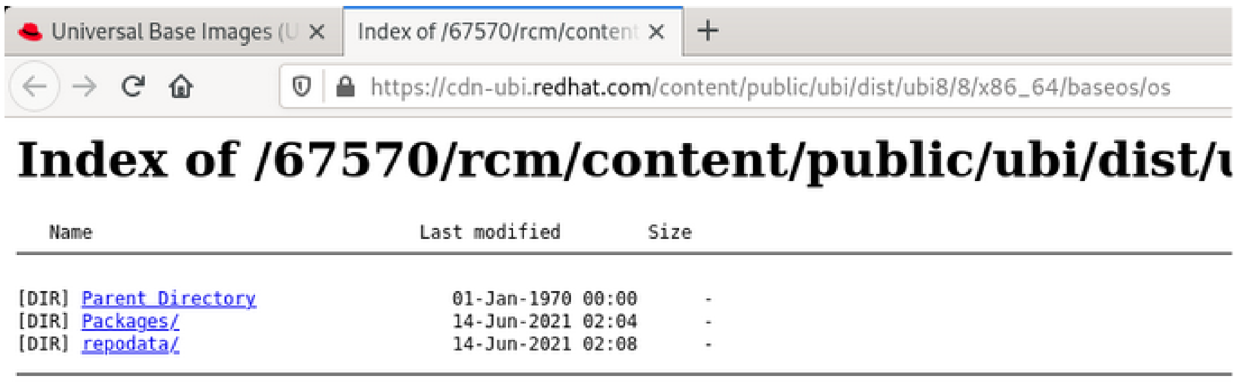
Consider running this interactive shell to install **httpd** (apache):

```
[student@workstation ~]$ podman run -it registry.access.redhat.com/ubi8 /bin/bash
[root@7a6899de957c /]# yum install httpd -y
Updating Subscription Management repositories.
Unable to read consumer identity
Subscription Manager is operating in container mode.

This system is not registered with an entitlement server. You can use subscription-
manager to register.

Red Hat Universal Base Image 8 (RPMs) - BaseOS
2.4 MB/s | 803 kB     00:00
Red Hat Universal Base Image 8 (RPMs) - AppStream
30 MB/s | 3.0 MB     00:00
Red Hat Universal Base Image 8 (RPMs) - CodeReady Builder
332 kB/s |  20 kB     00:00
Dependencies resolved.
==========================================================================================
======================================================================
 Package                          Architecture          Version
Repository                          Size
==========================================================================================
======================================================================
Installing:
 httpd                            x86_64                2.4.37-
47.module+el8.6.0+15654+427eba2e.2                 ubi-8-appstream-rpms
1.4 M
Installing dependencies:
 apr
 ...SNIP...
```

Works! Why? Open your browser to the **baseurl** given in the **ubi8.repo**. You can insert the x86_64 arch for this example: https://cdn-ubi.redhat.com/content/public/ubi/dist/ubi8/8/x86_64/baseos/os


Universal Base Images (U
Index of /67570/rcm/content
https://cdn-ubi.redhat.com/content/public/ubi/dist/ubi8/8/x86_64/baseos/os

Index of /67570/rcm/content/public/ubi/dist/u

```
   Name                          Last modified      Size
[DIR] Parent Directory             01-Jan-1970 00:00    -
[DIR] Packages/                    14-Jun-2021 02:04    -
[DIR] repodata/                    14-Jun-2021 02:08    -
```

Figure 3. ubi Package Repository

SEE also https://access.redhat.com/articles/4238681

3.3. How are the ubi images created by Red Hat? Do we publish the ubi Containerfile?

The ubi images distributed by Red Hat are not built in the same way a developer may customize one to create a layered image for their application. Instead, they are created using a tarball extracted from the root filesystem of kickstarted virtual machines.

A good explanation is given in http://crunchtools.com/ubi-build/

> The contents of most base images are created with operating system installers. While many base images utilize a Dockerfile, it probably doesn’t provide what you’re looking for. There’s an underlying chicken and egg problem because without a package manager installed and configured, the step by step instructions of how every file is laid out on disk can’t be easily expressed in a Dockerfile.

3.4. How are the names autogenerated by podman determined ?

Names will be autogenerated for containers with the form adjective_famousperson

SEE the sourcecode: https://github.com/containers/podman/blob/main/vendor/github.com/docker/docker/pkg/namesgenerator/names-generator.go

```
        left = [...]string{
                "admiring",
                "adoring",
                "affectionate",
                "agitated",
                "amazing",
                "angry",

        right = [...]string{
                // Muhammad ibn Jābir al-Ḥarrānī al-Battānī was a founding father of
astronomy. https://en.wikipedia.org/wiki/Mu%E1%B8%A5ammad_ibn_J%C4%81bir_al-
%E1%B8%A4arr%C4%81n%C4%AB_al-Batt%C4%81n%C4%AB
                "albattani",

                // Frances E. Allen, became the first female IBM Fellow in 1989. In 2006,
she became the first female recipient of the ACMs Turing Award.
https://en.wikipedia.org/wiki/Frances_E._Allen
                "allen",

...SNIP...
func main() {
begin:
        rand.Seed(time.Now().UnixNano())
        name := fmt.Sprintf("%s_%s", left[rand.Intn(len(left))], right[rand.Intn(len
(right))])
        if name == "boring_wozniak" /* Steve Wozniak is not boring */ {
                goto begin
        }
        fmt.Println(name)
}
```

3.5. What is actually running inside the httpd image from rhscl?

Run an apache daemon using the `rhscl/httpd-24-rhel7` image. Add in a name like `myhttpd` so that we can refer to this container later.

```
[student@workstation ~]$ podman run --name myhttpd -d
registry.access.redhat.com/rhscl/httpd-24-rhel7:latest

[student@workstation ~]$ podman ps --no-trunc
CONTAINER ID                                                      IMAGE
COMMAND             CREATED          STATUS             PORTS   NAMES
CONTAINER ID                                                      IMAGE
COMMAND             CREATED          STATUS             PORTS        NAMES
2c3c93d8cfe4cceadc359d98377ca24e7c87576b390a7d505c316369b8720422
registry.access.redhat.com/rhscl/httpd-24-rhel7:latest  /usr/bin/run-httpd  11 seconds
```

```
ago  Up 12 seconds ago              myhttpd
```

The `podman ps` output indicates that `/usr/bin/run-httpd` is the containerized process executed by our container runtime.

To inspect further what is running in a container, we could run another program inside the same namespaces using `podman exec`:

```
[student@workstation ~]$ podman exec -it myhttpd /bin/bash
bash-4.2$
bash-4.2$ ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
default        1       0  0 14:55 ?        00:00:00 httpd -D FOREGROUND
default       40       1  0 14:55 ?        00:00:00 /usr/bin/cat
default       41       1  0 14:55 ?        00:00:00 /usr/bin/cat
default       42       1  0 14:55 ?        00:00:00 /usr/bin/cat
default       43       1  0 14:55 ?        00:00:00 /usr/bin/cat
default       44       1  0 14:55 ?        00:00:00 httpd -D FOREGROUND
default       45       1  0 14:55 ?        00:00:00 httpd -D FOREGROUND
default       54       1  0 14:55 ?        00:00:00 httpd -D FOREGROUND
default       66       1  0 14:55 ?        00:00:00 httpd -D FOREGROUND
default       69       1  0 14:55 ?        00:00:00 httpd -D FOREGROUND
default       90       0  0 15:09 pts/0    00:00:00 /bin/bash
default       99      90  0 15:10 pts/0    00:00:00 ps -ef
```

What happened to the `run-httpd` processs? pid=1 from the output above indicates `httpd -D FOREGROUND` is running. No `run-httpd` seems to be running here.

Inspect the contents of `/usr/bin/run-httpd`. Use `podman exec` like before:

```
[student@workstation ~]$ podman exec -it myhttpd /bin/bash
bash-4.2$ cat /usr/bin/run-httpd
...SNIP…
process_extending_files ${HTTPD_APP_ROOT}/src/httpd-pre-init/
${HTTPD_CONTAINER_SCRIPTS_PATH}/pre-init/

exec httpd -D FOREGROUND $@
```

`usr/bin/run-httpd` is a script (a wrapper). The last command executed is `exec httpd -D FOREGROUND`. We did see this process and it was pid=1 in this namespace.

The `exec` is important here. `exec` is a shell (bash) built-in command. From `man exec`:

> exec [-cl] [-a name] [command [arguments]] If command is specified, it

> replaces the shell. No new process is created. The arguments become the arguments to command.

In summary, the container runtime executes `/usr/bin/run-httpd` in isolation. This command then executes `httpd -D FOREGROUND` with the `exec` built-in. This `httpd` replaces the `run-httpd` process becoming pid=1 in this namespace.

3.6. How to use a systemd.unit file so that a container is started on system boot?

Assuming you have a container named `myhttpd` defined previously, use `podman generate systemd`

```
[student@workstation ~]$ podman ps
CONTAINER ID  IMAGE                                                COMMAND
CREATED         STATUS              PORTS       NAMES
2c3c93d8cfe4  registry.access.redhat.com/rhscl/httpd-24-rhel7:latest  /usr/bin/run-
http...  28 minutes ago  Up 28 minutes ago              myhttpd

[student@workstation ~]$ podman generate systemd -n myhttpd
# container-myhttpd.service
# autogenerated by Podman 4.0.2
# Wed Sep 21 14:27:51 EDT 2022

[Unit]
Description=Podman container-myhttpd.service
Documentation=man:podman-generate-systemd(1)
Wants=network-online.target
After=network-online.target
RequiresMountsFor=/run/user/1000/containers

[Service]
Environment=PODMAN_SYSTEMD_UNIT=%n
Restart=on-failure
TimeoutStopSec=70
ExecStart=/usr/bin/podman start myhttpd
ExecStop=/usr/bin/podman stop -t 10 myhttpd
ExecStopPost=/usr/bin/podman stop -t 10 myhttpd
PIDFile=/run/user/1000/containers/overlay-
containers/2c3c93d8cfe4cceadc359d98377ca24e7c87576b390a7d505c316369b8720422/userdata/conm
on.pid
Type=forking

[Install]
WantedBy=default.target
```

This systemd.unit file can used to start and stop the myhttpd container via `systemctl` commands:

```
[student@workstation ~]$ podman stop myhttpd
[student@workstation ~]$ mkdir -p ~/.config/systemd/user
[student@workstation ~]$ podman generate systemd -n myhttpd >
~/.config/systemd/user/myhttpd.service
[student@workstation ~]$ systemctl --user daemon-reload
[student@workstation ~]$ systemctl --user enable myhttpd.service
[student@workstation ~]$ systemctl --user start myhttpd
[student@workstation ~]$ podman ps
CONTAINER ID  IMAGE                                                 COMMAND
CREATED        STATUS              PORTS       NAMES
2c3c93d8cfe4  registry.access.redhat.com/rhscl/httpd-24-rhel7:latest  /usr/bin/run-
http...  30 minutes ago  Up 20 seconds ago              myhttpd

[student@workstation ~]$ systemctl --user status myhttpd
 myhttpd.service - Podman container-myhttpd.service
   Loaded: loaded (/home/student/.config/systemd/user/myhttpd.service; enabled; vendor
preset: enabled)
   Active: active (running) since Wed 2022-09-21 14:28:50 EDT; 39s ago
     Docs: man:podman-generate-systemd(1)
  Process: 6294 ExecStart=/usr/bin/podman start myhttpd (code=exited, status=0/SUCCESS)

[student@workstation ~]$ systemctl --user stop myhttpd
[student@workstation ~]$ podman ps
CONTAINER ID  IMAGE       COMMAND     CREATED     STATUS      PORTS       NAMES
```

To have this container automatically start on boot you can leverage `loginctl enable-linger`. You will need to run this with elevated privileges:

```
[student@workstation ~]$ sudo loginctl enable-linger student
[student@workstation ~]$ sudo reboot
Connection to 172.25.252.1 closed by remote host.
Connection to 172.25.252.1 closed.
(after a reboot)
[student@workstation ~]$ podman ps
CONTAINER ID  IMAGE                                                 COMMAND
CREATED        STATUS                  PORTS       NAMES
2c3c93d8cfe4  registry.access.redhat.com/rhscl/httpd-24-rhel7:latest  /usr/bin/run-
http...  34 minutes ago  Up About a minute ago              myhttpd
[student@workstation ~]$ systemctl --user status myhttpd
 myhttpd.service - Podman container-myhttpd.service
   Loaded: loaded (/home/student/.config/systemd/user/myhttpd.service; enabled; vendor
preset: enabled)
   Active: active (running) since Wed 2022-09-21 14:31:41 EDT; 1min 37s ago
     Docs: man:podman-generate-systemd(1)
```

```
  Process: 1349 ExecStart=/usr/bin/podman start myhttpd (code=exited, status=0/SUCCESS)
 Main PID: 1603 (conmon)
```

SEE also https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html-single/building_running_and_managing_containers/index#proc_enabling-systemd-services_assembly_porting-containers-to-systemd-using-podman

3.7. How to extract metadata from `podman inspect` ?

Containers and images have a large number of properties defined as part of the OCI specification. These details are typically not displayed when running commands like `podman ps`. `podman inspect` can be used to dump all of the properties or keys (metadata) of a container.

```
[student@workstation ~]$ podman inspect --help
Display the configuration of object denoted by ID

Description:
  Displays the low-level information on an object identified by name or ID.
  For more inspection options, see:

  [student@workstation ~]$ podman ps
  CONTAINER ID  IMAGE                                                    COMMAND
CREATED         STATUS              PORTS       NAMES
  2c3c93d8cfe4  registry.access.redhat.com/rhscl/httpd-24-rhel7:latest  /usr/bin/run-
http...  47 minutes ago  Up 14 minutes ago              myhttpd
  [student@workstation ~]$ podman inspect myhttpd | head
  [
       {
            "Id": "2c3c93d8cfe4cceadc359d98377ca24e7c87576b390a7d505c316369b8720422",
            "Created": "2022-09-21T13:58:47.711790266-04:00",
            "Path": "container-entrypoint",
            "Args": [
                 "/usr/bin/run-httpd"
            ]
```

`podman inspect` will return a JSON object with all of the `"key": value` pairs included in a container's metadata.

There are different techniques for parsing this information. Some methods use straightforward shell-based approaches piping this data into tools like `grep`, `awk`, and/or `sed`

Suppose you wanted to identify the global PID for the containerized process in the `myhttpd` container. Some combination of these tools could be used like the following:

```
[student@workstation ~]$ podman inspect myhttpd | grep -i pid
               "Pid": 1614,
               "ConmonPid": 1603,
          "ConmonPidFile": "/run/user/1000/containers/overlay-
containers/2c3c93d8cfe4cceadc359d98377ca24e7c87576b390a7d505c316369b8720422/userdata/conm
on.pid",
          "PidFile": "/run/user/1000/containers/overlay-
containers/2c3c93d8cfe4cceadc359d98377ca24e7c87576b390a7d505c316369b8720422/userdata/pidf
ile",
               "PidMode": "private",
               "PidsLimit": 0,
[student@workstation ~]$ podman inspect myhttpd | grep -i \"Pid\":
               "Pid": 1614,
[student@workstation ~]$ podman inspect myhttpd | grep -i \"Pid\": | awk -F: '{print $1}'
               "Pid"
[student@workstation ~]$ podman inspect myhttpd | grep -i \"Pid\": | awk -F: '{print $2}'
 1614,
[student@workstation ~]$ podman inspect myhttpd | grep -i \"Pid\": | awk -F: '{print $2}'
| sed 's/,//'
 1614
 [student@workstation ~]$ ps -ef | grep 1614
 101000      1614    1603  0 14:31 ?        00:00:00 httpd -D FOREGROUND
 101000      1724    1614  0 14:31 ?        00:00:00 /usr/bin/cat
 101000      1725    1614  0 14:31 ?        00:00:00 /usr/bin/cat
 101000      1726    1614  0 14:31 ?        00:00:00 /usr/bin/cat
 101000      1727    1614  0 14:31 ?        00:00:00 /usr/bin/cat
 101000      1728    1614  0 14:31 ?        00:00:00 httpd -D FOREGROUND
 101000      1734    1614  0 14:31 ?        00:00:00 httpd -D FOREGROUND
 101000      1736    1614  0 14:31 ?        00:00:00 httpd -D FOREGROUND
 101000      1752    1614  0 14:31 ?        00:00:00 httpd -D FOREGROUND
 101000      1753    1614  0 14:31 ?        00:00:00 httpd -D FOREGROUND
 student     2355    2023  0 14:54 pts/0    00:00:00 grep --color=auto 1614
```

Because the default output from `podman inspect` is a JSON object, tools that are built to parse JSON might be less problematic. Consider the `jq` utility:

```
[student@workstation ~]$ podman inspect myhttpd | jq
(lots of output)
[student@workstation ~]$ podman inspect myhttpd | jq | head
```

To filter the output stream using `jq` **filters** can be applied based on the object and the values you are interested in.

It is best to understand JSON a bit first.

JSON = JavaScript Object Notation

It is way to organize/structure data in a readable format. This is primarily done through a list of "key" and "value" pairs in the format:

```
"Key1": "value1",
"Key2": "value2",
...etc...
```

The **value** can be any of the following types:

- **String**: Several plain text characters which usually form a word enclosed in quotes " "
- **Boolean**: Value will be either true or false.
- **Number**: An integer
- **Object**: An associative array of key/value pairs ... a "dictionary" { }
- **Array**: An associative array of values ... a "list" [ ]

Here are some examples from the `podman inspect` output:

```
[student@workstation ~]$ podman inspect myhttpd | jq | less
[
  {
    "Id": "2c3c93d8cfe4cceadc359d98377ca24e7c87576b390a7d505c316369b8720422",
    "Created": "2022-09-21T13:58:47.711790266-04:00", <---------------------  STRING
    "Path": "container-entrypoint",
    "Args": [       <-------------------------------------------------------  ARRAY
      "/usr/bin/run-httpd"
    ],
    "State": {         <---------------------------------------------------- OBJECT
      "OciVersion": "1.0.2-dev",
      "Status": "running",
      "Running": true,    <------------------------------------------------- BOOLEAN
      "Paused": false,
      "Restarting": false,
      "OOMKilled": false,
      "Dead": false,
      "Pid": 1614,  <------------------------------------------------------- NUMBER
      "ConmonPid": 1603,
...SNIP...
```

Understanding these different value types is critical to parsing JSON.

With `jq` a filter can be defined which will parse the JSON input and return out the values for the keys identified in the filter.

As you see in the example above, some keys are nested inside an OBJECT. Others are part of an ARRAY. Take note of this while you study some examples:

```
[student@workstation ~]$ podman inspect myhttpd | jq
[
  {
    "Id": "2c3c93d8cfe4cceadc359d98377ca24e7c87576b390a7d505c316369b8720422",
    "Created": "2022-09-21T13:58:47.711790266-04:00",
    "Path": "container-entrypoint",
    "Args": [
      "/usr/bin/run-httpd"
    ],
    "State": {
      "OciVersion": "1.0.2-dev",
      "Status": "running",
      "Running": true,
      "Paused": false,
      "Restarting": false,
      "OOMKilled": false,
      "Dead": false,
      "Pid": 1614,
...SNIP...
```

To parse this, notice that first line is **[** indicating that everything you see here is stored inside an ARRAY. To filter any key therefore we must include **.[]** in the filter so that all the keys in this ARRAY are included.

```
[student@workstation ~]$ podman inspect myhttpd | jq ".[]"
{
  "Id": "2c3c93d8cfe4cceadc359d98377ca24e7c87576b390a7d505c316369b8720422",
  "Created": "2022-09-21T13:58:47.711790266-04:00",
  "Path": "container-entrypoint",
  "Args": [
    "/usr/bin/run-httpd"
  ],
  "State": {
    "OciVersion": "1.0.2-dev",
    "Status": "running",
    "Running": true,
    "Paused": false,
    "Restarting": false,
    "OOMKilled": false,
    "Dead": false,
    "Pid": 1614,
...SNIP...
```

The first three keys listed in this object have STRING values. To return the value for one of them we can expand our filter:

```
[student@workstation ~]$ podman inspect myhttpd | jq ".[] .Id"
"2c3c93d8cfe4cceadc359d98377ca24e7c87576b390a7d505c316369b8720422"
[student@workstation ~]$ podman inspect myhttpd | jq ".[] .Created"
"2022-09-21T13:58:47.711790266-04:00"
[student@workstation ~]$ podman inspect myhttpd | jq ".[] .Path"
"container-entrypoint"
```

For an ARRAY we can identify the value by its *position* or *index* in the ARRAY. The first item in the list has an *index* of **0**. It increments by 1 from there (ie 0, 1, 2, 3, etc)

```
[student@workstation ~]$ podman inspect myhttpd | jq ".[] .Args"
[
  "/usr/bin/run-httpd"
]
[student@workstation ~]$ podman inspect myhttpd | jq ".[] .Args[0]"
"/usr/bin/run-httpd"
[student@workstation ~]$ podman inspect myhttpd | jq ".[] .Args[1]"

[student@workstation ~]$
```

It should make sense why there is a *null* value returned when `.[] .Args[1]` is used. There is only 1 value in the `.Args[]` ARRAY. Remember the first index is 0. The second is 1.

To filter OBJECTS like `State {}` above, try these:

```
[student@workstation ~]$ podman inspect myhttpd | jq ".[] .State"
{
  "OciVersion": "1.0.2-dev",
  "Status": "running",
  "Running": true,
  "Paused": false,
  "Restarting": false,
  "OOMKilled": false,
  "Dead": false,
  "Pid": 1614,
  "ConmonPid": 1603,
  "ExitCode": 0,
  "Error": "",
  "StartedAt": "2022-09-21T14:31:41.526823412-04:00",
    "FinishedAt": "2022-09-21T14:30:06.842203892-04:00",
    "Health": {
      "Status": "",
      "FailingStreak": 0,
      "Log": null
    },
    "CgroupPath": "/user.slice/user-1000.slice/user@1000.service",
    "CheckpointedAt": "0001-01-01T00:00:00Z",
    "RestoredAt": "0001-01-01T00:00:00Z"
  }
[student@workstation ~]$ podman inspect myhttpd | jq ".[] .State .Pid"
1614
[student@workstation ~]$ podman inspect myhttpd | jq ".[] .State .Running"
true
```

With `jq` there is much more possible. Consider `man jq` and other `jq` tutorials for more advanced JSON parsing.

3.8. How to use `podman inspect --format` with a Go template?

A **Go template** is a pattern used in golang that can be useful for filtering input text or generating specific output.

The **Go template** is an expression passed to `podman inspect <container> --format`. From https://pkg.go.dev/text/template

> Execution of the template walks the structure and sets the cursor, represented by a period '.' and called "dot", to the value at the current location in the structure as execution proceeds.
>
> The input text for a template is UTF-8-encoded text in any format. "Actions" are data evaluations or control structures delimited by "{{" and "}}"; all text outside actions is copied to the output unchanged.

Just like with `jq`, it is critical to understand the different types of values in the input data object: STRING, BOOLEAN, NUMBER, OBJECT, ARRAY

Here are some patterns that will output a few different string values:

```
[student@workstation ~]$ podman inspect myhttpd --format '{{.}}'
{0xc000b43b00}
[student@workstation ~]$ podman inspect myhttpd --format '{{.Id}}'
```

```
2c3c93d8cfe4cceadc359d98377ca24e7c87576b390a7d505c316369b8720422
[student@workstation ~]$ podman inspect myhttpd --format '{{.Created}}'
2022-09-21 13:58:47.711790266 -0400 EDT
[student@workstation ~]$ podman inspect myhttpd --format '{{.Path}}'
container-entrypoint
```

For values nested inside of an OBJECT:

```
[student@workstation ~]$ podman inspect myhttpd --format '{{.State}}'
{1.0.2-dev running true false false false false 1614 1603 0  2022-09-21
14:31:41.526823412 -0400 EDT 2022-09-21 14:30:06.842203892 -0400 EDT { 0 []} false
/user.slice/user-1000.slice/user@1000.service 0001-01-01 00:00:00 +0000 UTC 0001-01-01
00:00:00 +0000 UTC    false}
[student@workstation ~]$ podman inspect myhttpd --format '{{.State.Pid}}'
1614
[student@workstation ~]$ podman inspect myhttpd --format '{{.State.Running}}'
true
```

For ARRAYs use the keyword **index** like:

```
[student@workstation ~]$ podman inspect myhttpd --format '{{.Args}}'
[/usr/bin/run-httpd]

[student@workstation ~]$ podman inspect myhttpd --format '{{index .Args 0}}'
/usr/bin/run-httpd

[student@workstation ~]$ podman inspect myhttpd --format '{{index .Args 1}}'
ERRO[0000] Printing inspect output: template: all inspect:1:13: executing "all inspect"
at <index .Args 1>: error calling index: reflect: slice index out of range
```

More complex processing of ARRAYs can occur using the **range** reserved word. Consider:

```
[student@workstation ~]$ podman inspect myhttpd --format '{{.BoundingCaps}}'
[CAP_CHOWN CAP_DAC_OVERRIDE CAP_FOWNER CAP_FSETID CAP_KILL CAP_NET_BIND_SERVICE
CAP_NET_RAW CAP_SETFCAP CAP_SETGID CAP_SETPCAP CAP_SETUID CAP_SYS_CHROOT]
[student@workstation ~]$

[student@workstation ~]$ podman inspect myhttpd --format '{{index .BoundingCaps 0}}'
CAP_CHOWN

[student@workstation ~]$ podman inspect myhttpd --format '{{index .BoundingCaps 1}}'
CAP_DAC_OVERRIDE

[student@workstation ~]$ podman inspect myhttpd --format '{{index .BoundingCaps 2}}'
```

```
CAP_FOWNER

[student@workstation ~]$ podman inspect myhttpd --format '{{range .BoundingCaps}}cap:
{{end}}'
cap: cap: cap: cap: cap: cap: cap: cap: cap: cap: cap: cap:

[student@workstation ~]$ podman inspect myhttpd --format '{{range
.BoundingCaps}}cap:{{.}} {{end}}'
cap:CAP_CHOWN cap:CAP_DAC_OVERRIDE cap:CAP_FOWNER cap:CAP_FSETID cap:CAP_KILL
cap:CAP_NET_BIND_SERVICE cap:CAP_NET_RAW cap:CAP_SETFCAP cap:CAP_SETGID cap:CAP_SETPCAP
cap:CAP_SETUID cap:CAP_SYS_CHROOT

[student@workstation ~]$ podman inspect myhttpd --format '{{range
.BoundingCaps}}cap:{{.}}\n{{end}}'
cap:CAP_CHOWN
cap:CAP_DAC_OVERRIDE
cap:CAP_FOWNER
cap:CAP_FSETID
cap:CAP_KILL
cap:CAP_NET_BIND_SERVICE
cap:CAP_NET_RAW
cap:CAP_SETFCAP
cap:CAP_SETGID
cap:CAP_SETPCAP
cap:CAP_SETUID
cap:CAP_SYS_CHROOT
```

Conditionals are possible within a **Go template** as well. Suppose we are worried if a container includes the capability **CAP_KILL** :

```
[student@workstation ~]$ podman inspect myhttpd --format '{{range .BoundingCaps}} {{.}}
{{end}}'
 CAP_CHOWN  CAP_DAC_OVERRIDE  CAP_FOWNER  CAP_FSETID  CAP_KILL  CAP_NET_BIND_SERVICE
CAP_NET_RAW  CAP_SETFCAP  CAP_SETGID  CAP_SETPCAP  CAP_SETUID  CAP_SYS_CHROOT

[student@workstation ~]$ podman inspect myhttpd --format '{{range .BoundingCaps}}{{if eq
. "CAP_KILL"}}eek this can kill{{end}} ok {{end}}'
 ok  ok  ok  ok eek this can kill ok  ok  ok  ok  ok  ok  ok  ok
```

You can also use this with `podman ps` to help create custom table that are useful to inspect information.

For example,

```
[student@workstation ~]$ podman ps -a --format json
[student@workstation ~]$ podman ps -a --format='{{.Names}} {{.State}} {{.Image}}'
```

```
[student@workstation ~]$ podman ps -a --format='table {{.Names}} {{.State}} {{.Image}}'
NAMES              STATE                  IMAGE
fervent_rhodes     Exited (0) 7 hours ago registry.access.redhat.com/rhel7:latest
happy_herschel     Exited (0) 5 hours ago registry.access.redhat.com/rhel7:latest
magical_pare       Exited (0) 4 hours ago registry.access.redhat.com/rhel7:latest
pensive_edison     Exited (0) 4 hours ago registry.access.redhat.com/ubi8:latest
practical_swirles  Exited (0) 4 hours ago registry.access.redhat.com/ubi8:latest
elegant_chebyshev  Exited (0) 4 hours ago registry.access.redhat.com/ubi8:latest
sad_haslett        Exited (0) 2 hours ago registry.access.redhat.com/rhscl/httpd-24-
rhel7:latest
myhttpd            Up 2 hours ago         registry.access.redhat.com/rhscl/httpd-24-
rhel7:latest
```

Parsing metadata like these examples will also apply to later Chapters that involve Red Hat Openshift Container Platform. SEE https://www.openshift.com/blog/customizing-oc-output-with-go-templates for ideas on how to apply these parsing techniques to `oc`

3.9. What does `podman pause` do to a running container ?

`podman pause` uses the cgroup "freezer" to freeze (halt) a task without stopping it or without the task knowing.

This is NOT supported with rootless due to a limit in the freezer cgroup v1.

The container and its processes are paused while the image is committed. This minimizes the likelihood of data corruption when creating the new image. (man podman-commit)

From https://www.kernel.org/doc/Documentation/cgroup-v1/freezer-subsystem.txt

> The cgroup freezer will also be useful for checkpointing running groups of tasks. The cgroup freezer is hierarchical. Freezing a cgroup freezes all tasks belonging to the cgroup and all its descendant cgroups

Try the following as the root user:

```
[student@workstation ~]$ sudo su -
[root@workstation ~]# podman run -d --name mywebapp -p 8888:8080
registry.access.redhat.com/rhscl/httpd-24-rhel7
3bc5ea1dbfc1694bd8a1cca6e57c54638e2b1c1f818c966b047abc95fc9d443d

[root@workstation ~]# curl -I http://localhost:8888
HTTP/1.1 403 Forbidden
Date: Wed, 21 Sep 2022 20:33:09 GMT
```

```
Server: Apache/2.4.34 (Red Hat) OpenSSL/1.0.2k-fips
Last-Modified: Wed, 23 Mar 2022 14:51:57 GMT
ETag: "f91-5dae3defb5d40"
Accept-Ranges: bytes
Content-Length: 3985
Content-Type: text/html; charset=UTF-8
```

Now pause the **mywebapp** container with `podman pause`

```
[root@workstation ~]# podman pause mywebapp
3bc5ea1dbfc1694bd8a1cca6e57c54638e2b1c1f818c966b047abc95fc9d443d
[root@workstation ~]# cat /sys/fs/cgroup/freezer/machine.slice/libpod-
3bc5ea*/freezer.state
FROZEN

[root@workstation ~]# curl -I http://localhost:8888
(hangs)
^C
```

This container is not responding as we would expect from being "paused". Additionally, the state is marked as **Paused**

```
[root@workstation ~]# podman ps -a
CONTAINER ID  IMAGE                                               COMMAND
CREATED        STATUS      PORTS                   NAMES
3bc5ea1dbfc1  registry.access.redhat.com/rhscl/httpd-24-rhel7:latest  /usr/bin/run-
http...  3 minutes ago  Paused      0.0.0.0:8888->8080/tcp  mywebapp
```

This process will not consume CPU time and will be in "D" state:

```
[root@workstation ~]# podman inspect 3bc5ea1dbfc1 --format '{{.State.Pid}}'
4625

[root@workstation ~]# ps -ef | grep 4625
devops      4625    4614  0 16:32 ?        00:00:00 httpd -D FOREGROUND
devops      4685    4625  0 16:32 ?        00:00:00 /usr/bin/cat
devops      4686    4625  0 16:32 ?        00:00:00 /usr/bin/cat
devops      4687    4625  0 16:32 ?        00:00:00 /usr/bin/cat
devops      4688    4625  0 16:32 ?        00:00:00 /usr/bin/cat
devops      4689    4625  0 16:32 ?        00:00:00 httpd -D FOREGROUND
devops      4695    4625  0 16:32 ?        00:00:00 httpd -D FOREGROUND
devops      4703    4625  0 16:32 ?        00:00:00 httpd -D FOREGROUND
devops      4716    4625  0 16:32 ?        00:00:00 httpd -D FOREGROUND
devops      4719    4625  0 16:32 ?        00:00:00 httpd -D FOREGROUND
```

```
root         4910    3843  0 16:37 pts/0    00:00:00 grep --color=auto 4625

[root@workstation ~]# ps -aux | grep 4625
devops       4625  0.0  0.2 373552 17420 ?        Ds   16:32   0:00 httpd -D FOREGROUND
```

Now run `podman unpause`

```
[root@workstation ~]# podman unpause mywebapp
3bc5ea1dbfc1694bd8a1cca6e57c54638e2b1c1f818c966b047abc95fc9d443d
```

It will be available and responsive again:

```
[root@workstation ~]# curl -I http://localhost:8888
HTTP/1.1 403 Forbidden
Date: Wed, 21 Sep 2022 20:40:33 GMT
Server: Apache/2.4.34 (Red Hat) OpenSSL/1.0.2k-fips
Last-Modified: Wed, 23 Mar 2022 14:51:57 GMT
ETag: "f91-5dae3defb5d40"
Accept-Ranges: bytes
Content-Length: 3985
Content-Type: text/html; charset=UTF-8

[root@workstation ~]# cat /sys/fs/cgroup/freezer/machine.slice/libpod-
3bc5ea*/freezer.state
THAWED

[root@workstation ~]# ps -aux | grep 4625
devops       4625  0.0  0.2 373552 17420 ?        Ss   16:32   0:00 httpd -D FOREGROUND

[root@workstation ~]# exit
logout
[student@workstation ~]$
```

ifndef::env-github[:icons: font] ifdef::env-github[] :status: :outfilesuffix: .adoc :caution-caption: :fire: :important-caption: :exclamation: :note-caption: :paperclip: :tip-caption: :bulb: :warning-caption: :warning: endif::[] :imagesdir: ./images/

3.10. How does overlay work?

The overlay kernel driver provides a POSIX filesystem by creating the union of several different directory paths together.

```
[student@workstation ~]$ modinfo overlay
```

```
filename:       /lib/modules/4.18.0-372.9.1.el8.x86_64/kernel/fs/overlayfs/overlay.ko.xz
alias:          fs-overlay
license:        GPL
description:    Overlay filesystem
...SNIP...
```

This filesystem is used as default mount provided to containers. The data written to this filesystem should be considered *ephemeral*, meaning short-lived or fleeting. Once a container is removed from a container host (via `podman rm`) the data written by that containerized process is also removed. This storage driver is used by both podman and crio (ie openshift).

For systems using podman, the underlying storage driver used for *ephemeral* operations is configured in `/etc/containers/storage.conf`

```
[student@workstation ~]$ cat /etc/containers/storage.conf  | grep -v ^#
[storage]

driver = "overlay"
runroot = "/run/containers/storage"
graphroot = "/var/lib/containers/storage"
```

Although not explicitly given above, the "overlay" driver used by the latest RHEL and RHEL CoreOS systems is overlay version 2. From https://docs.docker.com/storage/storagedriver/overlayfs-driver/#how-the-overlay2-driver-works

> The overlay2 driver natively supports up to 128 lower OverlayFS layers. The original overlay driver only worked with 2 layers, extra layers in overlay relied on hard-linked directories. This created excessive use of inodes (known limitation)

To really appreciate how the overlay driver works start by becoming the `root` user and create a few directories and files:

```
[student@workstation ~]$ sudo su -
Last login: Wed Sep 21 16:27:30 EDT 2022 on pts/0
[root@workstation ~]# mkdir -p /data/lower{1..3} /data/upper /data/work /data/merged
[root@workstation ~]# touch /data/lower1/file{1..3}  /data/lower2/file{4..6}
/data/lower3/file{7..9}  /data/upper/file10

[root@workstation ~]# tree /data
/data
├─── lower1
│    ├─── file1
│     ├──── file2
│     └──── file3
├──── lower2
│     ├──── file4
│     ├──── file5
│     └──── file6
├──── lower3
│     ├──── file7
│     ├──── file8
│     └──── file9
├──── merged
├──── upper
│     └──── file10
└──── work

6 directories, 10 files
```

Next, mount lower1, lower2, lower3, and upper together in a union mount using the overlay driver.

```
[root@workstation ~]# mount -t overlay overlay -o
lowerdir=/data/lower1:/data/lower2:/data/lower3,upperdir=/data/upper,workdir=/data/work
/data/merged
[root@workstation ~]#

[root@workstation ~]# df /data/merged
Filesystem     1K-blocks    Used Available Use% Mounted on
overlay         10371052 7556520   2814532  73% /data/merged

[root@workstation ~]# mount | grep merged
overlay on /data/merged type overlay (rw,relatime,seclabel,lowerdir
=/data/lower1:/data/lower2:/data/lower3,upperdir=/data/upper,workdir=/data/work)
```

Notice how the overlay driver expects several unique mount options including **lowerdir**, **upperdir**, and **workdir**. The names of our directories could have been any string we wanted, but just to keep this activity straightforward, the names of the directories closely match the mount options that use them.

At this point, your shell has access to a POSIX compliant filesystem mounted on `/data/merged`. Try to make a few changes to the files in `/data/merged`:

```
[root@workstation ~]# ls -l /data/merged
total 0
-rw-r--r--. 1 root root 0 Sep 22 10:23 file1
-rw-r--r--. 1 root root 0 Sep 22 10:23 file10
-rw-r--r--. 1 root root 0 Sep 22 10:23 file2
-rw-r--r--. 1 root root 0 Sep 22 10:23 file3
```

```
-rw-r--r--. 1 root root 0 Sep 22 10:23 file4
-rw-r--r--. 1 root root 0 Sep 22 10:23 file5
-rw-r--r--. 1 root root 0 Sep 22 10:23 file6
-rw-r--r--. 1 root root 0 Sep 22 10:23 file7
-rw-r--r--. 1 root root 0 Sep 22 10:23 file8
-rw-r--r--. 1 root root 0 Sep 22 10:23 file9

# modify an existing file
[root@workstation ~]# echo hello > /data/merged/file9

# remove an existing file
[root@workstation ~]# rm -f /data/merged/file1

# create a new file
[root@workstation ~]# touch /data/merged/file11

# create a new directory
[root@workstation ~]# mkdir /data/merged/dir1
[root@workstation ~]#
```

All of these operations seem to work, lets confirm in `/data/merged`:

```
[root@workstation ~]# ls -l /data/merged
total 4
drwxr-xr-x. 2 root root 6 Sep 22 10:40 dir1
-rw-r--r--. 1 root root 0 Sep 22 10:23 file10
-rw-r--r--. 1 root root 0 Sep 22 10:40 file11
-rw-r--r--. 1 root root 0 Sep 22 10:23 file2
-rw-r--r--. 1 root root 0 Sep 22 10:23 file3
-rw-r--r--. 1 root root 0 Sep 22 10:23 file4
-rw-r--r--. 1 root root 0 Sep 22 10:23 file5
-rw-r--r--. 1 root root 0 Sep 22 10:23 file6
-rw-r--r--. 1 root root 0 Sep 22 10:23 file7
-rw-r--r--. 1 root root 0 Sep 22 10:23 file8
-rw-r--r--. 1 root root 6 Sep 22 10:40 file9
```

`dir1` and `file11` are now present. `file1` is gone. `file9` is now a **6 byte** file. All changes that we should expect from POSIX filesystem.

Where there any changes to the lowerdir directories: `/data/lower1`, `/data/lower2`, or `/data/lower3`

```
[root@workstation ~]# ls -l /data/lower{1..3}
/data/lower1:
total 0
-rw-r--r--. 1 root root 0 Sep 22 10:23 file1
```

```
-rw-r--r--. 1 root root 0 Sep 22 10:23 file2
-rw-r--r--. 1 root root 0 Sep 22 10:23 file3

/data/lower2:
total 0
-rw-r--r--. 1 root root 0 Sep 22 10:23 file4
-rw-r--r--. 1 root root 0 Sep 22 10:23 file5
-rw-r--r--. 1 root root 0 Sep 22 10:23 file6

/data/lower3:
total 0
-rw-r--r--. 1 root root 0 Sep 22 10:23 file7
-rw-r--r--. 1 root root 0 Sep 22 10:23 file8
-rw-r--r--. 1 root root 0 Sep 22 10:23 file9
```

No changes at all. `file1` is still there. `file9` is still a **0 byte** file. `dir1` and `file11` are not present at all.

So where are all these changes being tracked ? The *upperdir* !!

```
[root@workstation ~]# ls -l /data/upper
total 4
drwxr-xr-x. 2 root root    6 Sep 22 10:40 dir1
c---------. 2 root root 0, 0 Sep 22 10:40 file1
-rw-r--r--. 1 root root    0 Sep 22 10:23 file10
-rw-r--r--. 1 root root    0 Sep 22 10:40 file11
-rw-r--r--. 1 root root    6 Sep 22 10:40 file9
```

Very interesting collection of files in `data/upper`.

- First, notice the presence of `dir1` and `file11`. Those were not in `/data/upper` at the start of this exercise.
- Next, `file1` is a new but it is not a "regular" file. Notice `c---------.`. This is an indication that it is a special file called a "character" file. Character files are used throughout Linux for various purposes. Here it is being used by the overlay driver to **mask** a file so that it is not presented on the mountpoint.
- Last, notice that `file9` is a **6 byte** file matching what we see in `/data/merged`. `file9` exists in both `/data/lower2` and `data/upper`. The overlay driver presents the data from only `data/upper` however. So, the order of the lowerdir in relation to the upperdir is important.

When running a container with `podman run` the libpod and the container runtime tools mount using this same overlay driver. The `lowerdir` are based on the image in the `podman run` command. The `upperdir` is a new directory where the changes can be tracked similar to `/data/upper` in this activity.

Try creating a new file within a the ephemeral storage of a container:

```
[root@workstation ~]# exit
logout
[student@workstation ~]$ podman run --name overlaytest1 ubi8 touch /var/tmp/amazingfile
[student@workstation ~]$
[student@workstation ~]$ podman inspect overlaytest1 --format
'{{.GraphDriver.Data.UpperDir}}'
/home/student/.local/share/containers/storage/overlay/0498243e4703e41120240e8077d017ac65d
cabc0f4b7ef2514ab02841df3650c/diff

[student@workstation ~]$ ls -l
/home/student/.local/share/containers/storage/overlay/0498243e4703e41120240e8077d017ac65d
cabc0f4b7ef2514ab02841df3650c/diff/var/tmp
total 0
-rw-r--r--. 1 student student 0 Sep 22 11:01 amazingfile
```

Try removing a file:

```
[student@workstation ~]$ podman run --name overlaytest2 ubi8 rm -f /etc/motd
[student@workstation ~]$ podman inspect overlaytest2 --format
'{{.GraphDriver.Data.UpperDir}}'
/home/student/.local/share/containers/storage/overlay/b663a3bf74e20a2af37f0278a124c554ac0
3dc5859f9ed68fcb15c0105711652/diff

[student@workstation ~]$ ls -l
/home/student/.local/share/containers/storage/overlay/b663a3bf74e20a2af37f0278a124c554ac0
3dc5859f9ed68fcb15c0105711652/diff/etc/motd
c---------. 2 student student 0, 0 Sep 22 11:06
/home/student/.local/share/containers/storage/overlay/b663a3bf74e20a2af37f0278a124c554ac0
3dc5859f9ed68fcb15c0105711652/diff/etc/motd
```

It should make sense why we find a character device here in the upperdir for this particular container.

To clean up from this exercise simple `umount /data/merged`

```
[student@workstation ~]$ sudo umount /data/merged

[student@workstation ~]$ mount | grep merged
[student@workstation ~]$
```

3.11. Has overlay2 always been the ephemeral storage driver used on container hosts?

No. Previously `devicemapper` was the default storage driver in early RHEL7 and RHEL Atomic Host. This changed to use `overlay` starting with this errata https://access.redhat.com/errata/RHBA-2018:1064

See https://bugzilla.redhat.com/show_bug.cgi?id=1475625

3.12. How to apply selinux labels when mounting persistent storage?

The selinux label on a volume can be made persistent on a host using an `semanage fcontext` comamnd as discussed in the DO180 course material:

```
[student@workstation ~]$ sudo semanage fcontext -a -t container_file_t
'/home/student/mydata(/.*)?'

[student@workstation ~]$ sudo restorecon -Rv /home/student/mydata/
/home/student/mydata not reset as customized by admin to
unconfined_u:object_r:container_file_t:s0
```

Alternatively use the :Z option like:

```
[student@workstation ~]$ mkdir mydata1

[student@workstation ~]$  podman unshare chown 1001 mydata1

[student@workstation ~]$  podman run -d -v /home/student/mydata1:/var/www/html:Z
registry.redhat.io/rhscl/httpd-24-rhel7
9c9d8752085e22ad32407b1f655a1a49e2427ceeae46ca6455665c6b0412db96

[student@workstation ~]$ podman exec 9c ls -ldZ /var/www/html
drwxrwxr-x. default root system_u:object_r:container_file_t:s0:c1002,c1014 /var/www/html

[student@workstation ~]$ podman exec 9c ls -ldZ /var/www/html
drwxrwxr-x. default root system_u:object_r:container_file_t:s0:c1002,c1014 /var/www/html

[student@workstation ~]$ podman exec 9c touch /var/www/html/index.html

[student@workstation ~]$ ls -lZ mydata1
total 0
-rw-r--r--. 1 101000 student system_u:object_r:container_file_t:s0:c1002,c1014 0 Sep 14
11:03 index.html
```

It is also possible to use one of the persistent volumes managed with `podman volume`:

```
[student@workstation ~]$ podman run -d -v myvol:/var/www/html rhscl/httpd-24-rhel7
6e5f9dc5474c4853e0bf01e508ba2471bbc190830a1c51b06204dd436846f07e

[student@workstation ~]$ podman volume list
DRIVER   VOLUME NAME
local    myvol

[student@workstation ~]$ podman volume inspect myvol
[
     {
          "Name": "myvol",
          "Driver": "local",
          "Mountpoint":
"/home/student/.local/share/containers/storage/volumes/myvol/_data",
          "CreatedAt": "2022-09-22T11:31:06.486128153-04:00",
          "Labels": {},
          "Scope": "local",
          "Options": {},
          "UID": 1001,
          "MountCount": 0,
          "NeedsCopyUp": true
     }
]

[student@workstation ~]$ podman inspect 6e --format '{{.Mounts}}'
[{volume myvol /home/student/.local/share/containers/storage/volumes/myvol/_data
/var/www/html local  [noexec nosuid nodev rbind] true rprivate}]

[student@workstation ~]$ podman exec -it 6e /bin/bash
bash-4.2$ df -h /var/www/html
Filesystem      Size  Used Avail Use% Mounted on
/dev/vda3       9.9G  7.3G  2.7G  73% /var/www/html


bash-4.2$ touch /var/www/html/index.html
bash-4.2$ exit
exit

[student@workstation ~]$ ls
/home/student/.local/share/containers/storage/volumes/myvol/_data -lZ
total 0
-rw-r--r--. 1 101000 student system_u:object_r:container_file_t:s0 0 Sep 22 11:32
index.html
```

3.13. How can you remove host volumes using podman system prune ?

Consider creating a container using a new volume called **myvol**:

```
[student@workstation ~]$ podman run -d -v myvol:/var/www/html rhscl/httpd-24-rhel7
6e5f9dc5474c4853e0bf01e508ba2471bbc190830a1c51b06204dd436846f07e

[student@workstation ~]$ podman volume list
DRIVER   VOLUME NAME
local    myvol
```

In this case, `podman` created a directory on the host for us.

```
[student@workstation ~]$ podman inspect 6e --format '{{.Mounts}}'
[{volume myvol /home/student/.local/share/containers/storage/volumes/myvol/_data
/var/www/html local  [noexec nosuid nodev rbind] true rprivate}]
```

Use this storage as you would any persistent volume:

```
[student@workstation ~]$ podman exec -it 6e /bin/bash
bash-4.2$ df -h
Filesystem      Size  Used Avail Use% Mounted on
fuse-overlayfs  9.9G  6.3G  3.7G  64% /
tmpfs            64M     0   64M   0% /dev
tmpfs           580M  100K  580M   1% /etc/hosts
shm              63M     0   63M   0% /dev/shm
/dev/vda3       9.9G  6.3G  3.7G  64% /var/www/html
tmpfs           2.9G     0  2.9G   0% /sys/fs/cgroup
devtmpfs        2.8G     0  2.8G   0% /dev/tty
tmpfs           2.9G     0  2.9G   0% /proc/acpi
tmpfs           2.9G     0  2.9G   0% /proc/scsi
tmpfs           2.9G     0  2.9G   0% /sys/firmware
tmpfs           2.9G     0  2.9G   0% /sys/fs/selinux
bash-4.2$ touch /var/www/html/index.html
bash-4.2$ exit
exit
```

Now try to delete the container using `podman system prune` with the `--volumes` argument:

```
[student@workstation ~]$ podman ps
CONTAINER ID  IMAGE                                                    COMMAND
```

```
CREATED        STATUS            PORTS  NAMES
6e5f9dc5474c  registry.access.redhat.com/rhscl/httpd-24-rhel7:latest  /usr/bin/run-
http...  2 minutes ago  Up 2 minutes ago         youthful_mahavira
[student@workstation ~]$ podman stop 6e
6e5f9dc5474c4853e0bf01e508ba2471bbc190830a1c51b06204dd436846f07e
[student@workstation ~]$ podman system prune --volumes

WARNING! This will remove:
        - all stopped containers
        - all volumes not used by at least one container
        - all stopped pods
        - all dangling images
        - all build cache
Are you sure you want to continue? [y/N] y
Deleted Pods
Deleted Containers
6e5f9dc5474c4853e0bf01e508ba2471bbc190830a1c51b06204dd436846f07e
Deleted Volumes
myvol
```

3.14. Will `podman system prune --volumes` delete ALL volumes?

No. **volumes** created outside of `podman` will not be removed/pruned. Try it:

```
[student@workstation ~]$ podman volume list
[student@workstation ~]$

[student@workstation ~]$ mkdir mydir
[student@workstation ~]$ podman unshare chown 1001:1001 mydir
[student@workstation ~]$ podman unshare chcon -t container_file_t mydir
[student@workstation ~]$ podman run -d -v /home/student/mydir:/var/www/html rhscl/httpd-
24-rhel7
5e0175ff9761696c20887cd450a4500552e3ed877cfd62fc4f43fb4c0d39c03f

[student@workstation ~]$ podman volume list
[student@workstation ~]$
---

No volume listed.  This volume was not created with `podman volume`.


[source,bash]
```

[student@workstation ~]$ podman inspect 5e --format '{{.Mounts}}'

[student@workstation ~]$ ls -ldZ /home/student/mydir drwxrwxr-x. 2 101000 101000 unconfined_u:object_r:container_file_t:s0 6 Sep 22 16:07 /home/student/mydir

```
Now, stop the container and try to `prune` it:


[source,bash]
```

[student@workstation ~]$ podman stop 5e 5e0175ff9761696c20887cd450a4500552e3ed877cfd62fc4f43fb4c0d39c03f [student@workstation ~]$ podman system prune --volumes

WARNING! This will remove: - all stopped containers - all volumes not used by at least one container - all stopped pods - all dangling images - all build cache Are you sure you want to continue? [y/N] y Deleted Pods Deleted Containers 5e0175ff9761696c20887cd450a4500552e3ed877cfd62fc4f43fb4c0d39c03f Deleted Volumes

```
Yet, the host directory `/home/student/mydir` is still present:

[source,bash]
```

[student@workstation ~]$ ls -ldZ /home/student/mydir drwxrwxr-x. 2 101000 101000 unconfined_u:object_r:container_file_t:s0 6 Sep 22 16:07 /home/student/mydir

```
There are also other times `podman system prune` will not remove host directories.  SEE
https://bugzilla.redhat.com/show_bug.cgi?id=1811570#c15 :

[quote]
----
podman system prune should NOT be removing buildah containers/images.
----

:pygments-style: tango
:source-highlighter: pygments
:toc:
:toclevels: 7
:sectnums:
:sectnumlevels: 6
:numbered:
:chapter-label:
:icons: font
:icons: font
```

```
:imagesdir: ./images/


=== What is used to configure networking for containers run as root?


[source,bash]
```

[student@workstation ~]$ podman info | grep networkBackend networkBackend: cni

```
The Container Networking Interface (CNI)  https://github.com/containernetworking/cni aims
to standardize the network interface for containers in cloud native environments, such as
Kubernetes and Red Hat OpenShift Container Platform
consists of a specification and libraries for writing plugins to configure network
interfaces in Linux containers

What other plugins are available ?
https://github.com/containernetworking/plugins

Some CNI network plugins, maintained by the containernetworking team.  There are others
maintained by different teams.  Consider

There is an ovs cni plugin among others:
Ovs-cni plugin: https://github.com/kubevirt/ovs-cni/blob/master/docs/cni-plugin.md

[source,bash]
```

[student@workstation ~]$ podman network inspect podman [{ "name": "podman", "id": "2f259bab93aaaaa2542ba43ef33eb990d0999ee1b9924b557b7be53c0b7a1bb9", "driver": "bridge", "network_interface": "cni-podman0", "created": "2022-09-22T12:50:38.202589559-04:00", "subnets": [{ "subnet": "10.88.0.0/16", "gateway": "10.88.0.1" }], "ipv6_enabled": false, "internal": false, "dns_enabled": false, "ipam_options": { "driver": "host-local" } }]

[student@workstation ~]$ sudo su - [root@workstation ~]# bridge link show 4: virbr0-nic: <BROADCAST,MULTICAST> mtu 1500 master virbr0 state disabled priority 32 cost 100

[root@workstation ~]# ip addr show (note missing interface for cni-podman0)

```
Now, let□s run a container and check the networking on the host again:

[source,bash]
```

[root@workstation ~]# podman run -d rhscl/httpd-24-rhel7 44d328fb7ca9f951e5e165a4eff4860789c446fc35c249844259b9f3320e67fa [root@workstation ~]# bridge link show 4: virbr0-nic: <BROADCAST,MULTICAST> mtu 1500 master virbr0 state disabled priority 32 cost 100 6: vethf643eab8@virbr0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 master cni-podman0 state forwarding priority 32 cost 2

[root@workstation ~]# ip addr show cni-podman0 5: cni-podman0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000 link/ether 32:f4:04:21:cd:64 brd ff:ff:ff:ff:ff:ff inet 10.88.0.1/16 brd 10.88.255.255 scope global cni-podman0 valid_lft forever preferred_lft forever inet6 fe80::30f4:4ff:fe21:cd64/64 scope link valid_lft forever preferred_lft forever

```
=== How does ip allocation work with cni ipam host-local ?

[NOTE]
======
Running containers rootless will rely on `/usr/bin/slirp4netns` instead of the CNI bridge
`cni-podman0` used by the root user.
======

First, become *root* and verify the network settings.

[source,bash]
```

[student@workstation ~]$ sudo su - Last login: Thu Sep 22 10:23:11 EDT 2022 on pts/0

[root@workstation podman]# podman network ls NETWORK ID NAME DRIVER 2f259bab93aa podman bridge

[root@workstation podman]# podman network inspect podman [{ "name": "podman", "id": "2f259bab93aaaaa2542ba43ef33eb990d0999ee1b9924b557b7be53c0b7a1bb9", "driver": "bridge", "network_interface": "cni-podman0", "created": "2022-09-22T11:46:57.484659338-04:00", "subnets": [{ "subnet": "10.88.0.0/16", "gateway": "10.88.0.1" }], "ipv6_enabled": false, "internal": false, "dns_enabled": false, "ipam_options": { "driver": "host-local" } }]

```
*IPAM* = IP Address Management

Notice the configuration above indicates that the ipam_options will use the "host-local"
driver.  The directory used for this driver type is `/var/lib/cni/networks/podman`

The IP address assigned to a particular container is listed as a flat text file while its
contents matches its networking namespace (or containerID).

The last_reserved_ip.0 is a "helper" file indicating the last assigned IP address.

Consider this example:
```

```
[source,bash]
```

[root@workstation ~]# podman run -d --name mynetworktest rhscl/httpd-24-rhel7 11fd588f624b29b70c7fe2a97ef08f7934ca72623e3aed8d6fea7936be125917

[root@workstation ~]# cd /var/lib/cni/networks/podman/

[root@workstation podman]# ls 10.88.0.2 last_reserved_ip.0 lock

[root@workstation podman]# cat 10.88.0.2 11fd588f624b29b70c7fe2a97ef08f7934ca72623e3aed8d6fea7936be125917

```
Check to see that the string in this file matches the Container ID for the
`mynetworktest` container:

[source,bash]
```

[root@workstation podman]# podman ps CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES 11fd588f624b registry.access.redhat.com/rhscl/httpd-24-rhel7:latest /usr/bin/run-http… 35 seconds ago Up 35 seconds ago mynetworktest

```
Try launching a few more containers:

[source,bash]
```

[root@workstation podman]# podman run -d rhscl/httpd-24-rhel7 9b4cbeca79b04671135ac604635095c94d8f2a51f99d2c2676628d0c12291a55

[root@workstation podman]# podman run -d rhscl/httpd-24-rhel7 cfba6c4fa574e5f1efc0e8b3c215c43efdc838690003fc5bf77eb983425e3897

[root@workstation podman]# podman run -d rhscl/httpd-24-rhel7 11ed2853870d72440eef540657a5016b6efc313d16aba056421857628e957415

```
Now, check the contents in `/var/lib/cni/networks/podman`:


[source,bash]
```

[root@workstation podman]# ls -l /var/lib/cni/networks/podman total 20 -rw-r—r--. 1 root root 70 Sep 22 11:38 10.88.0.2 -rw-r—r--. 1 root root 70 Sep 22 11:41 10.88.0.3 -rw-r—r--. 1 root root 70 Sep 22 11:41 10.88.0.4 -rw-r—r--. 1 root root 70 Sep 22 11:41 10.88.0.5 -rw-r—r--. 1 root root 9 Sep 22 11:41

last_reserved_ip.0 -rwxr-x---. 1 root root 0 Sep 22 11:38 lock

```
Cleanup with:

[source,bash]
```

[root@workstation ~]# podman rm -a -f 32e2db9f03612fb8ad76f7dff01501eac4051bba7cabf71a209356bab460f04e 505dca427887333d55fc7c7ac80054afc7f39e014ab84801146906b17e8b2ae7 60e4ebf255e2817da98efda381e69622a3391f5226541565690bfb2f02c64f03 76b23247927546f45cee71946af865ca404293eabeb491024987518bb5be3306 a0d9ad7000cb12d3b1447e8d506fab0c5a0df8c819bda89b9cf8736483e41539 [root@workstation ~]# exit logout

```
:pygments-style: tango
:source-highlighter: pygments
:toc:
:toclevels: 7
:sectnums:
:sectnumlevels: 6
:numbered:
:chapter-label:
:icons: font
:icons: font
:imagesdir: ./images/

==  Chapter 4: Managing Container Images

:sectnums:
:pygments-style: tango
:source-highlighter: pygments
:toc:
:toclevels: 7
:sectnums:
:sectnumlevels: 6
:numbered:
:chapter-label:
:icons: font
:icons: font
:imagesdir: ./images/

=== Where can I find the actual OCI specification?

OCI spec: https://opencontainers.org/
```

```
Runtime-spec https://github.com/opencontainers/runtime-spec

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

=== How can you block access to certain registries system-wide?

This should now be possible because of
https://bugzilla.redhat.com/show_bug.cgi?id=1787667

..but you can t really - https://bugzilla.redhat.com/show_bug.cgi?id=1811098

Using the newer podman v2 configuration format:

[source,bash]
```

[student@workstation ~]$ sudo vi /etc/containers/registries.conf unqualified-search-registries = ["registry.access.redhat.com", "registry.redhat.io", "quay.io"]

prefix = "quay.io" location = "quay.io" insecure = false blocked = true

```
Now, lets give this a test by pulling from `quay.io`.  Both of these should fail based on
the `location` and `blocked = true` in the `registries.conf`

[source,bash]
```

[student@workstation ~]$ podman pull quay.io/redhattraining/httpd-parent:latest Trying to pull quay.io/redhattraining/httpd-parent:latest... Error: initializing source docker://quay.io/redhattraining/httpd-parent:latest: registry quay.io is blocked in /etc/containers/registries.conf or /home/student/.config/containers/registries.conf.d

[student@workstation ~]$ podman pull quay.io/ajblum/mytest:latest Trying to pull quay.io/ajblum/mytest:latest... Error: initializing source docker://quay.io/ajblum/mytest:latest: registry quay.io is blocked in /etc/containers/registries.conf or /home/student/.config/containers/registries.conf.d

```
It s also possible to block registries from a particular namespace.  For this, use the
location for matching instead of the prefix:

[source,bash]
```

unqualified-search-registries = ["registry.access.redhat.com", "registry.redhat.io", "quay.io"]

location = "quay.io/ajblum" insecure = false blocked = true

[student@workstation ~]$ podman pull quay.io/ajblum/mytest:latest Trying to pull quay.io/ajblum/mytest:latest... Error: initializing source docker://quay.io/ajblum/mytest:latest: registry quay.io is blocked in /etc/containers/registries.conf or /home/student/.config/containers/registries.conf.d

```
Fails as expected, let's try a different repo from `quay.io`

[source,bash]
```

[student@workstation ~]$ podman pull quay.io/redhattraining/httpd-parent:latest Trying to pull quay.io/redhattraining/httpd-parent:latest... Getting image source signatures

```
Works.  So, only a specific namespace can be blocked.

Ok - Looks good, right ?  Now, we create a local `registries.conf` as a non-root user
(even a blank file will work):

[source,bash]
```

[student@workstation ~]$ mkdir -p ~/.config/containers/ [student@workstation ~]$ touch ~/.config/containers/registries.conf

[student@workstation ~]$ podman pull quay.io/ajblum/mytest:latest Trying to pull quay.io/ajblum/mytest:latest... Getting image source signatures

```
Worked … eek !  But, really nothing can stop a motivated user to work around this global
config.

From BZ https://bugzilla.redhat.com/show_bug.cgi?id=1811098

[quote]
----
a local user could still pull an image via curl or by pointing the tools to another path.
----

A systems administrator would have to resort to other (more drastic) techniques like
egress firewall rules or other network proxy/appliance to intercept these user requests.

=== How to communicate with the registry API directly?

Commands like `podman search` build RESTful requests sent to various registries.
Consider searching `quay.io` for a repository called *mytest*:

[source,bash]
```

[student@workstation ~]$ podman search quay.io/mytest INDEX NAME DESCRIPTION STARS OFFICIAL AUTOMATED quay.io quay.io/ihoukai/mytest 0 quay.io quay.io/little_arhat/mytest test of homu/quay integration 0 quay.io quay.io/guenael/mytest 0 quay.io quay.io/ajblum/mytest 0

```
It is possible to build a request to a registries' API without using podman or skopeo at
all.  Consider this example using `curl`:

[source,bash]
```

[student@workstation ~]$ curl https://quay.io/v2/ajblum/mytest/tags/list {"name":"ajblum/mytest","tags":["1.0","latest","2.0","3.0","4.0","5.0"]}

```
Additional curl troubleshooting: https://access.redhat.com/articles/3560571

Different registries use different API's.

Docker Registry API docs: https://docs.docker.com/registry/spec/api/
For Quay: https://docs.quay.io/api/swagger/

Consider the documentation from quay.io and build a request URL to list the public
repositories available through the `ajblum` namespace.

Navigate your browser to https://docs.quay.io/api/swagger/ and scroll down to the *List,
create and manage repositories* seen here:

image::api-repos.png[]

Expand the operation GET `/api/v1/repository` and set public=true and namespace=ajblum as
seen here:

image::api-get-repo.png[]

Now, scroll down a bit further and click on *Try it out!*

image::api-try.png[]

Copy the *Request URL* and give it a try using `curl`

[source,bash]
```

[student@workstation ~]$ curl -L "https://quay.io/api/v1/repository?public=true&namespace=ajblum" {"repositories": [{"namespace": "ajblum", "name": "mytest", "description": "", "is_public": true, "kind": "image",...SNIP...

[student@workstation ~]$ curl -Ls "https://quay.io/api/v1/repository?public=true&namespace=ajblum"

| jq '.repositories[].name' "mytest" "myapp" "httpd-systemd" "versioned-hello" "myubi" "foo" "helloworld" "debezium-connector-postgres" "rhel7-attr" "hello-openshift" "myubitest" "mysigtest" "do180"

```
=== How to communicate with the registry API using `skopeo`?

[source,bash]
```

[student@workstation ~]# skopeo inspect docker://quay.io/ajblum/mytest { "Name": "quay.io/ajblum/mytest", "Tag": "latest", "Digest": "sha256:6cd0217844a2d778786dcc8c9c948aecc6ca1a36f8f16e5e4bbd4151f7ba5a61", "RepoTags": ["1.0", "latest", "2.0", "3.0", "4.0", "5.0" …SNIP…

```
=== Other ways to copy images locally other than podman pull?

`skopeo` is a powerful tool to use when working with container image registries.

Consider how `skopeo` can be used to pull an image into a local directory for direct
inspection:

[source,bash]
```

[student@workstation ~]$ mkdir /tmp/mytest [student@workstation ~]$ skopeo copy docker://quay.io/ajblum/mytest:1.0 dir:/tmp/mytest [student@workstation mytest]$ cd /tmp/mytest/ [student@workstation mytest]$ ls [student@workstation mytest]$ cat manifest.json [student@workstation mytest]$ cat manifest.json | json_reformat [student@workstation mytest]$ file a38d7adc1eb9f56b95435dfb6a51d26e225ef0181c0c71f9f8434c79e98aa59f [student@workstation mytest]$ tar xvzf a38d7adc1eb9f56b95435dfb6a51d26e225ef0181c0c71f9f8434c79e98aa59f

```
Other ways you might use skopeo to copy images:

[source,bash]
```

[student@workstation ~]$ skopeo copy docker://quay.io/ajblum/mytest:1.0 containers-storage:quay.io/ajblum/mytest:1.0

[student@workstation ~]$ skopeo copy docker://quay.io/ajblum/mytest:1.0 oci-archive:/tmp/mytest/mytest.tar

[student@workstation ~]$ podman load -i /tmp/mytest/mytest.tar Getting image source signatures

[student@workstation ~]$ podman images REPOSITORY TAG IMAGE ID CREATED SIZE <none> <none> a6a3e178a6bc 4 months ago 215MB

```
This is very useful for disconnected runtime enviornments without direct internet access.
Images can be copied and later transferred to those disconnected environments.

What about registry.redhat.io ?

This works:

[source,bash]
```

[student@workstation ~]# skopeo inspect docker://registry.access.redhat.com/rhel { "Name": "registry.access.redhat.com/rhel", "Digest": "sha256:2d215868e282e68998adece762d374ea49d66266d9dee67776eddc80a3d8e168", "RepoTags": ["7.3-74",

```
But, not this:

[source,bash]
```

[student@workstation ~]# skopeo inspect docker://registry.redhat.io/rhel FATA[0000] unable to retrieve auth token: invalid username/password

```
Skopeo will use the same authentication used by podman.  So, use `podman login` or
`skopeo login` first.


=== How can you pull an image using its digest?

Suppose you are interested in specific images from
registry.access.redhat.com/rhscl/httpd-24-rhel7

[source,bash]
```

[student@workstation storage]$ skopeo inspect docker://registry.access.redhat.com/rhscl/httpd-24-rhel7:latest | head -10 { "Name": "registry.access.redhat.com/rhscl/httpd-24-rhel7", "Digest": "sha256:02152fd99c0bcfae06af21301ad92ffa122a46e537465d2b6f064f56e5c0685f", "RepoTags": ["2.4-170.1638430400-source", "2.4-170", "2.4-172", "2.4-146-source", "2.4-136.1614612498", "2.4-170.1638430400",

```
Compare to

[source,bash]
```

[student@workstation ~]$ skopeo inspect docker://registry.access.redhat.com/rhscl/httpd-24-rhel7:2.4-172 | head -10 { "Name": "registry.access.redhat.com/rhscl/httpd-24-rhel7", "Digest": "sha256:ed835f1a45efb7dfd62894274692f494ddbf83d1072019ecafc040574cce5886", "RepoTags": ["2.4-170.1638430400-source", "2.4-170", "2.4-172", "2.4-146-source", "2.4-136.1614612498", "2.4-170.1638430400",

```
We could make a local copy using the tag 2.4-172 but lets try using this digest:

[source,bash]
-----
[student@workstation storage]$ podman pull registry.access.redhat.com/rhscl/httpd-24-
rhel7@sha256:ed835f1a45efb7dfd62894274692f494ddbf83d1072019ecafc040574cce5886
...SNIP...
fcea1b0658e6a351aec4119d8c9ee2adb725e151536b98aa8c13d4c6b8e8647b

[student@workstation storage]$ podman images
registry.access.redhat.com/rhscl/httpd-24-rhel7  <none>      fcea1b0658e6  2 months ago
329 MB
```

We see later how we can assign a local tag to this image if we want.

3.15. How to login to a registry using a service account?

Container images from some of the supported Red Hat registries require authentication. See https://access.redhat.com/RegistryAuthentication

One such registry is `registry.redhat.io`. Try pulling an image from this registry:

```
[student@workstation ~]$ podman pull registry.redhat.io/rhel7
Trying to pull registry.redhat.io/rhel7...Failed
error pulling image "registry.redhat.io/rhel7": unable to pull registry.redhat.io/rhel7:
unable to pull image: Error determining manifest MIME type for
docker://registry.redhat.io/rhel7:latest: unable to retrieve auth token: invalid
username/password
```

As discussed in the DO180 course, `podman login` is one way to authenticate with `registry.redhat.io`

```
[student@workstation ~]$ podman login --help
Login to a container registry

Description:
  Login to a container registry on a specified server.

Usage:
  podman login [options] [REGISTRY]

Examples:
  podman login quay.io
  podman login --username ... --password ... quay.io
  podman login --authfile dir/auth.json quay.io
```

Using your customer portal (ie access.redhat.com) **username** and **password** however is not recommended for security reasons. Do a little test and see.

Make sure to use your customer portal login and password so that authentication is successful:

```
[student@workstation ~]$ podman login -u your-username registry.redhat.io
Password:
Login Succeeded!

[student@workstation ~]$ cat ${XDG_RUNTIME_DIR}/containers/auth.json
{
    "auths": {
        "registry.redhat.io": {
            "auth": "some00000random1111string"
        }
    }
}
```

It may seem like the **auth** string "some00000random1111string" is encrypted and safe from malicious attack. But, in fact, this string is *encoded* not encrypted.

Meaning, it is easy to take an encoded string and decode it back to the original string. Try it using your auth string (the string given in the code snip is an example and not valid):

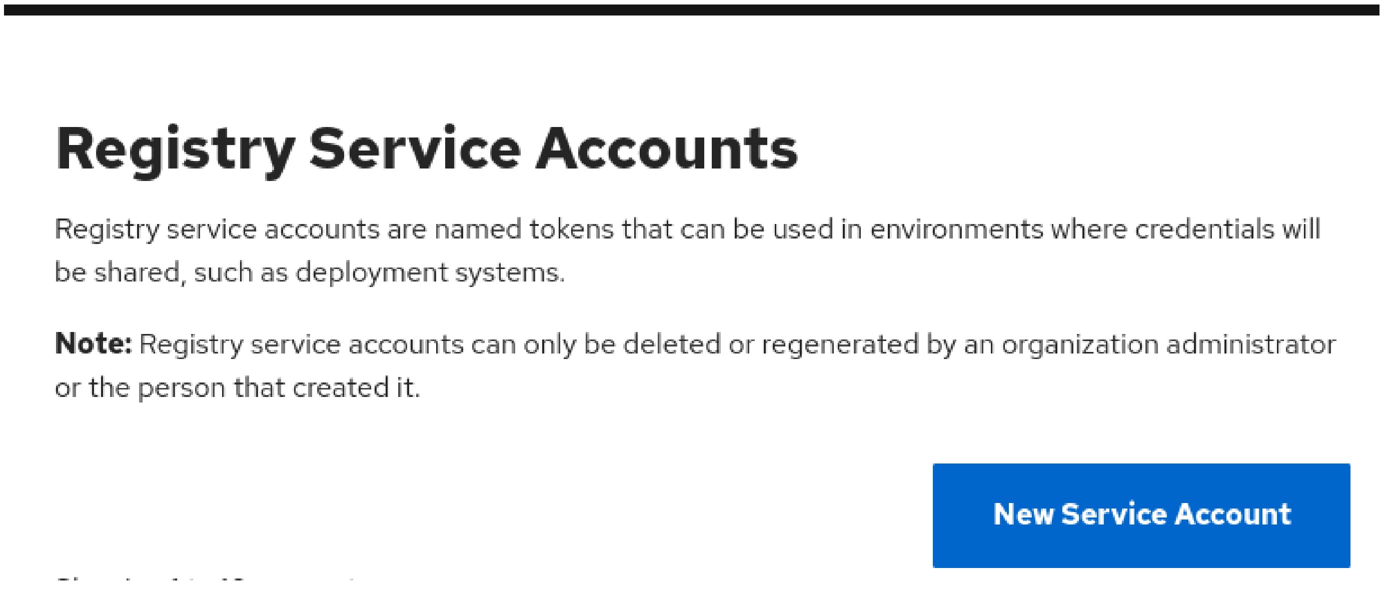
```
[student@workstation ~]$ echo -n "some00000random1111string" | base64 -d
(username and password in plain text)
```

Not good. Let’s logout and clean up our shell history:

```
[student@workstation ~]$ podman logout registry.redhat.io
Removed login credentials for registry.redhat.io
[student@workstation ~]$ history -c
[student@workstation ~]$
```

Instead, login into registry.redhat.io using a **Registry Service Account** Navigate your browser to https://access.redhat.com/terms-based-registry/#/

Log into the customer portal via your browser if you have not already.

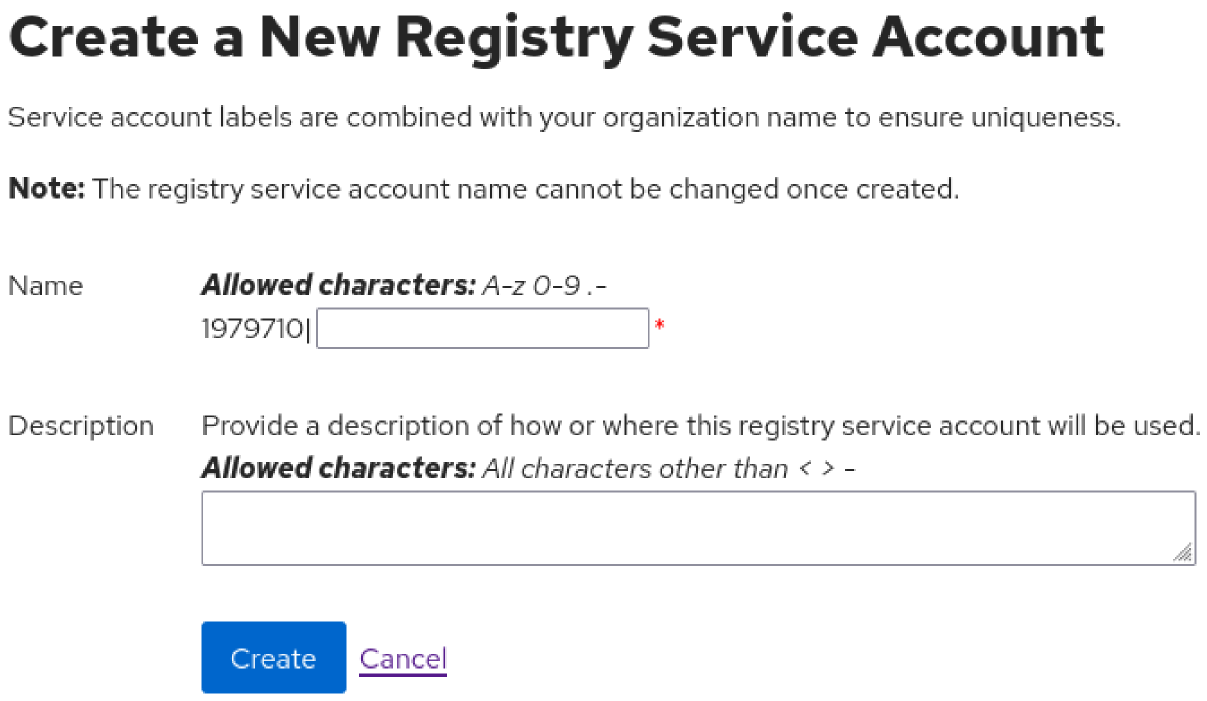


Registry Service Accounts

Registry service accounts are named tokens that can be used in environments where credentials will be shared, such as deployment systems.

Note: Registry service accounts can only be deleted or regenerated by an organization administrator or the person that created it.

Click on the blue "New Service Account"



Create a New Registry Service Account

Service account labels are combined with your organization name to ensure uniqueness.

Note: The registry service account name cannot be changed once created.

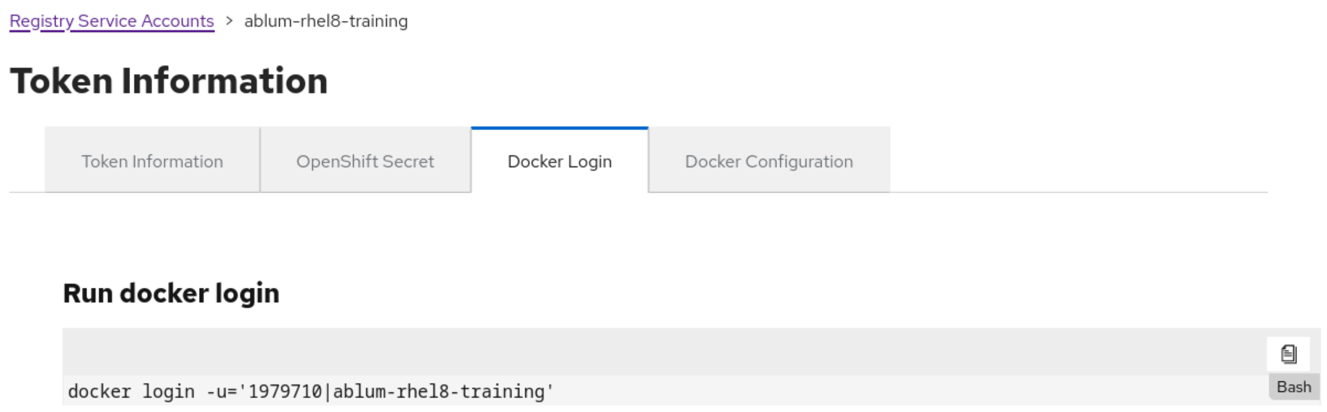
Name *Allowed characters:* A-z 0-9 .-
1979710| *

Description Provide a description of how or where this registry service account will be used.
Allowed characters: All characters other than < > -

Fill in a name that includes some value that you will remember. Additionally, you can enter a Description as well. Then click **Create**.

This will create a new **Registry Service Account** you can use with podman as well Red Hat Openshift to authenticate to `registry.redhat.io`

To test, navigate to the newly created Service Account. Click the tab **Docker Login** like in this example:



```
docker login -u='1979710|ablum-rhel8-training'
```

Copy the login command and paste it into an editor like `gedit`. You will need to modify the command so that `podman login` is used and not `docker login`.

With the modified `podman login` run it on the workstation machine:

```
[student@workstation ~]$ podman login -u='your-service-account-name' -p=...SNIP...
registry.redhat.io
Login Succeeded!
```

The service account credentials are stored in `${XDG_RUNTIME_DIR}/containers/auth.json` just as the customer portal credentials were before. The service account credentials, however, are limited to providing registry access through the registry service and can be removed/revoked through the customer portal.

3.16. What is the expiration for authentication access tokens?

Normal username and password authentication with registry.redhat.io will result in the user's info being cached in `${XDG_RUNTIME_DIR}/containers/auth.json`

```
{
    "auths": {
        "registry.redhat.io": {
            "auth": "some00000random1111string"
        }
    }
}
```

When running `podman` commands `podman pull`, these credentials are used to obtain a token that is short-lived (300s). For example,

```
[student@workstation ~]$ curl -L -u myusername
```

```
"https://sso.redhat.com/auth/realms/rhcc/protocol/redhat-docker-v2/auth?service=docker-
registry&client_id=curl&scope=repository:rhel:pull"
Enter host password for user 'myusername':

{"token":...SNIP...,"expires_in":300,"issued_at":"2022-09-22T19:04:35Z"}
```

You can try the example above by replacing `myusername` with the login you use to the customer portal.

As given in this output, this short-lived token will expire after **300** seconds. After which time, a new request will need to be made for a new token using the cached credentials in `${XDG_RUNTIME_DIR}/containers/auth.json`

SEE https://access.redhat.com/articles/3560571 for more help using curl to troubleshoot authentication issues.

3.17. Working with different container formats using `podman save`

Suppose you would like to archive of `registry.redhat.io/rhscl/httpd-24-rhel7:latest` to a local file.

```
[student@workstation ~]$ podman images
REPOSITORY                         TAG         IMAGE ID      CREATED     SIZE
registry.redhat.io/rhscl/httpd-24-rhel7  latest      87504a9170d1  7 days ago  332 MB
```

One method would be to use `podman save` like the following:

```
[student@workstation ~]$ mkdir httpd-save

[student@workstation ~]$ podman save -o httpd-save/httpd.tar
registry.redhat.io/rhscl/httpd-24-rhel7:latest
Copying blob 7f77859de5cc done
Copying blob d668157579e4 done
Copying blob ab67ee0ba256 done
Copying blob e63854ca1fdd done
Copying config 87504a9170 done
Writing manifest to image destination
Storing signatures
```

Check the contents of `httpd.tar`

student@workstation ~]$ cd httpd-save/ [student@workstation httpd-save]$ [student@workstation httpd-save]$ tar xvf httpd.tar ---

This will expand the archive into a collection of files including several `json` files and others with `.tar` extensions. An inventory of these files is stored in the `manifest.json`:

```
[student@workstation httpd-save]$ cat manifest.json | json_reformat
[
    {
        "Config":
"87504a9170d1a4a5c10e0fffb69f1bf32f9e94362d96edab0a0c781c2c471548.json",
        "RepoTags": [
            "registry.redhat.io/rhscl/httpd-24-rhel7:latest"
        ],
        "Layers": [
            "7f77859de5ccd6cc3e73934985132cd2ac67a2f4fe46db2a8a2e8684317c39d5.tar",
            "d668157579e4aa075a093bcd581b9266c892079200632dbd9ff0f3deae4d4f1c.tar",
            "ab67ee0ba256b41dddbabfdfbe3dad2f587335c8e224138cbe1196f867dae8b2.tar",
            "e63854ca1fdd26fd51acce6e047e6ac78aae7da71d85c9b3847418c5e3c911a7.tar"
        ]
    }
]
```

The file identified in the `Config` is also a `json` file. It contains the image metadata typically explored using commands like `podman inspect`:

```
[student@workstation httpd-save]$ cat
87504a9170d1a4a5c10e0fffb69f1bf32f9e94362d96edab0a0c781c2c471548.json | jq | head
{
  "architecture": "amd64",
  "config": {
    "Hostname": "771ed3e47dd5",
    "Domainname": "",
    "User": "1001",
    "AttachStdin": false,
...SNIP...
```

The image layers are contained in the `.tar` archives.

```
[student@workstation httpd-save]$ ls *.tar
7f77859de5ccd6cc3e73934985132cd2ac67a2f4fe46db2a8a2e8684317c39d5.tar
d668157579e4aa075a093bcd581b9266c892079200632dbd9ff0f3deae4d4f1c.tar
ab67ee0ba256b41dddbabfdfbe3dad2f587335c8e224138cbe1196f867dae8b2.tar
e63854ca1fdd26fd51acce6e047e6ac78aae7da71d85c9b3847418c5e3c911a7.tar

[student@workstation httpd-save]$ tar tvf
7f77859de5ccd6cc3e73934985132cd2ac67a2f4fe46db2a8a2e8684317c39d5.tar | head
dr-xr-xr-x root/root         0 2022-09-15 10:30 ./
dr-xr-xr-x root/root         0 2017-12-14 12:23 ./boot/
```

```
drwxr-xr-x root/root         0 2022-09-15 10:28 ./dev/
drwxr-xr-x root/root         0 2022-09-15 10:28 ./proc/
drwxr-xr-x root/root         0 2022-09-15 10:30 ./run/
drwxr-xr-x root/root         0 2022-09-15 10:30 ./run/sepermit/
drwxr-xr-x root/root         0 2022-09-15 10:30 ./run/log/
drwxr-xr-x root/root         0 2022-09-15 10:30 ./run/lock/
drwxr-xr-x root/root         0 2022-09-15 10:30 ./run/lock/subsys/
drwxrwxr-x root/lock         0 2022-09-15 10:30 ./run/lock/lockdev/
```

These layers will make up the contents of the `lowerdir` in the `overlayfs` mount.

The format used in this example is one of the original image formats based on docker v1. From `man podman-save --format` can be used to generate images in different formats according to this table:

Format	Description
docker-archive	A tar archive interoperable with docker load(1) (the default)
oci-archive	A tar archive using the OCI Image Format
oci-dir	A directory using the OCI Image Format
docker-dir	dir transport (see containers-transports(5)) with v2s2 manifest type

Try to save using `--format=docker-dir`

```
[student@workstation httpd-save]$ rm -rf ~/httpd-save/*

[student@workstation httpd-save]$ podman save -o ~/httpd-save --format=docker-dir
registry.redhat.io/rhscl/httpd-24-rhel7:latest

[student@workstation httpd-save]$ cat manifest.json | json_reformat
{
    "schemaVersion": 2,
    "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
    "config": {
        "mediaType": "application/vnd.docker.container.image.v1+json",
        "size": 8166,
        "digest":
"sha256:87504a9170d1a4a5c10e0fffb69f1bf32f9e94362d96edab0a0c781c2c471548"
    },
```

To save as `oci-archive`

```
[student@workstation httpd-save]$ rm -rf ~/httpd-save/*
```

```
[student@workstation httpd-save]$ podman save -o ~/httpd-save/httpd.tar --format=oci-
archive registry.redhat.io/rhscl/httpd-24-rhel7:latest
```

Then extract to explore the contents:

```
[student@workstation httpd-save]$ tar xvf httpd.tar
blobs/
blobs/sha256/
blobs/sha256/275175e1b487d514267f6a455d07e61f704facd3e5d871ad1e74cf2f4f503af9
blobs/sha256/4897ac2964752dc45938d9fded59412a245429903d5facfbc507b21ce48ea2c0
blobs/sha256/939a1202230b5fb245052c0f3458565307cde22fcb88fa3498410e9d4e09ed15
blobs/sha256/981b2ef3f241320f27d8e989922b5efa0602527fa67891055f335a116b6131b5
blobs/sha256/d336fcea58570b9194f5d07e09cfa82cb1b92ce53f35694ef2269716b92bd786
blobs/sha256/da7f54b2dbe3875198ebf2fd99c131ef84a820d025226cd0b2356f4531e4241a
index.json
oci-layout

[student@workstation httpd-save]$ cat index.json | json_reformat
{
    "schemaVersion": 2,
    "manifests": [
        {
            "mediaType": "application/vnd.oci.image.manifest.v1+json",
            "digest":
"sha256:981b2ef3f241320f27d8e989922b5efa0602527fa67891055f335a116b6131b5",
            "size": 879,
            "annotations": {
                "org.opencontainers.image.ref.name": "registry.redhat.io/rhscl/httpd-24-
rhel7:latest"
            }
        }
    ]
}
```

The OCI Image spec can be found here: https://github.com/opencontainers/image-spec/blob/main/spec.md

Cleanup with:

```
[student@workstation httpd-save]$ cd ~
[student@workstation ~]$ rm -rf httpd-save
[student@workstation ~]$
```

3.18. What does the message "Storing signatures" mean?

When working with various container tools while retrieving or copying a container image, you may have noticed the message `Storing signatures` like:

```
[student@workstation ~]$ podman pull registry.redhat.io/ubi8:latest
Trying to pull registry.redhat.io/ubi8:latest...
...SNIP...
Storing signatures
10f854072e7e7b7a715bcd78cf7925851159f9db82a2ff1c9b35806356352029
```

or

```
[student@workstation ~]$ skopeo copy docker://registry.redhat.io/ubi8:latest containers-
storage:localhost/ubi8:latest
Storing signatures
```

In either case, the local storage will include references to these images:

```
[student@workstation ~]$ podman images
REPOSITORY                               TAG         IMAGE ID      CREATED      SIZE
localhost/ubi8                           latest      10f854072e7e  2 weeks ago  227 MB
registry.redhat.io/ubi8                  latest      10f854072e7e  2 weeks ago  227 MB
```

In this case, these images were copied from Red Hat’s container registry. Red Hat signs the containers it distributes so that any system can verify the containers came from Red Hat.

`podman` and `skopeo` will use the image trust configured in `/etc/containers/policy.json`

```
[student@workstation ~]$ podman image trust show
default                     accept
registry.access.redhat.com  signedBy                security@redhat.com,
security@redhat.com  https://access.redhat.com/webassets/docker/content/sigstore
registry.redhat.io          signedBy                security@redhat.com,
security@redhat.com  https://registry.redhat.io/containers/sigstore
                            insecureAcceptAnything
```

Inspect `/etc/containers/policy.json`

```
[student@workstation ~]$ cat /etc/containers/policy.json
"transports": {
    "docker": {
```

```
  "registry.access.redhat.com": [
{
    "type": "signedBy",
    "keyType": "GPGKeys",
    "keyPath": "/etc/pki/rpm-gpg/RPM-GPG-KEY-redhat-release"
}
  ],
  "registry.redhat.io": [
{
    "type": "signedBy",
    "keyType": "GPGKeys",
    "keyPath": "/etc/pki/rpm-gpg/RPM-GPG-KEY-redhat-release"
}
  ]
},
```

This indicates that images are validated from `registry.access.redhat.com` and `registry.redhat.io` using the same GPGKeys `rpm` uses to validate software packages installed through the RHEL package manager (ie yum/rpm).

Once the container images are copied to the local system however those signatures are removed.

Some tools like `skopeo copy` will report fatal messages when using an archive destination transport like oci-archive:

```
[student@workstation ~]$ skopeo copy docker://registry.redhat.io/ubi8:latest oci-
archive://home/student/mytest/ubi8.tar
Getting image source signatures
Checking if image destination supports signatures
FATA[0001] Can not copy signatures to oci-archive://home/student/mytest/ubi8.tar::
Pushing signatures for OCI images is not supported
```

This is expected. `podman pull` will remove the signatures after validating but does so silently.

From https://github.com/containers/skopeo/issues/589

> Neither Docker daemon nor OCI supports storing the signatures, and `skopeo copy` refuses to just silently drop them.
>
> You can use skopeo copy --remove-signatures to make the copy anyway; the signatures will be still read and policy.json will still be enforced, they just won’t be written into the destination.

Try making the `skopeo copy` again using the `--remove-signatures` option:

```
[student@workstation ~]$ skopeo copy docker://registry.redhat.io/ubi8:latest oci-
archive://home/student/mytest/ubi8.tar --remove-signatures
Copying blob 1b3417e31a5e done
Copying blob 809fe483e885 done
Copying config 5291d146cb done
Writing manifest to image destination
Storing signatures
```

Worked! This oci-archive can be loaded into the local container storage for use by the container runtime with:

```
[student@workstation ~]$ podman load -i /home/student/mytest/ubi8.tar
Getting image source signatures
Copying blob 1b3417e31a5e skipped: already exists
Copying blob 809fe483e885 skipped: already exists
Copying config 5291d146cb done
Writing manifest to image destination
Storing signatures
Loaded image(s): sha256:5291d146cbbe8d356ca11a987a2b2c44269a768d460afa101e01ed7e7fb245b8
```

3.19. Do images from other public registries include signatures?

It depends on the image. Many images are not signed at all. It is also possible that they are signed but verification on local container host has not been configured with `podman image trust`

```
[student@workstation ~]$ rm -rf mytest

[student@workstation ~]$ mkdir mytest

[student@workstation ~]$ skopeo copy docker://quay.io/ajblum/hello-openshift:latest
dir:/home/student/mytest

[student@workstation ~]$ ls /home/student/mytest
7af3297a3fb4487b740ed6798163f618e6eddea1ee5fa0ba340329fcae31c8f6
b30065c58b6f2272f190bddd84e9adb6900f8946f92900e18d19622413d3ebc0  version
a3ed95caeb02ffe68cdd9fd84406680ae93d633cb16422d00e8a7c22955b46d4  manifest.json
```

Compare with:

```
[student@workstation ~]$ mkdir mytest1
```

```
[student@workstation ~]$ skopeo copy docker://registry.redhat.io/ubi8:latest
dir:/home/student/mytest1

[student@workstation ~]$ ls /home/student/mytest1
10f854072e7e7b7a715bcd78cf7925851159f9db82a2ff1c9b35806356352029  manifest.json
signature-3  signature-6
1b3417e31a5e0e64f861e121d4efed3152e75aaa85026cd784cd0070e063daa3  signature-1
signature-4  version
809fe483e88523e7021d76b001a552856f216430023bdc0aeff8fce8df385535  signature-2
signature-5
```

If you would like to sign your custom images with your personal GPG signatures consider:https://developers.redhat.com/blog/2019/10/29/verifying-signatures-of-red-hat-container-images

Chapter 4. Chapter 5: Creating Custom Container Images

4.1. Why do I get permission denied building images after running su - (or sudo su -) ?

It is common for container build environments to run as a specific user. Gitlab, for example, uses the `gitlab-runner` user when executing CI/CD jobs that include `buildah` or `podman build` commands. Sometimes using different users can impact authentication when building container images.

Consider this example where the `devops` user on the `workstation` machine is used for building container images.

First, assume that as the `student` user, you have already logged into the registry `registry.redhat.io`:

```
[student@workstation ~]$ podman login --get-login registry.redhat.io
rhn-support-ablum
```

Instead of building with the `student` user, the `devops` user will be executing the `podman build`. Become the `devops` user and create a simple `Containerfile` that leverages the `uib8:latest` image from `registry.redhat.io`:

```
[student@workstation ~]$ sudo su - devops

[devops@workstation ~]$  mkdir echo1

[devops@workstation ~]$ cd echo1/

[devops@workstation echo1]$ vim Containerfile

FROM registry.redhat.io/ubi8:latest
ENTRYPOINT ["/usr/bin/echo", "Hello World!"]
```

Now, try and build this:

```
[devops@workstation echo1]$ podman build -t mytest:1.0 .
STEP 1/2: FROM registry.redhat.io/ubi8:latest
Trying to pull registry.redhat.io/ubi8:latest...
Error: error creating build container: initializing source
docker://registry.redhat.io/ubi8:latest: unable to retrieve auth token: invalid
username/password: unauthorized: Please login to the Red Hat Registry using your Customer
```

```
Portal credentials. Further instructions can be found here:
https://access.redhat.com/RegistryAuthentication
```

Logging into `registry.redhat.io` will satisfy the authentication requirements when working with `registry.redhat.io`.

```
[devops@workstation ~]$ podman login -u rhn-support-ablum registry.redhat.io
Password:
Login Succeeded!
[devops@workstation ~]$
```

But, consider that the credentials are being cached in a surprising location.

From `man podman-login`:

> The default path for reading and writing credentials is `${XDG_RUNTIME_DIR}/containers/auth.json`

In our example here, the `auth.json` does not exist with this path:

```
[devops@workstation ~]$ cat ${XDG_RUNTIME_DIR}/containers/auth.json
cat: /containers/auth.json: No such file or directory

[devops@workstation ~]$ echo ${XDG_RUNTIME_DIR}

[devops@workstation ~]$
```

Why? `${XDG_RUNTIME_DIR}` was not set. This environment variable is set via `pam_systemd`:

From `man pam_systemd`:

> On login, this module — in conjunction with systemd-logind.service — ensures the following:
>
> If it does not exist yet, the user runtime directory /run/user/$UID is either created or mounted as new "tmpfs" file system with quota applied, and its ownership changed to the user that is logging in.

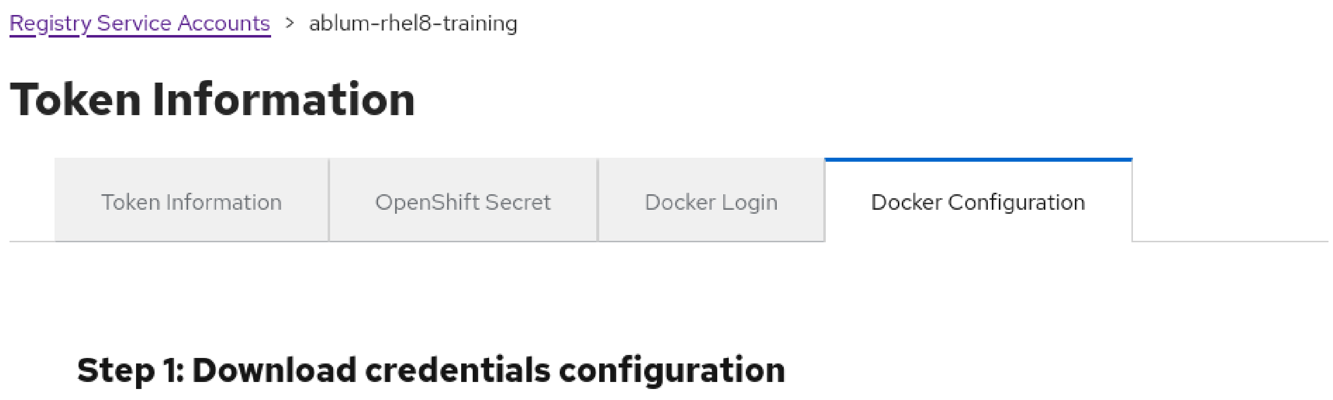
In our case, `podman login` will write to `/tmp` specifically:

```
[devops@workstation ~]$ ls /tmp/podman-run-1001/containers/auth.json
/tmp/podman-run-1001/containers/auth.json
```

Therefore, for a build environment, a better strategy would be to leverage an authfile, one that uses a service account (https://access.redhat.com/terms-based-registry/)

First login into https://access.redhat.com/terms-based-registry/ and navigate to your service account.

Click on the tab **Docker Configuration**:


Registry Service Accounts > ablum-rhel8-training
Token Information
Token Information
OpenShift Secret
Docker Login
Docker Configuration
Step 1: Download credentials configuration

Click **view its contents**. You will see the authfile contents in your browser. Paste those into `~/.docker/config.json`:

```
[devops@workstation ~]$ podman logout registry.redhat.io
[devops@workstation ~]$ mkdir ~/.docker
[devops@workstation ~]$ vim ~/.docker/config.json
{
  "auths": {
    "registry.redhat.io": {
      "auth": "MTk.......SNIP......Q=="
    }
  }
}
```

Building should now be successful without needing to run `podman login` beforehand.

```
[devops@workstation ~]$ cd echo1/

[devops@workstation echo1]$ podman build -t mytest:1.0 .
STEP 1/2: FROM registry.redhat.io/ubi8:latest
Trying to pull registry.redhat.io/ubi8:latest...
...SNIP...
Writing manifest to image destination
Storing signatures
STEP 2/2: ENTRYPOINT ["/usr/bin/echo", "Hello World!"]
COMMIT mytest:1.0
--> 558683ae8b3
```

```
Successfully tagged localhost/mytest:1.0
558683ae8b33168e0b10a4abb79245aedee8e6e4c3c317099c37d454945365b3

[devops@workstation echo1]$ podman run mytest:1.0
Hello World!
```

Alternatively you can leverage the `--authfile` option to specify a different path as required by your build environment. The contents of this file are the same as the `config.json` provided by the Customer Portal service account.

```
[devops@workstation echo1]$ podman build -t mytest:1.0 --authfile ~/.docker/config.json .
STEP 1/2: FROM registry.redhat.io/ubi8:latest
Trying to pull registry.redhat.io/ubi8:latest...
...SNIP...
```

4.2. ENTRYPOINT vs CMD

From `man 5 Contianerfile` in Fedora:

ENTRYPOINT = An ENTRYPOINT helps you configure a container that can be run as an executable. When you specify an ENTRYPOINT, the whole container runs as if it was only that executable.

CMD = The main purpose of a CMD is to provide defaults for an executing container. These defaults may include an executable, or they can omit the executable.

It is best to understand the relationship between these with examples. First, create the following **Hello World** `Contianerfile`:

```
[student@workstation ~]# mkdir echo

[student@workstation ~]# cd echo

[student@workstation echo]# vim Containerfile
FROM registry.redhat.io/ubi8:latest
ENTRYPOINT ["/usr/bin/echo", "Hello world!"]
```

To build this container run the following `podman build` command:

```
[student@workstation echo]# podman build -t ubi8:echo1 .
```

Test the newly built container by running with:

```
[student@workstation echo]# podman run ubi8:echo1
Hello world!
```

The image metadata for the newly built container includes the `Entrypoint` key, but no Cmd:

```
[student@workstation echo1]$ podman inspect ubi8:echo1 --format '{{.Config.Entrypoint}}'
[/usr/bin/echo Hello world!]

[student@workstation echo1]$ podman inspect ubi8:echo1 --format '{{.Config.Cmd}}'
[]
```

Now, lets add in a CMD instruction. Move the `Hello world!` string from the `ENTRYPOINT` into the `CMD` instruction like:

```
[student@workstation echo]$ vim Containerfile

FROM registry.redhat.io/ubi8:latest
ENTRYPOINT ["/usr/bin/echo"]
CMD ["Hello world!"]
```

Build and run this container with the following:

```
[student@workstation echo]$  podman build -t ubi8:echo2 .
STEP 1/3: FROM registry.redhat.io/ubi8:latest
STEP 2/3: ENTRYPOINT ["/usr/bin/echo"]
--> d46473bb34f
STEP 3/3: CMD ["Hello world!"]
COMMIT ubi8:echo2
--> d4c6395eb2f
Successfully tagged localhost/ubi8:echo2
d4c6395eb2ff76a8f78461d201d7867054ae8e422cd7a862b456ac547e5c5319

[student@workstation echo]$ podman run ubi8:echo2
Hello world!
```

Use `podman inspect` to verify the Entrypoint and Cmd in the `ubi8:echo2` metadata:

```
[student@workstation echo]$  podman inspect ubi8:echo2 --format '{{.Config.Entrypoint}}'
[/usr/bin/echo]
[student@workstation echo]$  podman inspect ubi8:echo2 --format '{{.Config.Cmd}}'
[Hello world!]
```

Compare execution using `ubi8:echo1` with `ubi8:echo2`

```
[student@workstation echo]$ podman run ubi8:echo1
Hello world!
[student@workstation echo]$ podman run ubi8:echo2
Hello world!
```

No noticeable difference.

To highlight the difference, override the CMD built in the image when running these two:

```
[student@workstation echo]$ podman run ubi8:echo1 foo
Hello world! foo
[student@workstation echo]$ podman run ubi8:echo2 foo
foo
```

Try to use some command line options supported by the `echo` command:

```
[student@workstation echo]$ podman run ubi8:echo1 -ne foo
Hello world! -ne foo
[student@workstation echo]$ podman run ubi8:echo2 -ne foo
foo[student@workstation echo]$
```

Notice how the extra command arguments are echo'd on the terminal with `echo1`. This is probably not what you wanted. Instead, with `echo2`, the `-ne` arguments influence the behavior of the echo command (ie suppressing newline).

A container developer can influence the runtime with simple choices like what to include in ENTRYPOINT and CMD. Taken together, ENTRYPOINT + CMD, will be the process run in isolation.

4.3. Exec vs Shell

ENTRYPOINT and **CMD** instructions allow two different syntax:

Exec

```
# Executable form
ENTRYPOINT ["executable", "param1", "param2"]
```

Shell

```
# the command is run in a shell - /bin/sh -c
```

```
ENTRYPIOINT <command>
```

To better understand how that affects the runtime, create the following Containerfile:

```
[student@workstation echo]$ vim Containerfile
FROM registry.redhat.io/ubi8:latest
ENV FOO "Hello World!"
ENTRYPOINT ["/usr/bin/echo","$FOO"]
```

The intent of this Containerfile is to build a container image that will `echo` the string "Hello World!" on the terminal when run.

Build this into an image and give it a try:

```
[student@workstation echo]$ podman build -t ubi8:echo3 .
STEP 1/3: FROM registry.redhat.io/ubi8:latest
STEP 2/3: ENV FOO "Hello World!"
--> 7fde4a72da5
STEP 3/3: ENTRYPOINT ["/usr/bin/echo","$FOO"]
COMMIT ubi8:echo3
--> 3676c6dbf81
Successfully tagged localhost/ubi8:echo3
3676c6dbf81088a5a8a04646207bf255ac51e142c03729a7922ccac4470ca7d1

[student@workstation echo]$ podman run ubi8:echo3
$FOO
```

Did it do what we were expecting ? Why not ? Let's override the entrypoint and verify the environment variable `FOO` has a value set equal to "Hello World!"

```
[student@workstation echo]$ podman run --entrypoint "env" ubi8:echo3
FOO=Hello World!
```

Yes. Looks good. Yet, the value of `$FOO` is not being evaluated before passing it to the `/usr/bin/echo` command.

The form used in this example is *Exec* form. With *exec* form, the entrypoint executable isn't run inside a shell. Environment variables are available for use inside a shell. So, let's change to use the shell form and observe the behavior:

```
[student@workstation echo]$ vim Containerfile
FROM registry.redhat.io/ubi8:latest
ENV FOO "Hello World!"
```

```
ENTRYPOINT /usr/bin/echo $FOO

[student@workstation echo]$ podman build -t ubi8:echo4 .

[student@workstation echo]$ podman run ubi8:echo4
Hello World!
```

Compare the Entrypoint set in the metadata for each of these two container images:

```
[student@workstation echo]$ podman inspect ubi8:echo3 --format '{{.Config.Entrypoint}}'
[/usr/bin/echo $FOO]
[student@workstation echo]$ podman inspect ubi8:echo4 --format '{{.Config.Entrypoint}}'
[/bin/sh -c /usr/bin/echo $FOO]
```

Notice how `ubi8:echo4` has the `/usr/bin/echo $FOO` wrapped in a shell. The shell `/bin/sh` evaluates the `$FOO` variable before passing to `/usr/bin/echo`. This is why `ubi8:echo4` produces the result we expected. This is also why it is known as the **shell** format.

4.4. Which is preferred, exec or shell form?

shell form is helpful for some applications that are not capable of sourcing in environment variables on their own. Additionally, *shell* form prevents any CMD from affecting the way the entrypoint runs. For example,

```
[student@workstation echo]$ podman inspect ubi8:echo4 --format '{{.Config.Entrypoint}}'
[/bin/sh -c /usr/bin/echo $FOO]

[student@workstation echo]$ podman run ubi8:echo4 adshfjdkljsf
Hello World!
```

Remember `podman stop`? If you are using the *shell* form, the shell is sent a SIGTERM after `podman stop`. That signal is then passed down to the child process of the shell. With *exec* form, the containerized process receives the signal directly. It is for this reason *exec* preferred in most cases.

Many containers are built with unique shell scrips (ie wrapper or init scripts) that will `exec` the application after setting up the runtime environment. The

A good example of an image built this way is the Apache daemon from Red Hat Software Collections. Pull and inspect it:

```
[student@workstation echo]$ podman pull registry.access.redhat.com/rhscl/httpd-24-
rhel7:latest
Trying to pull registry.access.redhat.com/rhscl/httpd-24-rhel7:latest...
```

```
[student@workstation echo]$ podman inspect registry.access.redhat.com/rhscl/httpd-24-
rhel7:latest --format '{{.Config.Entrypoint}}'
[container-entrypoint]
[student@workstation echo]$ podman inspect registry.access.redhat.com/rhscl/httpd-24-
rhel7:latest --format '{{.Config.Cmd}}'
[/usr/bin/run-httpd]
```

Notice the *exec* form used for both the ENTRYPOINT and CMD. Now execute it with an interactive shell:

```
[student@workstation echo]$ podman run -it --entrypoint /bin/bash
registry.access.redhat.com/rhscl/httpd-24-rhel7:latest

bash-4.2$ which container-entrypoint
/usr/bin/container-entrypoint

bash-4.2$ cat /usr/bin/container-entrypoint
#!/bin/bash
exec "$@"


bash-4.2$ tail -3 /usr/bin/run-httpd
process_extending_files ${HTTPD_APP_ROOT}/src/httpd-pre-init/
${HTTPD_CONTAINER_SCRIPTS_PATH}/pre-init/

exec httpd -D FOREGROUND $@
```

In this case, `rhscl/httpd-24-rhel7:latest` is using the *exec* form executing wrapper scripts that include the `exec` bash built-in command. If you use the *exec* form this is a great way to do it. Just remember to use the `exec` commands in your wrapper scripts.

4.5. What impact does the RUN instruction have on Image layers?

To explore this question, create a container image with one RUN instruction like this:

```
[student@workstation ~]$ mkdir myls

[student@workstation ~]$ cd myls/

[student@workstation myls]$ vim Containerfile

FROM registry.redhat.io/ubi8:latest
ENTRYPOINT ["/usr/bin/ls"]
```

```
RUN touch /var/tmp/data

[student@workstation myls]$ podman build -t ubi8:myls1 .
STEP 1/3: FROM registry.redhat.io/ubi8:latest
STEP 2/3: ENTRYPOINT ["/usr/bin/ls"]
--> f0a1e9e7e7a
STEP 3/3: RUN touch /var/tmp/data
COMMIT ubi8:myls1
--> c7fa2956a3a
Successfully tagged localhost/ubi8:myls1
c7fa2956a3af3a388ab4a36736ac90aefc03e28ff7fc5e09c40b676720041943
```

Try running this container image to verify the presence of `/var/tmp/data`:

```
[student@workstation myls]$ podman run ubi8:myls1 /var/tmp/data
/var/tmp/data
[student@workstation myls]$ podman run ubi8:myls1 -l /var/tmp/data
-rw-r--r--. 1 root root 0 Sep 26 17:46 /var/tmp/data
```

This works because the entrypoint is `/usr/bin/ls` and the command argument was `/var/tmp/data`. Thus `/usr/bin/ls /var/tmp/data` was executed.

Check the number of LowerDir's defined for this image:

```
[student@workstation myls]$ podman inspect ubi8:myls1 --format
'{{.GraphDriver.Data.LowerDir}}'
/home/student/.local/share/containers/storage/overlay/f4a999c201294a0a171618e413f1ea76285
62e96d8ac148e00708e421aee56a8/diff:/home/student/.local/share/containers/storage/overlay/
b38cb92596778e2c18c2bde15f229772fe794af39345dd456c3bf6702cc11eef/diff
[student@workstation myls]$
```

Two different directories defined in LowerDir for `ubi8:myls1`

Compare that with the original parent image `ubi8:latest`:

```
[student@workstation myls]$ podman inspect ubi8:latest --format
'{{.GraphDriver.Data.LowerDir}}'
/home/student/.local/share/containers/storage/overlay/b38cb92596778e2c18c2bde15f229772fe7
94af39345dd456c3bf6702cc11eef/diff
```

Only one here. So the RUN instruction used in the Containerfile resulted in an extra LowerDir. Take it to an extreme with:

```
[student@workstation myls]$ vim Containerfile

FROM registry.redhat.io/ubi8:latest
ENTRYPOINT ["/usr/bin/ls"]
RUN touch /var/tmp/data0
RUN touch /var/tmp/data1
RUN touch /var/tmp/data2
RUN touch /var/tmp/data3
RUN touch /var/tmp/data4
RUN touch /var/tmp/data5
RUN touch /var/tmp/data6

[student@workstation myls]$ podman build -t ubi8:myls2 .
...SNIP...
STEP 9/9: RUN touch /var/tmp/data6
COMMIT ubi8:myls2
--> 267e6d287a5
Successfully tagged localhost/ubi8:myls2
267e6d287a5a732263c735eb705b546fa8279d2436731038d8607ef163714120
```

Now test the new image `ubi8:myls2` with:

```
[student@workstation myls]$ podman run ubi8:myls2 /var/tmp
data0
data1
data2
data3
data4
data5
data6

[student@workstation myls]$ podman run ubi8:myls2 -l /var/tmp
total 0
-rw-r--r--. 1 root root 0 Sep 26 17:54 data0
-rw-r--r--. 1 root root 0 Sep 26 17:54 data1
-rw-r--r--. 1 root root 0 Sep 26 17:54 data2
-rw-r--r--. 1 root root 0 Sep 26 17:54 data3
-rw-r--r--. 1 root root 0 Sep 26 17:54 data4
-rw-r--r--. 1 root root 0 Sep 26 17:54 data5
-rw-r--r--. 1 root root 0 Sep 26 17:54 data6
```

Compare the `LowerDir` of `ubi8:myls1` and `ubi8:myls2`:

```
[student@workstation myls]$ podman inspect ubi8:myls1 --format
```

```
='{{.GraphDriver.Data.LowerDir}}'
/home/student/.local/share/containers/storage/overlay/f4a999c201294a0a171618e413f1ea76285
62e96d8ac148e00708e421aee56a8/diff:/home/student/.local/share/containers/storage/overlay/
b38cb92596778e2c18c2bde15f229772fe794af39345dd456c3bf6702cc11eef/diff

[student@workstation myls]$ podman inspect ubi8:myls2 --format
='{{.GraphDriver.Data.LowerDir}}'
/home/student/.local/share/containers/storage/overlay/4a823da36520d215e0dd0e35587d362e9d9
716d6f0b9814ca469eaf660632d54/diff:/home/student/.local/share/containers/storage/overlay/
069228d3067934f431ac04bd6e9a4013ffbf6fde006f0564d9c5833f04aeab2c/diff:/home/student/.loca
l/share/containers/storage/overlay/89d8628d96cbbaf385203cff30eea10904fac833c0f4d353c59656
f4bb9816b9/diff:/home/student/.local/share/containers/storage/overlay/222b495c111adf7dcd4
98257cf1b83a9d787c768a5a1bc5d737cc282823e3218/diff:/home/student/.local/share/containers/
storage/overlay/5046e9430e404340e18e98d254e237b8059de8f67ba1a21d40df78ed23d75cf7/diff:/ho
me/student/.local/share/containers/storage/overlay/d1dfa09a0a9412f5d7f53a91860fca78422ce3
6d0606abe7492a8f4a8c98bca6/diff:/home/student/.local/share/containers/storage/overlay/f4a
999c201294a0a171618e413f1ea7628562e96d8ac148e00708e421aee56a8/diff:/home/student/.local/s
hare/containers/storage/overlay/b38cb92596778e2c18c2bde15f229772fe794af39345dd456c3bf6702
cc11eef/diff
```

Wow! Significantly more directories defined in the `LowerDir` for `ubi8:myls2`. Each `RUN` instruction introduces more image layers.

To avoid these extra layers, combine `RUN` instructions into one single `RUN` instruction.

To do this, use a shell trick based on the bash `&&` list operator. From https://www.gnu.org/software/bash/manual/html_node/Lists.html

> command1 && command2
>
> command2 is executed if, and only if, command1 returns an exit status of zero (success).

Modify the `Containerfile` with the following:

```
FROM registry.redhat.io/ubi8:latest
ENTRYPOINT ["/usr/bin/ls"]
RUN touch /var/tmp/data0 && touch /var/tmp/data1 && touch /var/tmp/data2 && touch
/var/tmp/data3 && touch /var/tmp/data4 && touch /var/tmp/data5 && touch /var/tmp/data6
```

Notice the use of `&&` and the single `RUN` instruction.

Now build and test:

```
[student@workstation myls]$ podman build -t ubi8:myls3 .
```

```
STEP 1/3: FROM registry.redhat.io/ubi8:latest
STEP 2/3: ENTRYPOINT ["/usr/bin/ls"]
--> Using cache f0a1e9e7e7a732331c73192ace30d511bd0d184187b3334fc605b98301fd299c
--> f0a1e9e7e7a
STEP 3/3: RUN touch /var/tmp/data0 && touch /var/tmp/data1 && touch /var/tmp/data2 &&
touch /var/tmp/data3 && touch /var/tmp/data4 && touch /var/tmp/data5 && touch
/var/tmp/data6
COMMIT ubi8:myls3
--> 8c7ee4f9721
Successfully tagged localhost/ubi8:myls3
8c7ee4f97216575745e47516f9ee4a867888385b54a8c0104afd7da042171db8

[student@workstation myls]$ podman run ubi8:myls3 /var/tmp
data0
data1
data2
data3
data4
data5
data6
```

Looks to have the desired number of files, similar to `ubi8:myls2`. What about the size of the `LowerDir`:

```
[student@workstation myls]$ podman inspect ubi8:myls3 --format
='{{.GraphDriver.Data.LowerDir}}'
/home/student/.local/share/containers/storage/overlay/f4a999c201294a0a171618e413f1ea76285
62e96d8ac148e00708e421aee56a8/diff:/home/student/.local/share/containers/storage/overlay/
b38cb92596778e2c18c2bde15f229772fe794af39345dd456c3bf6702cc11eef/diff
```

Nice. Back to only two in the `LowerDir`

Stylistically, however, our `Containerfile` is difficult to read. To improve the readability without modifying the functionality, separate each command on a newline, but make sure to add an escape character `\`. This will insure that the `podman build` will execute all of the commands within one single `RUN` layer:

```
[student@workstation myls]$ vim Containerfile

FROM registry.redhat.io/ubi8:latest
ENTRYPOINT ["/usr/bin/ls"]
RUN touch /var/tmp/data0 && \
    touch /var/tmp/data1 && \
    touch /var/tmp/data2 && \
    touch /var/tmp/data3 && \
    touch /var/tmp/data4 && \
    touch /var/tmp/data5 && \
    touch /var/tmp/data6
```

Notice how the last `touch` command *does not* use the shell escape `\` since we do want to end the `RUN` instruction at that point.

Build and test this version:

```
[student@workstation myls]$ podman build -t ubi8:myls4 .
STEP 1/3: FROM registry.redhat.io/ubi8:latest
STEP 2/3: ENTRYPOINT ["/usr/bin/ls"]
--> Using cache f0a1e9e7e7a732331c73192ace30d511bd0d184187b3334fc605b98301fd299c
--> f0a1e9e7e7a
STEP 3/3: RUN touch /var/tmp/data0 &&     touch /var/tmp/data1 &&     touch
/var/tmp/data2 &&     touch /var/tmp/data3 &&     touch /var/tmp/data4 &&     touch
/var/tmp/data5 &&     touch /var/tmp/data6
COMMIT ubi8:myls4
--> bf35668c798
Successfully tagged localhost/ubi8:myls4
bf35668c7982c9bd096bd58018b5dd21b8c017248c4d4f45b96260052a9fc94c

[student@workstation myls]$ podman run ubi8:myls4 /var/tmp
data0
data1
data2
data3
data4
data5
data6
```

Check the `LowerDir` for the `ubi8:myls4`

```
[student@workstation myls]$ podman inspect ubi8:myls4 --format
='{{.GraphDriver.Data.LowerDir}}'
/home/student/.local/share/containers/storage/overlay/f4a999c201294a0a171618e413f1ea76285
62e96d8ac148e00708e421aee56a8/diff:/home/student/.local/share/containers/storage/overlay/
b38cb92596778e2c18c2bde15f229772fe794af39345dd456c3bf6702cc11eef/diff
```

Nice.

4.6. Can you squash the excessive layers when building?

Yes. Layers can be squashed using --squash or --squash-all

```
[student@workstation myls]$ podman build --help | grep squash
      --squash                                      squash newly built layers into a
single new layer
      --squash-all                                  Squash all layers into a single
layer

[student@workstation myls]$ vim Containerfile
FROM registry.redhat.io/ubi8:latest
ENTRYPOINT ["/usr/bin/ls"]
RUN touch /var/tmp/data0
RUN touch /var/tmp/data1
RUN touch /var/tmp/data2
RUN touch /var/tmp/data3
RUN touch /var/tmp/data4
RUN touch /var/tmp/data5
RUN touch /var/tmp/data6
```

Now build with:

```
[student@workstation myls]$ podman build -t ubi8:squash --squash .
STEP 1/9: FROM registry.redhat.io/ubi8:latest
STEP 2/9: ENTRYPOINT ["/usr/bin/ls"]
STEP 3/9: RUN touch /var/tmp/data0
STEP 4/9: RUN touch /var/tmp/data1
STEP 5/9: RUN touch /var/tmp/data2
STEP 6/9: RUN touch /var/tmp/data3
STEP 7/9: RUN touch /var/tmp/data4
STEP 8/9: RUN touch /var/tmp/data5
STEP 9/9: RUN touch /var/tmp/data6
COMMIT ubi8:squash
Getting image source signatures
Copying blob b38cb9259677 skipped: already exists
Copying blob 23e15b9ab3f0 skipped: already exists
Copying blob f4cb19500042 done
Copying config 1a958042d3 done
Writing manifest to image destination
Storing signatures
--> 1a958042d30
Successfully tagged localhost/ubi8:squash
1a958042d30d08789a566e09578d503d300b0dcb0e0b1b03ed39aaff885b12e4
```

Checking the `LowerDir`:

```
[student@workstation myls]$ podman inspect ubi8:squash --format
```

```
'{{.GraphDriver.Data.LowerDir}}'
/home/student/.local/share/containers/storage/overlay/f4a999c201294a0a171618e413f1ea76285
62e96d8ac148e00708e421aee56a8/diff:/home/student/.local/share/containers/storage/overlay/
b38cb92596778e2c18c2bde15f229772fe794af39345dd456c3bf6702cc11eef/diff
```

There is some storage savings you will get from keeping some of the common layers shared across your container runtime. If you use --squash-all then you will be left with no shared layers missing out on page cache and potentially increasing the overall storage use on your runtime host.

Try it with:

```
[student@workstation myls]$ podman build -t ubi8:squashall --squash-all .
[student@workstation myls]$ podman inspect ubi8:squashall --format
'{{.GraphDriver.Data.LowerDir}}'
<no value>
```

All the data in this case is in the image's UpperDir which will become a unique LowerDir when running the squashall image.

4.7. How can you use "OR" logic instead of "AND" logic like the &&

Try creating a `Containerfile` that includes `RUN` instructions with both `&&` and `||` operators:

```
[student@workstation ~]$ mkdir test1
[student@workstation ~]$ cd test1/
[student@workstation test1]$ vim Containerfile

FROM registry.redhat.io/ubi8:latest
ENTRYPOINT ["/usr/bin/ls"]
RUN touch /var/tmp/test1 && touch /var/tmp/test2
RUN touch /var/tmp/foo/bar/test3 || touch /var/tmp/test3
```

In the last `RUN` instruction above, the first `touch` command should fail since the directory `/var/tmp/foo/bar/` does not exist. Because of this failure the build environment will `touch /var/tmp/test3` instead.

Build with:

```
[student@workstation test1]$ podman build -t ubi8:test1 .
STEP 1/4: FROM registry.redhat.io/ubi8:latest
STEP 2/4: ENTRYPOINT ["/usr/bin/ls"]
--> Using cache f0a1e9e7e7a732331c73192ace30d511bd0d184187b3334fc605b98301fd299c
--> f0a1e9e7e7a
STEP 3/4: RUN touch /var/tmp/test1 && touch /var/tmp/test2
--> c64767dfee3
STEP 4/4: RUN touch /var/tmp/foo/bar/test3 || touch /var/tmp/test3
touch: cannot touch '/var/tmp/foo/bar/test3': No such file or directory
COMMIT ubi8:test1
--> 4c7e6b63a4a
Successfully tagged localhost/ubi8:test1
4c7e6b63a4aaf25e858f60869694c18827eea437c670d23faa1e027c3f4a53e9
```

Notice the failure `touch: cannot touch '/var/tmp/foo/bar/test3': No such file or directory`. Normally, this would be fatal for `podman build`. In this case, however, `STEP 4/4` is able to continue with the `||` operator.

Check the resulting image with:

```
[student@workstation test1]$ podman run ubi8:test1 /var/tmp
test1
test2
test3

[student@workstation test1]$ podman run ubi8:test1 /var/tmp/foo/bar
ls: cannot access '/var/tmp/foo/bar': No such file or directory
```

Try again using `&&`:

```
[student@workstation test1]$ vim Containerfile

FROM registry.redhat.io/ubi8:latest
ENTRYPOINT ["/usr/bin/ls"]
RUN touch /var/tmp/test1 && touch /var/tmp/test2
RUN touch /var/tmp/foo/bar/test3 && touch /var/tmp/test3

[student@workstation test1]$ podman build -t ubi8:test2 .
STEP 1/4: FROM registry.redhat.io/ubi8:latest
STEP 2/4: ENTRYPOINT ["/usr/bin/ls"]
--> Using cache f0a1e9e7e7a732331c73192ace30d511bd0d184187b3334fc605b98301fd299c
--> f0a1e9e7e7a
STEP 3/4: RUN touch /var/tmp/test1 && touch /var/tmp/test2
--> Using cache c64767dfee3b820e505c49f9698e6e4c8a29422e3377aec817870370bd32b34f
--> c64767dfee3
STEP 4/4: RUN touch /var/tmp/foo/bar/test3 && touch /var/tmp/test3
touch: cannot touch '/var/tmp/foo/bar/test3': No such file or directory
```

```
Error: error building at STEP "RUN touch /var/tmp/foo/bar/test3 && touch /var/tmp/test3":
error while running runtime: exit stat
```

This failure is fatal and no image is built as a result.

4.8. COPY vs ADD

1. They both add files to the container filesystem
2. COPY only copies local files,
3. ADD can "copy" plus decompress .tar or retrieve files from a URL

To investigate `COPY` further, create a new working directory along with a new file called `important`:

```
[student@workstation ~]$ mkdir mycat

[student@workstation ~]$ cd mycat

[student@workstation mycat]$ echo "helloworld" > important
```

Now, create a `Containerfile` using the `COPY` instruction to copy the `important` file from the host/build system into the container image:

```
[student@workstation mycat]$ vim Containerfile
FROM registry.redhat.io/ubi8:latest
ENTRYPOINT ["/bin/cat"]
COPY important /tmp
```

Build a new image using this `Containerfile`:

```
[student@workstation mycat]$ podman build -t ubi8:mycat1 .
STEP 1/3: FROM registry.redhat.io/ubi8:latest
STEP 2/3: ENTRYPOINT ["/bin/cat"]
--> c39def794fb
STEP 3/3: COPY important /tmp
COMMIT ubi8:mycat1
--> baa30df71dc
Successfully tagged localhost/ubi8:mycat1
baa30df71dcff5da36a37aaee1175693f219d1c11ea686c23bef515944745b61
```

To test, run this image passing the file `/tmp/important` to the `/bin/cat` ENTRYPOINT:

```
[student@workstation mycat]$ podman run ubi8:mycat1 /tmp/important
helloworld
```

Now, lets try and use `ADD` instead of `COPY`

```
[student@workstation mycat]$ vim Containerfile

FROM registry.redhat.io/ubi8:latest
ENTRYPOINT ["/bin/cat"]
ADD important /tmp
```

Build and test as `ubi8:mycat2`:

```
[student@workstation mycat]$ podman build -t ubi8:mycat2 .
[student@workstation mycat]$  podman run ubi8:mycat2 /tmp/important
helloworld
```

No real difference in the result. `ADD` has more features than `COPY` however. Try creating a `tar` file using the `important` file:

```
[student@workstation mycat]$ tar cvf important.tar important
important
[student@workstation mycat]$ rm important
[student@workstation mycat]$ ls
Containerfile  important.tar
```

Now, update the `Containerfile` to `ADD` this tarball `important.tar`:

```
[student@workstation mycat]$ vim Containerfile

FROM registry.redhat.io/ubi8:latest
ENTRYPOINT ["/bin/cat"]
ADD important.tar /tmp
```

Build and test as before:

```
[student@workstation mycat]$ podman build -t ubi8:mycat3 .

[student@workstation mycat]$ podman run ubi8:mycat3 /tmp/important
helloworld
```

ADD can also be used to inject content from a URL (ie similar to a wget)

4.9. What file permissions are used with ADD and COPY?

Watch out for file permissions when injecting files or archives via ADD or COPY. Consider this example:

```
[student@workstation mycat]$ vim Containerfile

FROM registry.redhat.io/ubi8:latest
ENTRYPOINT ["/bin/cat"]
RUN useradd test1
RUN useradd test2
RUN useradd foo
ADD ./important.tar /tmp/
USER foo
```

Next, build and test:

```
[student@workstation mycat]$ podman build -t ubi8:mycat4 .

[student@workstation mycat]$ podman run ubi8:mycat4 /tmp/important
helloworld
```

Looks good.

Look closer at who actually owns the file /tmp/important injected with ADD:

```
[student@workstation mycat]$ podman run -it --entrypoint /bin/bash  ubi8:mycat4
[foo@daa59a55ff75 /]$ whoami
foo
[foo@daa59a55ff75 /]$ id
uid=1002(foo) gid=1002(foo) groups=1002(foo)

[foo@daa59a55ff75 /]$ ls -l /tmp/important
-rw-rw-r--. 1 test1 test1 11 Sep 26 18:59 /tmp/important
```

Can the foo user modify /tmp/imporant ?

```
[student@workstation mycat]$ podman run -it --entrypoint /bin/bash  ubi8:mycat4

[foo@daa59a55ff75 /]$ echo update >> /tmp/important
bash: /tmp/important: Permission denied
```

This can create real problems for an application. Care with file permissions needs to be taken when using `COPY` or `ADD`.

Fix this issue by adding a `chown`. But, where should you add it?

After the `USER` instruction?

```
FROM registry.redhat.io/ubi8:latest
ENTRYPOINT ["/bin/cat"]
RUN useradd test1
RUN useradd test2
RUN useradd foo
ADD ./important.tar /tmp/
USER foo
RUN chown foo:foo /tmp/important
```

Or before the `USER` instruction?

```
FROM registry.redhat.io/ubi8:latest
ENTRYPOINT ["/bin/cat"]
RUN useradd test1
RUN useradd test2
RUN useradd foo
ADD ./important.tar /tmp/
RUN chown foo:foo /tmp/important
USER foo
```

The order does matter in this case. When the `chown` appears after the `USER` instruction, the `podman build` will fail with the message:

```
STEP 8/8: RUN chown foo:foo /tmp/important
chown: changing ownership of '/tmp/important': Operation not permitted
Error: error building at STEP "RUN chown foo:foo /tmp/important": error while running
runtime: exit status 1
```

Testing with the `chown` before the `USER` instruction will successfully build in this case.

Chapter 5. Chapter 6: Deploying Containerized Applications on Openshift

===

===

===

===

Chapter 6. <CHAPTER TITLE>

6.1. <SECTION TITLE>

Section Info Here

Listing 4. Example Code box for CLI

```
[student@workstation ~]$ sudo yum module install container-tools
```

Listing 5. Example Code box for YAML

```
---
- name: Deploy HTTPD Server Demo
  hosts: server
  collections:

  tasks:

## Start and Run the HTTPD Container
    - name:  Start the Apache Container
      podman_container:
```

Example 1. LAB/Exercise: Hands-On Activity Example

1. Download a container image.
    a. Registry: **registry.access.redhat.com**
    b. Image: **ubi7**
2. Run the container

6.1.1. <Section_Sub_Intro_Here>

Chapter 7. <CHAPTER TITLE>

7.1. <SECTION TITLE>

Section Info Here

Listing 6. Example Code box for CLI

```
[student@workstation ~]$ sudo yum module install container-tools
```

Listing 7. Example Code box for YAML

```
---
- name: Deploy HTTPD Server Demo
  hosts: server
  collections:

  tasks:

## Start and Run the HTTPD Container
    - name:  Start the Apache Container
      podman_container:
```

Example 2. LAB/Exercise: Hands-On Activity Example

1. Download a container image.
   a. Registry: **registry.access.redhat.com**
   b. Image: **ubi7**
2. Run the container

7.1.1. <Section_Sub_Intro_Here>