Lab 1.1. Lab Environment Setup - GCE and VirtualBox VMs

The lab exercises included in this course can be practiced on a system with a minimal hardware profile of up to 2 CPU cores, 1 GB memory, 10 GB disk, and internet connection. We encourage the use of a cloud provider such as Google Cloud Platform (GCP) or Amazon Web Services (AWS) for the provisioning of the lab environment. However, if the cloud is not easily accessible, the lab environment can also be provisioned with a local hypervisor such as VirtualBox. Next we will walk through setting up a Free Tier eligible cloud Virtual Machine with the Google Compute Engine (GCE) cloud service, followed by a guide to setup a local VirtualBox VM.

Google Compute Engine (GCE) VM

When accessing cloud resources, one of our top concerns is the security of our connection. For that reason we will be using a secure shell client together with a set of keys to establish a secure connection between our host system and the remote VM.

Considering a Ubuntu host machine, we will run the **ssh-keygen** tool to generate a set of keys. Our aim is to create a set of keys to allow for a non-root user login to the remote GCE VM. Our user will be named **student**, and the SSH keys will be generated only for this user. This simplifies the remote login process for the remote **student** user, by removing the necessity of typing passwords at the login prompt. Open a terminal on the host machine, and run the following command, and then press **Enter** for all defaults:

```
user@host:~$ ssh-keygen -C student
```

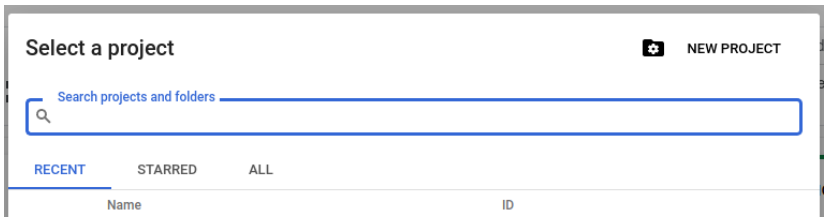
The keys are generated in the the **/home/user/.ssh/** directory:

```
user@host:~$ ls -l /home/user/.ssh/

...
-rw------- 1 user user  2590 Aug  9 17:06 id_rsa
-rw-r--r-- 1 user user   561 Aug  9 17:06 id_rsa.pub
...
```

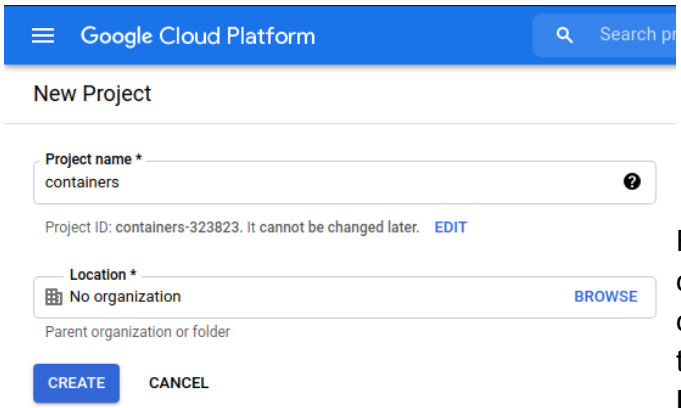
This guide assumes that a Google Cloud Platform (GCP) account has been set up already – although a credit card may be requested during the sign-up process, it should not be charged for the usage of a Free Tier service, as long as the usage limits are not exceeded. The setup is targeting a cloud VM instance which is currently part of Google Cloud Platform's Free Tier offering – resources that are free of charge with a specified monthly usage limit. Please check back periodically to stay up to date with the cloud provider's definition of Free Tier, qualifying services, and usage limits.

NOTE: The following set of screenshots may have a different look on your system, since the cloud services are constantly evolving and features may change without notice.

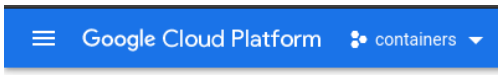
Once logged into the GCP account, from the Console click the project menu at the right of the GCP **Navigation menu**, select **NEW PROJECT**:



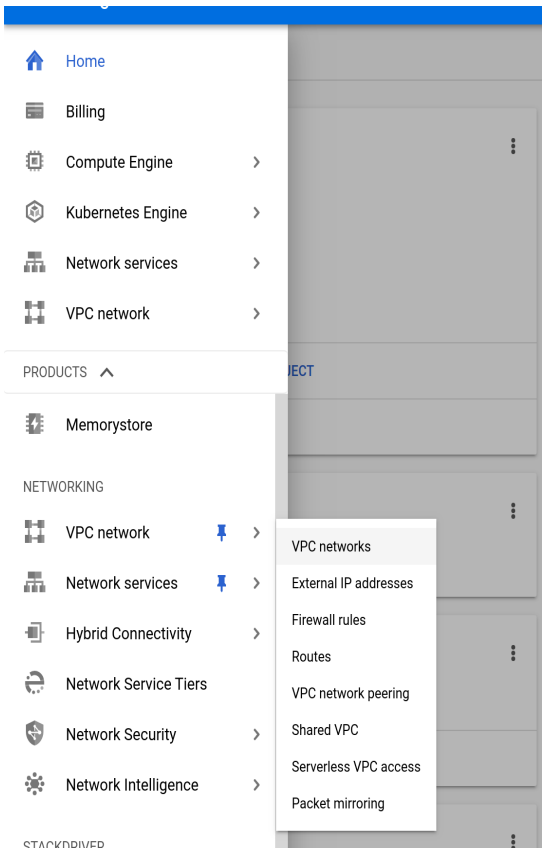
On the new project page provide a new **Project name**, let's name it **containers**, and click **CREATE**:



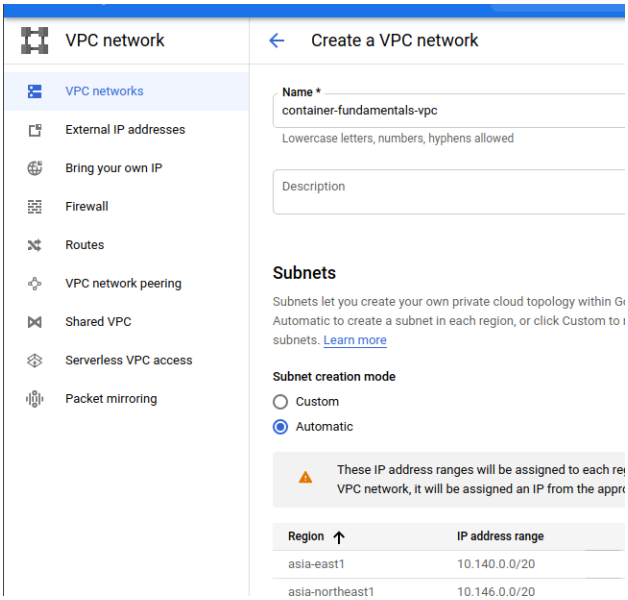
From the **Notifications** pop-up window click **SELECT PROJECT**, and the newly created **containers** project should show as the current project at the right of the GCP **Navigation menu**:



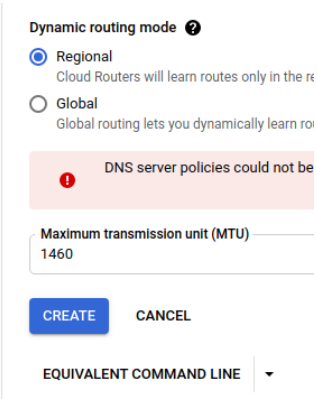
On the GCP Console select the **Navigation menu** from the top left corner.
Scroll down to the **NETWORKING** section, select **VPC network**, then select **VPC networks** from the sub-menu:



If prompted, click **ENABLE**.
On the VPC network screen select **CREATE VPC NETWORK** at the top.
Provide a name: **container-fundamentals-vpc**.
Select **Automatic** Subnet creation mode.

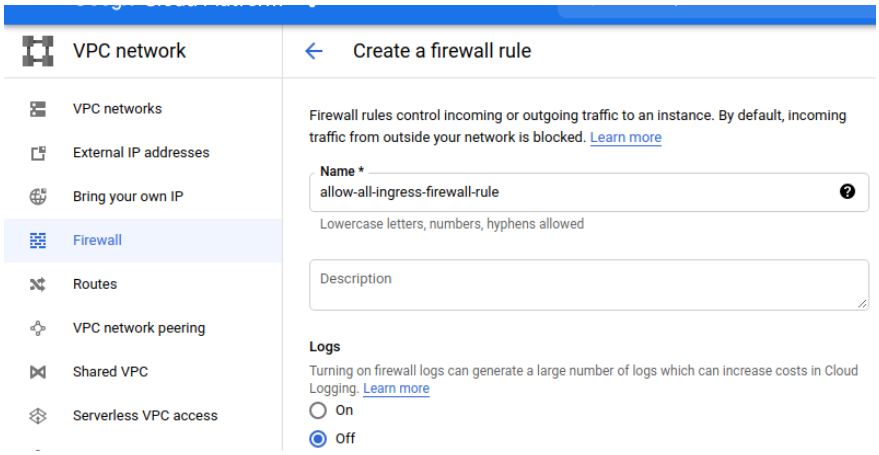


Leave Firewall rules unchecked. We will create a firewall rule later.
Select **CREATE** to finalize the creation of the VPC network.



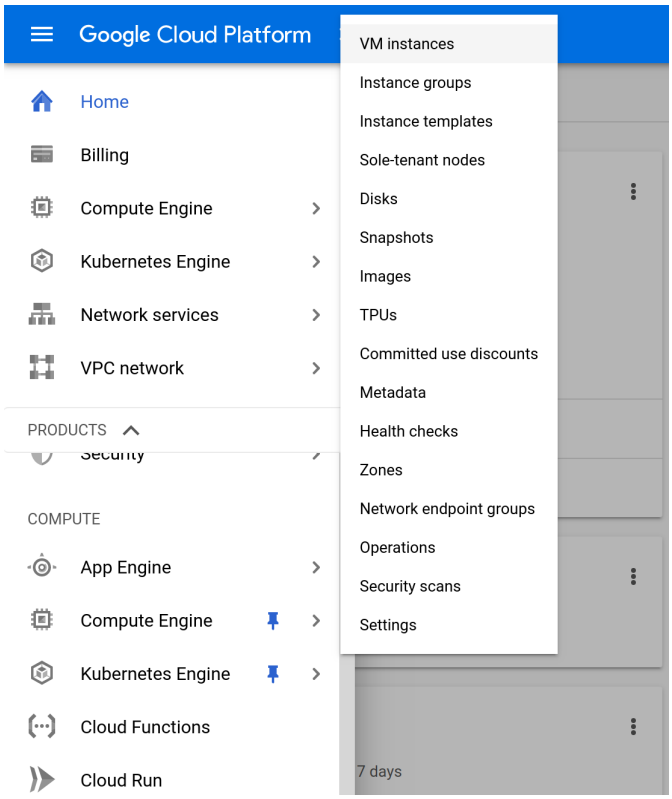
Select from the menu **Firewall**.
On the Firewall screen select **CREATE FIREWALL RULE**.

Provide a name: **allow-all-ingress-firewall-rule**.

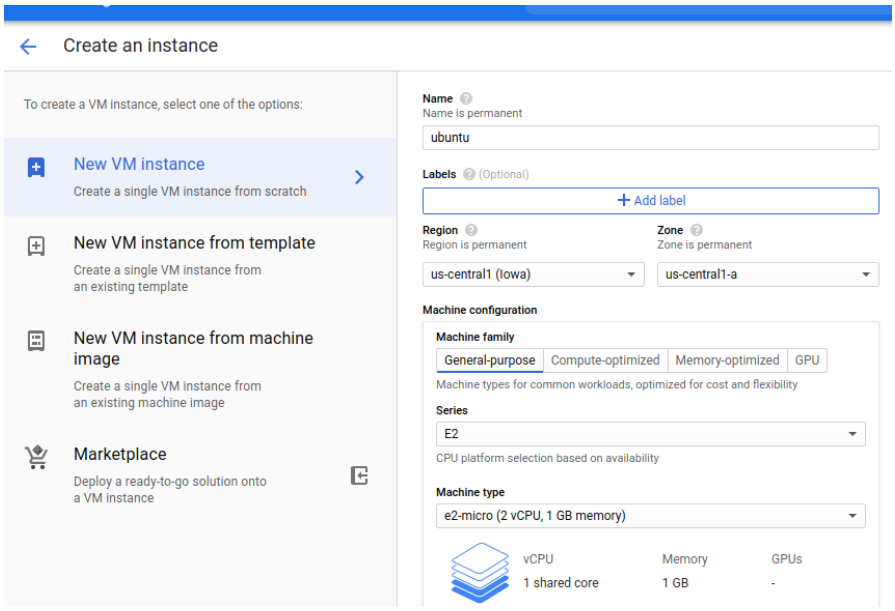


Select from the Network dropdown list the VPC network created earlier **container-fundamentals-vpc**.
Priority: **1000**.
Direction of traffic: **Ingress**.
Action on match: **Allow**.
Targets: **All instances in the network**.
Source IP ranges: **0.0.0.0/0**
Protocols and ports: **Allow all**.
Select **CREATE** to finalize the creation of the Firewall rule.

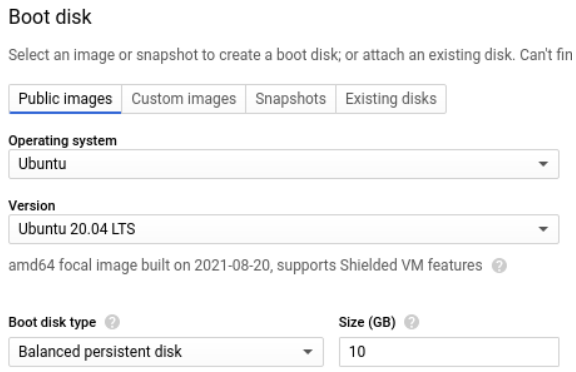
On the GCP Console select the **Navigation menu** from the top left corner.
Scroll down to the **COMPUTE** section, select **Compute Engine**, then select **VM instances** from the sub-menu.



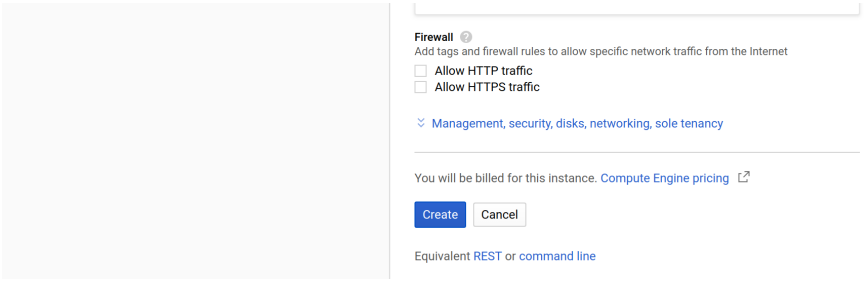
On the VM instances screen select **CREATE INSTANCE**.
Provide a name: **ubuntu**.
Select from the Region dropdown list the one of the regions supporting Free Tier VM instances (Oregon: us-west1, Iowa: us-central1, South Carolina: us-east1).
Select **General-purpose** Machine family.
Select Series **E2**.
Select from the Machine type dropdown list **e2-micro (2 vCPU, 1 GB memory)**.
Select to **Change** the Boot disk to modify the instance OS image.



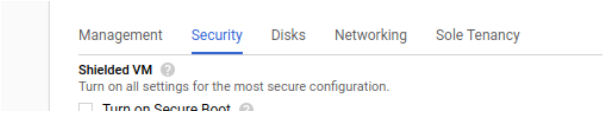
From the Boot disk page select the **Public images** tab, Operating system **Ubuntu**, Version **Ubuntu 20.04 LTS**.
Click **Select**.



Expand the link **Management, security, disks, networking, sole tenancy**.



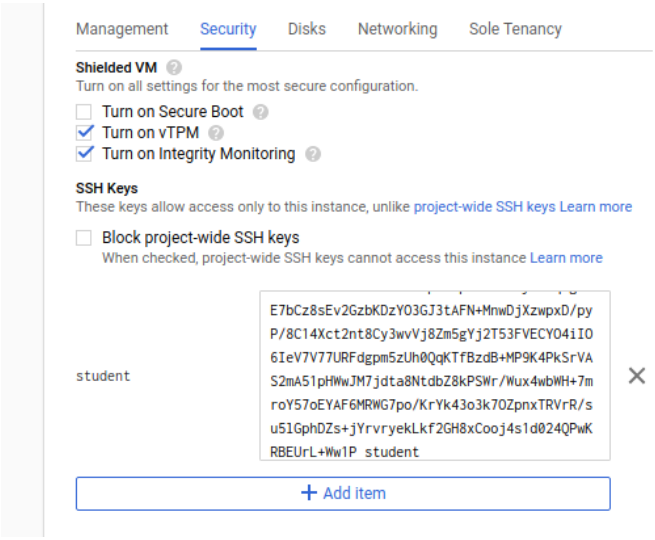
Select the **Security** tab:



On your host machine, locate the key pair generated earlier with the ssh-keygen tool, and visualize the content of the **/home/user/.ssh/id_rsa.pub** public key file:

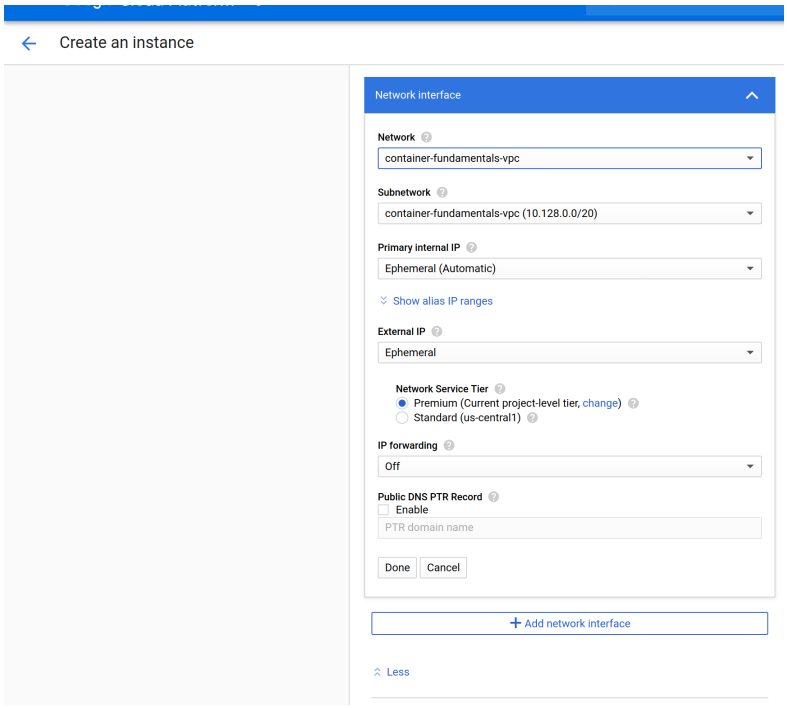
```
user@host:~$ sudo cat /home/user/.ssh/id_rsa.pub

ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDIc94kEU6KCI1b9/
...
KrYk43o3k7OZpnxTRVrR/su5lGphDZs+jYrvryekLkf2GH8xCooj4s1d024QPwKRBEUrL+
Ww1P student
```

Copy the entire content of the file, and paste it into the text box labeled **Enter public SSH key**. The user name **student** will immediately be recognized and extracted by the SSH key management system:

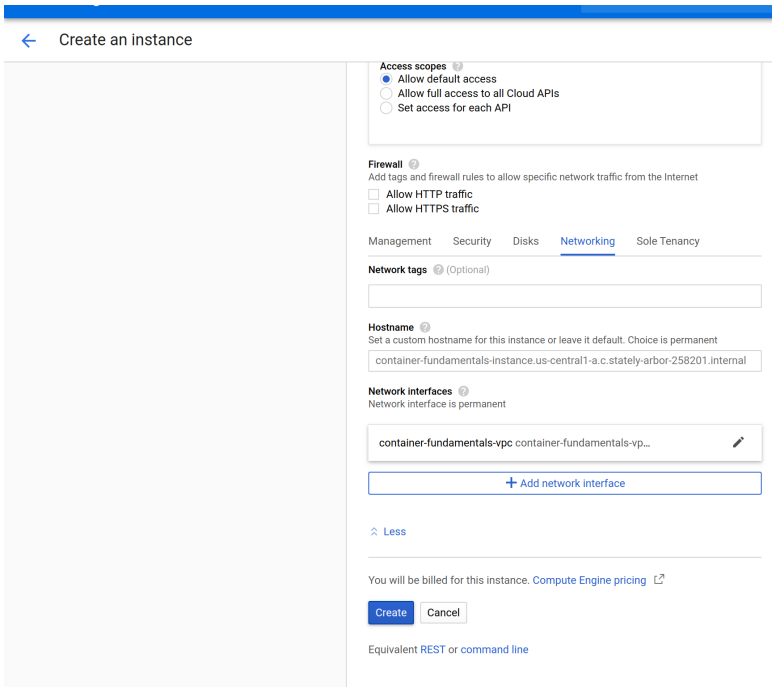


Select the **Networking** tab:

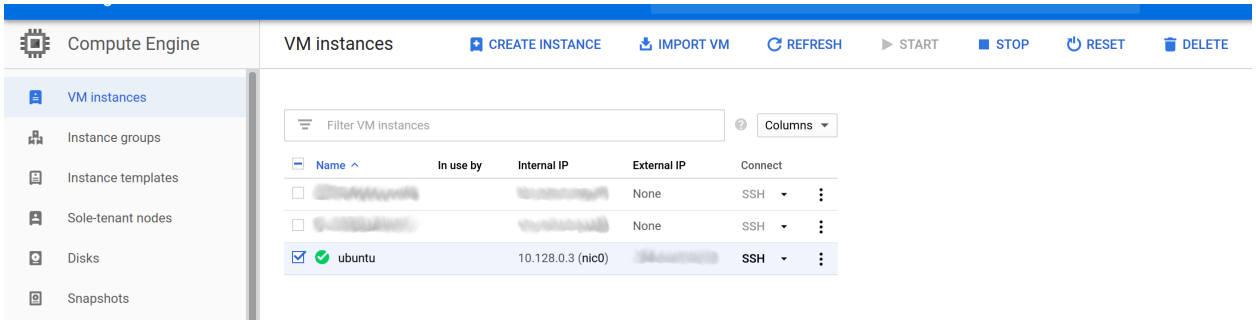
Select to edit the **Network interfaces** field.
From the Network dropdown list select VPC network created earlier and its associated Subnetwork.
Select **Done**.



Select **Create** to finalize the creation of the VM instance.



From the VM instances page we can select the newly created instance followed by the desired action from the top menu: Start, Stop, Reset, Delete, Suspend, Resume.
Running VM instance can be accessed directly by selecting **SSH**, which will open an in-browser terminal:



However, the in-browser terminal may not be as feature rich as our favorite terminal application, therefore we will access our remote VM by running the **ssh** command from our favorite terminal instead. From our desired terminal the **ssh** command will allow us to access the remote VM without the need to type in a password at the login prompt, thanks to the keys we have generated earlier. The External IP will be visible on your VM instances page and must be used together with the desired user **student** in the **ssh** command below:

```
user@host:~$ ssh student@<External-IP>
```

The sample External IP below is for illustration purposes only, and should not be used for lab exercises. Type **yes** when prompted to confirm connection intent:

```
user@host:~$ ssh student@34.120.187.202

The authenticity of host '34.120.187.202 (34.120.187.202)' can't be
established.
ECDSA key fingerprint is
SHA256:U5+A5MXV92hvR4hZGtV1oWvSwYYpM/fhujpAnd3Y6fQ.
Are you sure you want to continue connecting (yes/no/[fingerprint])?

yes

Warning: Permanently added '34.120.187.202' (ECDSA) to the list of
known hosts.
Welcome to Ubuntu 20.04.2 LTS (GNU/Linux 5.8.0-1038-gcp x86_64)
```

```
...
Last login: Tue Aug 10 00:02:33 2021 from ...
```

After a successful login, our prompt from the VM should look like this:

```
student@ubuntu:~$
```

At this point we are ready for lab exercises. Keep in mind that before remotely accessing the GCE **ubuntu** VM, the VM needs to be started from the Console, and needs to remain in a **Running** state throughout the exercise. Once an exercise is completed, the VM can be stopped or shut down, without any loss of work, and then started back up when needed again. Chances are very high that the VM's public IP address will change across VM restarts. Consult the VM management Console to retrieve the latest public IP address of the GCE VM prior to running the **ssh student@<External-IP>** command.

VirtualBox VM

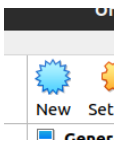
While we encourage learners to explore cloud infrastructure providers to run the lab exercises, access to cloud services may not always be possible in all situations. With that in mind, we are going to present the provisioning of a VM utilizing a popular hypervisor, VirtualBox.

Let's download and install the latest VirtualBox release matching our host operating system. VirtualBox is supported by Linux, OS X, Windows, and Solaris hosts. This installation will assume a native Ubuntu (18.04 LTS or 20.04 LTS) Linux host operating system. Navigate to https://www.virtualbox.org/wiki/Downloads and click on the desired host platform to download and install the desired VirtualBox package. Also, download the VirtualBox Extension Pack, which may be known as "guest additions" – a software pack to be used in a later step. Keep in mind that the versions of the VirtualBox package and extension pack should match:

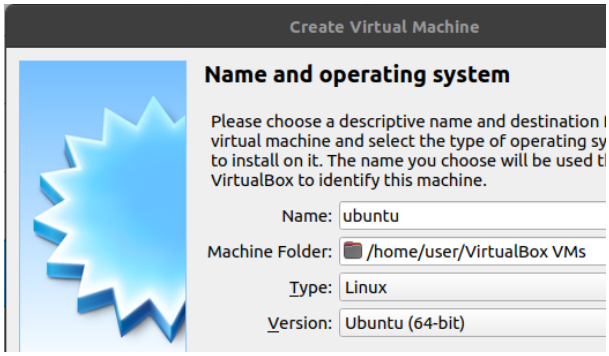
In preparation, we should also download a desired guest operating system image. For the purpose of this course, we are downloading the Ubuntu Server 20.04.2 LTS image file. Navigate to https://ubuntu.com/download/server, select **Option 2 – Manual server installation**, then click on the green download button:

Save the **iso** image file to a desired and easily accessible location on your host system.

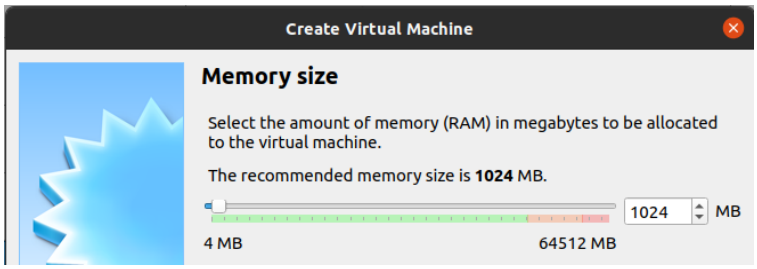
Assuming that we have successfully downloaded and installed VirtualBox, successfully downloaded the VirtualBox guest additions and the Ubuntu iso, let's provision our Virtual Machine. Start VirtualBox, and from the **Oracle VM VirtualBox Manager** interface select **New**, or press simultaneously **CTRL** + **N**, or expand the **Machine** menu and select **New**:



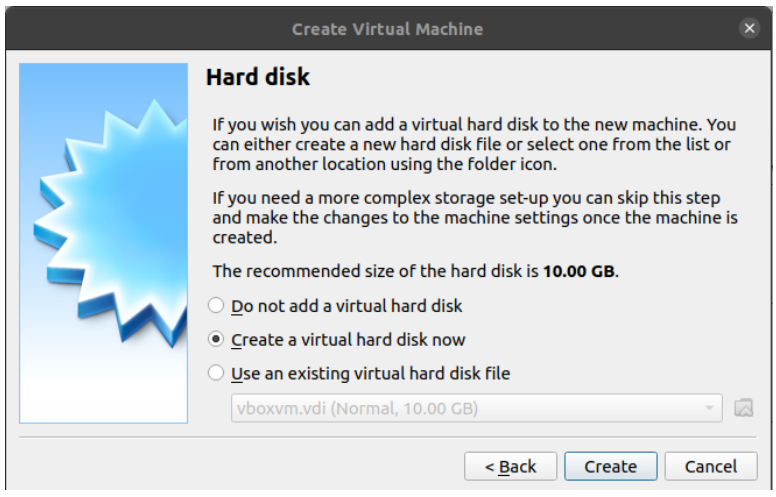
On the **Create Virtual Machine** interface provide the VM Name **ubuntu**, select Type **Linux**, select Version **Ubuntu (64-bit)**, then click **Next**:



On the following step set the Memory size to at least the minimum recommended **1024 MB** and click **Next**:

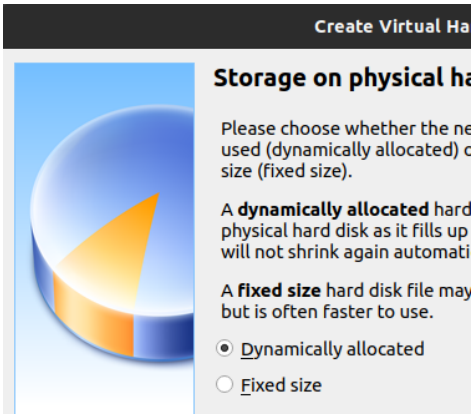


Then select **Create a virtual hard disk now** and click **Create**:

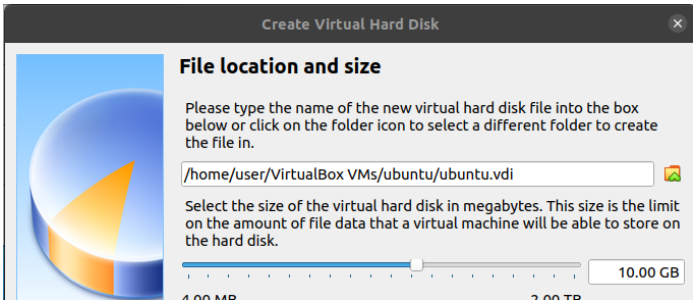


Select the **VDI** virtual hard disk file type, click **Next**:

Select **Dynamically allocated**, click **Next**:

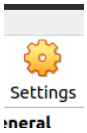


Leave the default **10 GB** size or adjust based your host's available free disk space, then click **Create**:

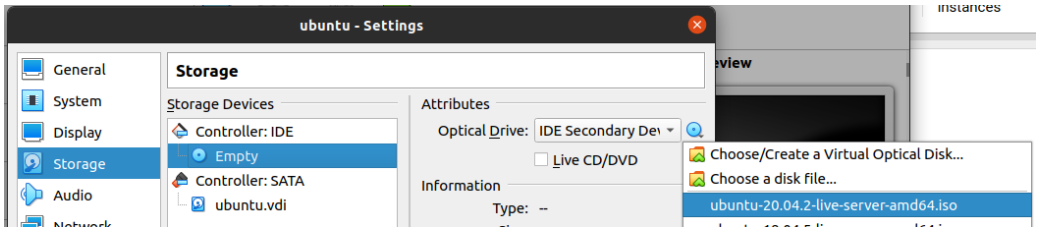


The ubuntu VM has been provisioned, and it is currently powered off.

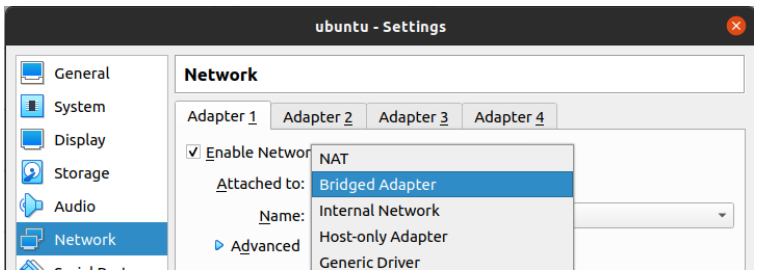
Before attempting to boot the VM, let's make a few changes in its settings:



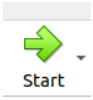
Select the **Storage** category, select the **Empty** IDE storage device, and by clicking on the small disk icon select the desired **iso** image from the drop down, or navigate to the desired **iso** image file with **Choose a disk file...** :



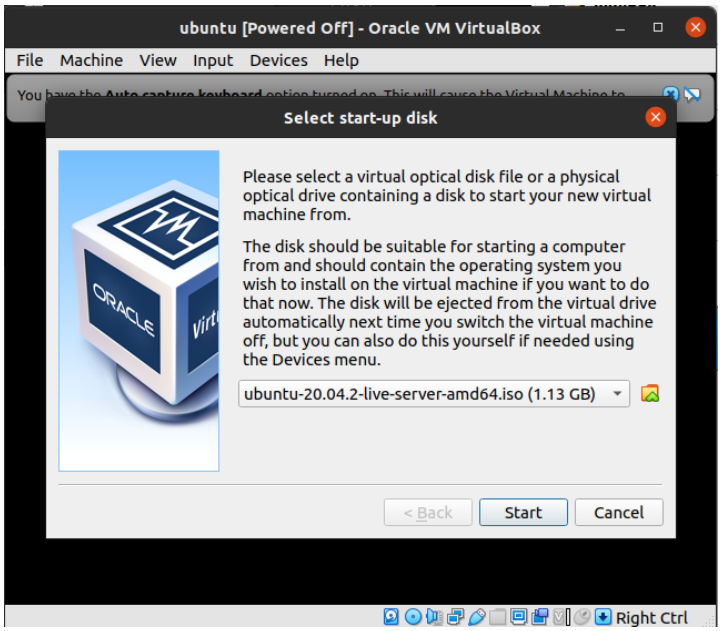
The VM is provisioned by default with a networking device operating in **NAT** mode. Since **NAT** may require additional configuration to support certain aspects of container operations, we will change it to a **Bridged Adapter** mode instead.
Select the **Network** category, then select Attached to: Bridged Adapter, and the Name: to be the physical device that connects your host to the internet (could be an ethernet physical adapter – **ensx / enpx...**, or a wireless lan physical adapter – **wlpxxxx...**):



Now we are ready to start the ubuntu VM:



Verify the desired startup disk is selected, which should be the desired **iso** image file of the **Ubuntu Server 20.04.2 LTS**, then click **Start**:

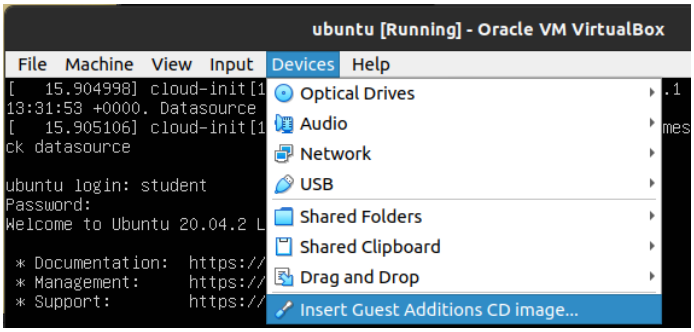


Using your keyboard arrows proceed through the following configuration screens:
Select your desired language then press **Enter**.
For keyboard configuration press **Enter** for default.
For Network connections press **Enter** for defaults.
For Proxy press **Enter** for defaults.
For Ubuntu archive mirror press **Enter** for defaults.
For Guided storage press **Enter** for defaults.
For Storage configuration select **Continue** and press **Enter** for defaults.
For Profile setup: Your name: **student**, Server's name: **ubuntu**, Username: **student**, Password: any password of your choice that is easy for you to remember and use when logging into the VirtualBox ubuntu VM. Use the arrows or TAB to navigate between fields. Select **Done** and press **Enter**.
For SSH Setup select **Install OpenSSH server**, then select **Done** and press **Enter**.

Skip the Featured Server Snaps, just select **Done** and press **Enter**.

After the installation is completed, the **Reboot Now** option will become available, select it and press **Enter**. At the Failed unmounting error message just press **Enter**.
After the reboot is completed, press **Enter** to see the **ubuntu login:** promp. This is where we type the user name **student**, and the password we chose in an earlier step.

There are a few additional steps we need to complete before we can access the local VirtualBox VM from our favorite terminal application. First, we will insert the guest additions image:



Then, at the prompt, we will run the following two commands:

```
student@ubuntu:~$ lsmod | grep -io vboxguest

vboxguest


student@ubuntu:~$ modinfo vboxguest
```

Let's retrieve the private IP address of the ubuntu VM, by running the following command. The IP address displayed can then be used from our favorite terminal running on the host system:

```
student@ubuntu:~$ hostname -I

192.168.0.199
```

On the host system, run the **ssh** command from a terminal, type **yes** and then the student's password when prompted. Keep in mind that the password will remain hidden as it is typed:

```
user@host:~$ ssh student@192.168.0.199

The authenticity of host '192.168.0.199 (192.168.0.199)' can't be
established.
ECDSA key fingerprint is
SHA256:PxHIiogt1ChpOT34F1OZgMSIay/+gr2oG+NzZI6nC9g.
Are you sure you want to continue connecting (yes/no/[fingerprint])?

yes

Warning: Permanently added '192.168.0.199' (ECDSA) to the list of
known hosts.
student@192.168.0.199's password:
Welcome to Ubuntu 20.04.2 LTS (GNU/Linux 5.4.0-81-generic x86_64)
...
Last login: Tue Aug 10 13:35:27 2021
```

One last step is to ensure that the **student** user is able to run sudo on the guest VM without being prompted for a password. Run the following command and type the student's password when prompted, which will remain hidden as it is typed:

```
student@ubuntu:~$ sudo -i

[sudo] password for student:


root@ubuntu:~# cd /etc/sudoers.d/

root@ubuntu:/etc/sudoers.d# ls -la

total 12
drwxr-xr-x  2 root root 4096 Feb  1  2021 ./
drwxr-xr-x 95 root root 4096 Aug 10 18:40 ../
-r--r-----  1 root root  958 Feb  3  2020 README


root@ubuntu:/etc/sudoers.d# touch student

root@ubuntu:/etc/sudoers.d# echo "student ALL=(ALL:ALL) NOPASSWD:ALL"
> student
```

```
root@ubuntu:/etc/sudoers.d# cat student

student ALL=(ALL:ALL) NOPASSWD:ALL


root@ubuntu:/etc/sudoers.d# ls -la

total 16
drwxr-xr-x  2 root root 4096 Aug 10 14:16 .
drwxr-xr-x 95 root root 4096 Aug 10 13:31 ..
-r--r-----  1 root root  958 Feb  3  2020 README
-rw-r--r--  1 root root   35 Aug 10 14:16 student


root@ubuntu:/etc/sudoers.d# chmod 440 student

root@ubuntu:/etc/sudoers.d# ls -la

total 16
drwxr-xr-x  2 root root 4096 Aug 10 14:16 .
drwxr-xr-x 95 root root 4096 Aug 10 13:31 ..
-r--r-----  1 root root  958 Feb  3  2020 README
-r--r-----  1 root root   35 Aug 10 14:16 student
```

At this point we are ready for lab exercises. Keep in mind that before remotely accessing the VirtualBox **ubuntu** VM, the VM needs to be started from the **Oracle VM VirtualBox Manager**, and needs to remain in a **Running** state throughout the exercise. Once an exercise is completed, the VM can be stopped or shut down, without any loss of work, and then started back up when needed again. Chances are very high that the VM's private IP address will be preserved across restarts, and the **ssh student@192.x.x.x** command to remain unchanged. In the case the **ssh** command stops working, revisit earlier steps to log directly into the **ubuntu** VM and retrieve its IP address, and then update the **ssh** command.

Lab 2.1. Cgroups

Cgroups allow users to bundle processes together and limit, account and isolate the group's resources, such as CPU, memory, disk I/O, network, devices, and hugepages.

First, as **root**, let's install a tool that allows us to interact with cgroups:

```
student@ubuntu:~$ sudo apt update

student@ubuntu:~$ sudo apt install -y cgroup-tools

Reading package lists... Done
Building dependency tree
Reading state information... Done
The following package was automatically installed and is no longer required:
  libnuma1
Use 'apt autoremove' to remove it.
The following additional packages will be installed:
  libcgroup1
The following NEW packages will be installed:
  cgroup-tools libcgroup1
0 upgraded, 2 newly installed, 0 to remove and 6 not upgraded.
Need to get 108 kB of archives.
...
```

Now we can list all cgroups on our system:

```
student@ubuntu:~$ sudo lscgroup

rdma:/
net_cls,net_prio:/
hugetlb:/
```

```
memory:/
memory:/user.slice
memory:/system.slice
memory:/system.slice/systemd-update-utmp.service
memory:/system.slice/snap-core-8268.mount
memory:/system.slice/lvm2-monitor.service
...
```

And also list cgroups associated with a process:

```
student@ubuntu:~$ sudo cat /proc/<PID>/cgroup

student@ubuntu:~$ sudo cat /proc/1/cgroup

12:pids:/init.scope
11:rdma:/
10:memory:/init.scope
9:freezer:/
8:hugetlb:/
7:perf_event:/
6:cpuset:/
5:net_cls,net_prio:/
4:devices:/init.scope
3:cpu,cpuacct:/init.scope
2:blkio:/init.scope
1:name=systemd:/init.scope
0::/init.scope
```

Let's explore a particular cgroup, called **freezer**, which allows a group of tasks to be suspended and then resumed. We will demonstrate that a **frozen** (suspended) process does not allow any operations on it until it is **thawed** (resumed). Let's start by creating a new cgroup hierarchy under the freezer cgroup:

```
student@ubuntu:~$ cd /sys/fs/cgroup/freezer/

student@ubuntu:/sys/fs/cgroup/freezer$ sudo mkdir mycgroup

student@ubuntu:/sys/fs/cgroup/freezer$ cd mycgroup/

student@ubuntu:/sys/fs/cgroup/freezer/mycgroup$ ls

cgroup.clone_children  freezer.parent_freezing   freezer.state  tasks
cgroup.procs           freezer.self_freezing     notify_on_release
```

The new directory is populated by default upon its creation. The **tasks** file, initially empty, would otherwise hold PIDs of processes associated with the cgroup. Let's verify the empty file, then create a new process and associate it with our cgroup:

```
student@ubuntu:/sys/fs/cgroup/freezer/mycgroup$ cat tasks
```

Let's open a second terminal as a new **bash** process, and in the new terminal retrieve its PID:

```
student@ubuntu:~$ ps

  PID TTY          TIME CMD
20913 pts/1    00:00:00 bash
20943 pts/1    00:00:00 ps
```

Keep the second terminal running, and return to the first terminal to add the PID of the second terminal to the **tasks** file of the cgroup. (An "Invalid argument" or "No such process" error may be returned if an incorrect PID is used. A "Permission denied" error may be returned if the single quotes (') are misinterpreted after copy/paste, in which case a manual edit is recommended to delete and retype the two single quotes (')):

```
student@ubuntu:/sys/fs/cgroup/freezer/mycgroup$ sudo bash -c 'echo 20913 >>
tasks'

student@ubuntu:/sys/fs/cgroup/freezer/mycgroup$ cat tasks

20913
```

Now we should be able to freeze the processes associated with our cgroup:

```
student@ubuntu:/sys/fs/cgroup/freezer/mycgroup$ sudo bash -c 'echo FROZEN >
freezer.state'

student@ubuntu:/sys/fs/cgroup/freezer/mycgroup$ cat freezer.state

FROZEN
```

The state we just modified affects all the processes listed in the tasks file, that is the second terminal window. Return to the second terminal and try running the **date** command, for example. Nothing will be registered and displayed, as the process of the terminal is in a **frozen** (suspended) state.

Finally, let's **thaw** (resume) the second terminal process:

```
student@ubuntu:/sys/fs/cgroup/freezer/mycgroup$ sudo bash -c 'echo THAWED >
freezer.state'

student@ubuntu:/sys/fs/cgroup/freezer/mycgroup$ cat freezer.state

THAWED
```

Once **thawed** (resumed), the second terminal will display the **date** command we ran earlier, while it was in a **frozen** (suspended) state:

```
student@ubuntu:~$ date

Thu 10 Jun 2021 08:07:31 PM UTC
```

It is now safe to close the second terminal window (the one with the **date** command), and return to student user's home directory in the first terminal window:

```
student@ubuntu:/sys/fs/cgroup/freezer/mycgroup$ cd

student@ubuntu:~$
```

Lab 2.2. Namespaces

Namespaces allow users to isolate mount points, PIDs, networks, IPCs, UTS (hostname) and user IDs. In this exercise, we will focus on the network virtualization feature of namespaces.

Let's list all namespaces for a particular process:

```
student@ubuntu:~$ sudo ls -l /proc/<PID>/ns/

student@ubuntu:~$ sudo ls -l /proc/1/ns/

total 0
lrwxrwxrwx 1 root root 0 Jun 10 10:53 cgroup -> 'cgroup:[4026531835]'
lrwxrwxrwx 1 root root 0 Jun 10 10:53 ipc -> 'ipc:[4026531839]'
lrwxrwxrwx 1 root root 0 Jun 10 10:53 mnt -> 'mnt:[4026531840]'
lrwxrwxrwx 1 root root 0 Jun 10 10:53 net -> 'net:[4026531992]'
lrwxrwxrwx 1 root root 0 Jun 10 10:53 pid -> 'pid:[4026531836]'
lrwxrwxrwx 1 root root 0 Jun 10 10:53 pid_for_children -> 'pid:[4026531836]'
lrwxrwxrwx 1 root root 0 Jun 10 10:53 user -> 'user:[4026531837]'
lrwxrwxrwx 1 root root 0 Jun 10 10:53 uts -> 'uts:[4026531838]'
```

Let's explore the network namespaces, by creating two separate namespaces enabled to communicate with each other through a virtual ethernet tunnel.

First, create two namespaces:

```
student@ubuntu:~$ sudo ip netns add namespace1

student@ubuntu:~$ sudo ip netns add namespace2

student@ubuntu:~$ ip netns list

namespace2
```

```
namespace1
```

Then create a pair of interconnected virtual ethernet devices:

```
student@ubuntu:~$ sudo ip link add veth1 type veth peer name veth2
```

Link each device to a namespace respectively:

```
student@ubuntu:~$ sudo ip link set veth1 netns namespace1

student@ubuntu:~$ sudo ip link set veth2 netns namespace2
```

Bring up the devices while assigning them IP addresses:

```
student@ubuntu:~$ sudo ip netns exec namespace1 ip link set dev veth1 up

student@ubuntu:~$ sudo ip netns exec namespace2 ip link set dev veth2 up

student@ubuntu:~$ sudo ip netns exec namespace1 ip addr add 192.168.1.1/24 dev
veth1

student@ubuntu:~$ sudo ip netns exec namespace2 ip addr add 192.168.1.2/24 dev
veth2
```

And now verify the connectivity between the two namespaces as it is enabled by the veth pair tunnel. From the first namespace we should be able to ping the second one, and from the second namespace we should be able to ping the first one:

```
student@ubuntu:~$ sudo ip netns exec namespace1 ping -c5 192.168.1.2

PING 192.168.1.2 (192.168.1.2) 56(84) bytes of data.
64 bytes from 192.168.1.2: icmp_seq=1 ttl=64 time=0.044 ms
64 bytes from 192.168.1.2: icmp_seq=2 ttl=64 time=0.038 ms
64 bytes from 192.168.1.2: icmp_seq=3 ttl=64 time=0.037 ms
64 bytes from 192.168.1.2: icmp_seq=4 ttl=64 time=0.039 ms
64 bytes from 192.168.1.2: icmp_seq=5 ttl=64 time=0.041 ms

--- 192.168.1.2 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4089ms
rtt min/avg/max/mdev = 0.037/0.039/0.044/0.008 ms


student@ubuntu:~$ sudo ip netns exec namespace2 ping -c5 192.168.1.1
```

```
PING 192.168.1.1 (192.168.1.1) 56(84) bytes of data.
64 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=0.026 ms
64 bytes from 192.168.1.1: icmp_seq=2 ttl=64 time=0.046 ms
64 bytes from 192.168.1.1: icmp_seq=3 ttl=64 time=0.045 ms
64 bytes from 192.168.1.1: icmp_seq=4 ttl=64 time=0.045 ms
64 bytes from 192.168.1.1: icmp_seq=5 ttl=64 time=0.042 ms

--- 192.168.1.1 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4073ms
rtt min/avg/max/mdev = 0.026/0.040/0.046/0.011 ms
```

After a successful verification, we may delete the two namespaces:

```
student@ubuntu:~$ sudo ip netns delete namespace1

student@ubuntu:~$ sudo ip netns delete namespace2

student@ubuntu:~$ ip netns list
```

Lab 2.3. UnionFS

UnionFS transparently overlays files and directories of separate filesystems, to create a unified seamless filesystem. Each participant directory is referred to as a branch and we may set priorities and access modes while mounting branches.

Let's first install the required tool:

```
student@ubuntu:~$ sudo apt update

student@ubuntu:~$ sudo apt install -y unionfs-fuse

Reading package lists... Done
Building dependency tree
Reading state information... Done
The following package was automatically installed and is no longer required:
  libnuma1
Use 'apt autoremove' to remove it.
The following NEW packages will be installed:
  unionfs-fuse
0 upgraded, 1 newly installed, 0 to remove and 6 not upgraded.
Need to get 48.7 kB of archives.
After this operation, 146 kB of additional disk space will be used.
Get:1 http://us-central1.gce.archive.ubuntu.com/ubuntu focal/universe amd64
unionfs-fuse amd64 1.0-1ubuntu2 [48.7 kB]
Fetched 48.7 kB in 0s (302 kB/s)
Selecting previously unselected package unionfs-fuse.
(Reading database ... 63255 files and directories currently installed.)
Preparing to unpack .../unionfs-fuse_1.0-1ubuntu2_amd64.deb ...
Unpacking unionfs-fuse (1.0-1ubuntu2) ...
Setting up unionfs-fuse (1.0-1ubuntu2) ...
Processing triggers for man-db (2.9.1-1) ...
```

Let's create two separate directories (branches) with two files each respectively:

```
student@ubuntu:~$ mkdir dir1

student@ubuntu:~$ touch dir1/f1

student@ubuntu:~$ touch dir1/f2

student@ubuntu:~$ mkdir dir2

student@ubuntu:~$ touch dir2/f3

student@ubuntu:~$ touch dir2/f4
```

Let's create an empty directory, where the union filesystem will be mounted:

```
student@ubuntu:~$ mkdir union
```

Let's mount the two branches and verify the transparent overlay by listing all the files in the union:

```
student@ubuntu:~$ unionfs dir1/:dir2/ union/

student@ubuntu:~$ ls union/

f1  f2  f3  f4
```

Lab 3.1. Chroot

In this exercise we will explore chroot's ability to change the apparent root directory for a process. The setup will require an installation of a Debian-based guest system in a subdirectory of our Ubuntu host system. Although similar to the installation of a guest Operating System Virtual Machine, this scenario is missing the hypervisor component, allowing the guest OS to be installed directly on top of the host OS, while allowing the host kernel to manage the guest as well.

The tool required to install the Debian based guest on an existing running host is debootstrap. It installs a Debian based guest in a subdirectory of our Ubuntu system. Let's install debootstrap:

```
student@ubuntu:~$ sudo apt update

student@ubuntu:~$ sudo apt install -y debootstrap

Reading package lists... Done
Building dependency tree
Reading state information... Done
The following package was automatically installed and is no longer required:
  libnuma1
Use 'apt autoremove' to remove it.
Suggested packages:
  ubuntu-archive-keyring
The following NEW packages will be installed:
  debootstrap
0 upgraded, 1 newly installed, 0 to remove and 6 not upgraded.
Need to get 39.3 kB of archives.
After this operation, 301 kB of additional disk space will be used.
Get:1 http://us-central1.gce.archive.ubuntu.com/ubuntu focal-updates/main
amd64 debootstrap all 1.0.118ubuntu1.4 [39.3 kB]
...
```

Now let's set up a subdirectory on our host system for the guest:

```
student@ubuntu:~$ sudo mkdir /mnt/chroot-ubuntu-xenial
```

Install an Ubuntu Xenial guest suite in the target subdirectory created earlier from the archives mirror:

```
student@ubuntu:~$ sudo debootstrap <suite> <target> <mirror>

student@ubuntu:~$ sudo debootstrap xenial /mnt/chroot-ubuntu-xenial/
http://archive.ubuntu.com/ubuntu/

I: Retrieving InRelease
I: Checking Release signature
I: Valid Release signature (key id 790BC7277767219C42C86F933B4FE6ACC0B21F32)
I: Retrieving Packages
I: Validating Packages
I: Resolving dependencies of required packages...
I: Resolving dependencies of base packages...
I: Checking component main on http://archive.ubuntu.com/ubuntu...
I: Retrieving adduser 3.113+nmu3ubuntu4
I: Validating adduser 3.113+nmu3ubuntu4
I: Retrieving apt 1.2.10ubuntu1
I: Validating apt 1.2.10ubuntu1
I: Retrieving apt-utils 1.2.10ubuntu1
I: Validating apt-utils 1.2.10ubuntu1
I: Retrieving base-files 9.4ubuntu4
I: Validating base-files 9.4ubuntu4
...
```

Before verifying the guest installation with chroot, let's verify the version of our Ubuntu host OS:

```
student@ubuntu:~$ cat /etc/os-release

NAME="Ubuntu"
VERSION="20.04.2 LTS (Focal Fossa)"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 20.04.2 LTS"
VERSION_ID="20.04"
HOME_URL="https://www.ubuntu.com/"
SUPPORT_URL="https://help.ubuntu.com/"
BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/"
PRIVACY_POLICY_URL="https://www.ubuntu.com/legal/terms-and-policies/privacy-po
licy"
VERSION_CODENAME=focal
UBUNTU_CODENAME=focal
```

Now, with chroot, let's open a shell into the newly installed guest OS environment. Every subsequent command will run inside the chrooted environment as the new root of /mnt/chroot-ubuntu-xenial:

```
student@ubuntu:~$ sudo chroot /mnt/chroot-ubuntu-xenial/ /bin/bash

root@ubuntu:/# cat /etc/os-release

NAME="Ubuntu"
VERSION="16.04 LTS (Xenial Xerus)"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 16.04 LTS"
VERSION_ID="16.04"
HOME_URL="http://www.ubuntu.com/"
SUPPORT_URL="http://help.ubuntu.com/"
BUG_REPORT_URL="http://bugs.launchpad.net/ubuntu/"
UBUNTU_CODENAME=xenial
```

We can safely exit the chrooted environment and return to the **student** user on the host system:

```
root@ubuntu:/# exit

exit


student@ubuntu:~$
```

Lab 3.2. LXC

Lxc is an interface for Linux kernel OS level virtualization features. It allows the creation of linux containers, both unprivileged and privileged. Let's explore both aspects in this exercise.

Let's install it first:

```
student@ubuntu:~$ sudo apt update

student@ubuntu:~$ sudo apt install -y lxc

Reading package lists... Done
Building dependency tree
Reading state information... Done
The following package was automatically installed and is no longer required:
  libnuma1
Use 'sudo apt autoremove' to remove it.
The following additional packages will be installed:
  bridge-utils dns-root-data dnsmasq-base libidn11 liblxc-common liblxc1
libpam-cgfs lxc-utils lxcfs
  uidmap
Suggested packages:
  ifupdown btrfs-tools lxc-templates lxctl
The following NEW packages will be installed:
  bridge-utils dns-root-data dnsmasq-base libidn11 liblxc-common liblxc1
libpam-cgfs lxc lxc-utils
  lxcfs uidmap
0 upgraded, 11 newly installed, 0 to remove and 7 not upgraded.
...
```

Create an unprivileged container as the **student** user

An unprivileged container is the safest container to deploy in any environment. The UID of the container's root is mapped to a UID with few or no privileges on the host system. While an unprivileged container can be created by **root** also, we will explore the lxc configuration and the container creation steps as the **student** user. Prior to creating an unprivileged container, we have to set the configuration for UID and GID mapping, and a network device quota that allows the unprivileged user to create network devices on the host - which otherwise is not allowed by default.

Let's display the UID and GID map defined for the student user on the host (the values associated with the student user will be used later to customize the configuration):

```
student@ubuntu:~$ cat /etc/subuid

ubuntu:165536:65536
student:231072:65536

student@ubuntu:~$ cat /etc/subgid

ubuntu:165536:65536
student:231072:65536
```

Let's add the student user to a configuration file which allows the user to create network devices on the host, to be then used by the linux containers:

```
student@ubuntu:~$ sudo bash -c 'echo student veth lxcbr0 10  >>
/etc/lxc/lxc-usernet'

student@ubuntu:~$ cat /etc/lxc/lxc-usernet

# USERNAME TYPE BRIDGE COUNT
student veth lxcbr0 10
```

Let's set up the configuration file for lxc. First verify that it is not already setup:

```
student@ubuntu:~$ ls -a ~/ | grep config
```

Then continue by creating the necessary directories, copy over the default configuration file, ensure it is read-write enabled, and display its default content:

```
student@ubuntu:~$ mkdir -p ~/.config/lxc
```

```
student@ubuntu:~$ ls -a ~/.config/

.  ..  lxc

student@ubuntu:~$ cp /etc/lxc/default.conf ~/.config/lxc/default.conf

student@ubuntu:~$ chmod 664 ~/.config/lxc/default.conf

student@ubuntu:~$ cat ~/.config/lxc/default.conf

lxc.net.0.type = veth
lxc.net.0.link = lxcbr0
lxc.net.0.flags = up
lxc.net.0.hwaddr = 00:16:3e:xx:xx:xx
```

Now let's customize the configuration file with the UID and GID maps of the student user, extracted earlier from the **/etc/subuid** and **/etc/subgid** files:

```
student@ubuntu:~$ echo lxc.idmap = u 0 231072 65536 >>
~/.config/lxc/default.conf

student@ubuntu:~$ echo lxc.idmap = g 0 231072 65536 >>
~/.config/lxc/default.conf

student@ubuntu:~$ cat ~/.config/lxc/default.conf

lxc.net.0.type = veth
lxc.net.0.link = lxcbr0
lxc.net.0.flags = up
lxc.net.0.hwaddr = 00:16:3e:xx:xx:xx
lxc.idmap = u 0 231072 65536
lxc.idmap = g 0 231072 65536
```

~~Reboot your host, or log out and then log back in:~~

```
student@ubuntu:~$ sudo reboot
```

In order to prevent a possible permission error, we need to set an access control list on our **~/.local** directory. Using the same UID value extracted earlier from the **/etc/subuid** file, run the following command:

```
student@ubuntu:~$ setfacl -R -m u:231072:x ~/.local
```

At this point, we are ready to create an unprivileged container. We will use the **download** template which will present us a list of all available images designed to work without privileges. Once the image index is displayed, the tool will pause and expect at the prompt three separate entries from the user at the CLI: **distribution**, **release** and **architecture**. For this example **ubuntu**, **xenial** and **amd64** have been entered respectively at the prompts:

NOTE: If an ERROR is caused by the GPG key fetch step, run the below command with an additional option **`--keyserver keyserver.ubuntu.com`**.

```
student@ubuntu:~$ lxc-create --template download --name unpriv-cont-user

Setting up the GPG keyring
Downloading the image index

---
DIST  RELEASE     ARCH  VARIANT     BUILD
---
almalinux   8     amd64 default     20210608_22:59
almalinux   8     arm64 default     20210608_22:35
alpine3.10  amd64 default     20210610_13:00
alpine3.10  arm64 default     20210610_13:00
alpine3.10  armhf default     20210610_13:00
alpine3.10  i386  default     20210610_13:00
alpine3.10  ppc64el     default     20210610_13:00
alpine3.10  s390x default     20210610_13:00
alpine3.11  amd64 default     20210610_13:00
...
ubuntutrustyamd64 default     20210610_07:42
ubuntutrustyarm64 default     20210610_07:42
ubuntutrustyarmhf default     20210610_07:44
ubuntutrustyi386  default     20210610_07:42
ubuntutrustyppc64el     default     20210610_07:42
ubuntuxenialamd64 default     20210610_07:42
ubuntuxenialarm64 default     20210610_07:42
ubuntuxenialarmhf default     20210610_07:46
ubuntuxenialamd64 default     20210610_07:42
ubuntuxenialppc64el     default     20210610_07:42
ubuntuxenials390x default     20210610_07:42
voidlinux   current     amd64 default     20210610_17:10
voidlinux   current     arm64 default     20210610_17:10
voidlinux   current     armhf default     20210610_17:10
voidlinux   current     i386  default     20210610_17:10
---

Distribution:
ubuntu      ## Type ubuntu and press ENTER
Release:
```

```
xenial      ## Type xenial and press ENTER
Architecture:
amd64             ## Type amd64 and press ENTER

Downloading the image index
Downloading the rootfs
Downloading the metadata
The image cache is now ready
Unpacking the rootfs

---
You just created an Ubuntu xenial amd64 (20210610_07:42) container.

To enable SSH, run: apt install openssh-server
No default root or user password are set by LXC.
```

NOTE: If we already know the desired **distribution**, **release** and **architecture** values, we can simply run the following command (pay close attention to the extra set of **double hyphens (--)** separating the last three options from the original command):

```
student@ubuntu:~$ lxc-create --template download --name unpriv-cont-user --
--dist ubuntu --release xenial --arch amd64
```

With the container created, now we can start it:

```
student@ubuntu:~$ lxc-start -n unpriv-cont-user -d
```

We can list containers, and also display individual container details:

```
student@ubuntu:~$ lxc-ls -f

NAME             STATE   AUTOSTART GROUPS IPV4       IPV6 UNPRIVILEGED
unpriv-cont-user RUNNING 0         -      10.0.3.14 -    true


student@ubuntu:~$ lxc-info -n unpriv-cont-user

Name:           unpriv-cont-user
State:          RUNNING
PID:            22024
IP:             10.0.3.14
Memory use:     28.84 MiB
KMem use:       3.80 MiB
Link:           veth1001_no5r
```

```
 TX bytes:      1.75 KiB
 RX bytes:      2.65 KiB
 Total bytes:   4.40 KiB
```

We can log into the running container and interact with its environment. Let's display the container hostname, list of processes, and OS release, then exit the container:

```
student@ubuntu:~$ lxc-attach -n unpriv-cont-user

root@unpriv-cont-user:/# hostname

unpriv-cont-user


root@unpriv-cont-user:/# ps

  PID TTY          TIME CMD
  176 pts/2    00:00:00 agetty
  350 pts/2    00:00:00 bash
  354 pts/2    00:00:00 ps


root@unpriv-cont-user:/# cat /etc/os-release

NAME="Ubuntu"
VERSION="16.04.7 LTS (Xenial Xerus)"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 16.04.7 LTS"
VERSION_ID="16.04"
HOME_URL="http://www.ubuntu.com/"
SUPPORT_URL="http://help.ubuntu.com/"
BUG_REPORT_URL="http://bugs.launchpad.net/ubuntu/"
VERSION_CODENAME=xenial
UBUNTU_CODENAME=xenial


root@unpriv-cont-user:/# exit

exit


student@ubuntu:~$
```

Stopping and removing a container is quite simple:

```
student@ubuntu:~$ lxc-stop -n unpriv-cont-user

student@ubuntu:~$ lxc-destroy -n unpriv-cont-user

student@ubuntu:~$ lxc-ls -f
```

Create a privileged container (as **root**)

In the situation when a privileged container has to be created, the process is similar to an unprivileged container, with the only distinction that it is created by the **root** of the host. Again, we will be required to provide at the CLI prompt the desired distribution, release and architecture from the available image index:

```
student@ubuntu:~$ sudo lxc-create --template download --name priv-cont

Setting up the GPG keyring
Downloading the image index

---
DIST  RELEASE     ARCH  VARIANT     BUILD
---
almalinux   8     amd64 default     20210608_22:59
almalinux   8     arm64 default     20210608_22:35
alpine3.10  amd64 default     20210610_13:00
alpine3.10  arm64 default     20210610_13:00
alpine3.10  armhf default     20210610_13:00
alpine3.10  i386  default     20210610_13:00
alpine3.10  ppc64el     default     20210610_13:00
alpine3.10  s390x default     20210610_13:00
alpine3.11  amd64 default     20210610_13:00
...
ubuntutrustyamd64 default     20210610_07:42
ubuntutrustyarm64 default     20210610_07:42
ubuntutrustyarmhf default     20210610_07:44
ubuntutrustyi386  default     20210610_07:42
ubuntutrustyppc64el     default     20210610_07:42
ubuntuxenialamd64 default     20210610_07:42
ubuntuxenialarm64 default     20210610_07:42
ubuntuxenialarmhf default     20210610_07:46
ubuntuxenialppc64el     default     20210610_07:42
ubuntuxenials390x default     20210610_07:42
```

```
voidlinux   current     amd64 default     20210610_17:10
voidlinux   current     arm64 default     20210610_17:10
voidlinux   current     armhf default     20210610_17:10
voidlinux   current     i386  default     20210610_17:10
---

Distribution:
ubuntu      ## Type ubuntu and press ENTER
Release:
xenial      ## Type xenial and press ENTER
Architecture:
amd64             ## Type amd64 and press ENTER

Downloading the image index
Downloading the rootfs
Downloading the metadata
The image cache is now ready
Unpacking the rootfs

---
You just created an Ubuntu xenial amd64 (20210610_07:42) container.

To enable SSH, run: apt install openssh-server
No default root or user password are set by LXC.
```

NOTE: If we already know the desired **distribution**, **release** and **architecture** values, we can simply run the following command (pay close attention to the extra set of **double hyphens (--)** separating the last three options from the original command):

```
student@ubuntu:~$ sudo lxc-create --template download --name priv-cont --
--dist ubuntu --release xenial --arch amd64
```

With the container created, now we can start it:

```
student@ubuntu:~$ sudo lxc-start -n priv-cont -d
```

We can list containers, and also display individual container details:

```
student@ubuntu:~$ sudo lxc-ls -f

NAME      STATE   AUTOSTART GROUPS IPV4       IPV6 UNPRIVILEGED
priv-cont RUNNING 0         -      10.0.3.49 -    false
```

```
student@ubuntu:~$ sudo lxc-info -n priv-cont

Name:           priv-cont
State:          RUNNING
PID:            32925
IP:             10.0.3.49
CPU use:        0.31 seconds
BlkIO use:      17.14 MiB
Memory use:     29.35 MiB
KMem use:       3.49 MiB
Link:           vethujdQXr
 TX bytes:      1.35 KiB
 RX bytes:      1.75 KiB
 Total bytes:   3.10 KiB
```

We can log into the running container and interact with its environment. Let's display the container hostname, list of processes, and OS release, then exit the container:

```
student@ubuntu:~$ sudo lxc-attach -n priv-cont

root@priv-cont:~# hostname

priv-cont


root@priv-cont:~# ps

  PID TTY          TIME CMD
  356 pts/2    00:00:00 agetty
  366 pts/2    00:00:00 bash
  378 pts/2    00:00:00 ps


root@priv-cont:~# cat /etc/os-release

NAME="Ubuntu"
VERSION="16.04.7 LTS (Xenial Xerus)"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 16.04.7 LTS"
VERSION_ID="16.04"
HOME_URL="http://www.ubuntu.com/"
SUPPORT_URL="http://help.ubuntu.com/"
BUG_REPORT_URL="http://bugs.launchpad.net/ubuntu/"
VERSION_CODENAME=xenial
UBUNTU_CODENAME=xenial
```

```
root@priv-cont:~# exit

exit


student@ubuntu:~$
```

Stopping and removing a container is quite simple:

```
student@ubuntu:~$ sudo lxc-stop -n priv-cont

student@ubuntu:~$ sudo lxc-destroy -n priv-cont

student@ubuntu:~$ sudo lxc-ls -f
```

Lab 3.3. Systemd-nspawn

Another method of creating an isolated virtual environment where an application or an entire Operating System could run is by creating a systemd-container. Subsequently, this container can be managed with systemd tools.

Let's install the container tool:

```
student@ubuntu:~$ sudo apt update

student@ubuntu:~$ sudo apt install -y systemd-container

Reading package lists... Done
Building dependency tree
Reading state information... Done
The following package was automatically installed and is no longer required:
  libnuma1
Use 'sudo apt autoremove' to remove it.
The following additional packages will be installed:
  libnss-mymachines
The following NEW packages will be installed:
  libnss-mymachines systemd-container
0 upgraded, 2 newly installed, 0 to remove and 4 not upgraded.
Need to get 449 kB of archives.
...
```

Let's bootstrap a Debian base system in a target directory of the host system, more precisely in **/home/student/DebianContainer**. Debootstrap can only run as root:

```
student@ubuntu:~$ sudo debootstrap --arch=amd64 stable ~/DebianContainer
```

```
I: Keyring file not available at
/usr/share/keyrings/debian-archive-keyring.gpg; switching to https mirror
https://deb.debian.org/debian
I: Retrieving InRelease
I: Retrieving Packages
I: Validating Packages
I: Resolving dependencies of required packages...
I: Resolving dependencies of base packages...
I: Checking component main on https://deb.debian.org/debian...
...
```

Now we can safely create a container from the directory where the Debian base system is running. Again, we need to be root. With the container running, we can display its OS release file for validation:

```
student@ubuntu:~$ sudo systemd-nspawn -D ~/DebianContainer

Spawning container DebianContainer on /home/student/DebianContainer.
Press ^] three times within 1s to kill container.


root@DebianContainer:~# cat /etc/os-release

PRETTY_NAME="Debian GNU/Linux 10 (buster)"
NAME="Debian GNU/Linux"
VERSION_ID="10"
VERSION="10 (buster)"
VERSION_CODENAME=buster
ID=debian
HOME_URL="https://www.debian.org/"
SUPPORT_URL="https://www.debian.org/support"
BUG_REPORT_URL="https://bugs.debian.org/"
root@DebianContainer:~#
```

Leave the current terminal open with the running DebianContainer container, and open a second terminal on the same host VM. From the second terminal let's list containers and display container details:

```
student@ubuntu:~$ sudo machinectl list

MACHINE          CLASS     SERVICE         OS     VERSION ADDRESSES
DebianContainer container systemd-nspawn debian 10      -

1 machines listed.


student@ubuntu:~$ sudo machinectl status DebianContainer
```

```
DebianContainer(d9a7d21805bd7a14e3147d63f43d9b9f)
           Since: Fri 2021-06-11 19:43:15 UTC; 13min ago
          Leader: 156878 (bash)
         Service: systemd-nspawn; class container
            Root: /home/student/DebianContainer
              OS: Debian GNU/Linux 10 (buster)
            Unit: machine-DebianContainer.scope
                  └─payload
                    └─156878 -bash

Jun 11 19:43:15 ubuntu systemd[1]: Started Container DebianContainer.
```

```
student@ubuntu:~$ sudo machinectl show DebianContainer

Name=DebianContainer
Id=d9a7d21805bd7a14e3147d63f43d9b9f
Timestamp=Fri 2021-06-11 19:43:15 UTC
TimestampMonotonic=161082896669
Service=systemd-nspawn
Unit=machine-DebianContainer.scope
Leader=156878
Class=container
RootDirectory=/home/student/DebianContainer
State=running
```

The container can be terminated from within or from outside (from the second terminal). From within the container we may run **logout** or **exit** command:

```
root@DebianContainer:~# exit

logout
Container DebianContainer exited successfully.
student@ubuntu:~$
```

While from the second terminal we may run the following command:

```
student@ubuntu:~$ sudo machinectl terminate DebianContainer
```

Then we validate that the container has been terminated:

```
student@ubuntu:~$ sudo machinectl list

No machines.
```

Lab 4.1. Install runc

While runc was not designed to be used directly by end-users, it is used by more complex container tools, such as Docker or Podman. There are various methods of installing the runc CLI tool. For Ubuntu it is available to install directly from the Ubuntu software package repository. So let's proceed with the installation of runc on Ubuntu. First, update the system and run all required upgrades:

```
student@ubuntu:~$ sudo apt update

Hit:1 http://us-central1.gce.archive.ubuntu.com/ubuntu focal InRelease
Get:2 http://us-central1.gce.archive.ubuntu.com/ubuntu focal-updates InRelease
[114 kB]
Get:3 http://us-central1.gce.archive.ubuntu.com/ubuntu focal-backports
InRelease [101 kB]
Get:4 http://security.ubuntu.com/ubuntu focal-security InRelease [114 kB]
Get:5 http://us-central1.gce.archive.ubuntu.com/ubuntu focal-updates/main
amd64 Packages [1127 kB]
...


student@ubuntu:~$ sudo apt upgrade -y

Reading package lists... Done
Building dependency tree
Reading state information... Done
Calculating upgrade... Done
...
```

Now it is safe to install the runc package:

```
student@ubuntu:~$ sudo apt install -y runc
```

```
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following package was automatically installed and is no longer required:
  libnuma1
Use 'sudo apt autoremove' to remove it.
The following NEW packages will be installed:
  runc
0 upgraded, 1 newly installed, 0 to remove and 0 not upgraded.
Need to get 4087 kB of archives.
After this operation, 15.9 MB of additional disk space will be used.
Get:1 http://us-central1.gce.archive.ubuntu.com/ubuntu focal-updates/main
amd64 runc amd64 1.0.0~rc95-0ubuntu1~20.04.2 [4087 kB]
Fetched 4087 kB in 0s (30.6 MB/s)
Selecting previously unselected package runc.
(Reading database ... 126756 files and directories currently installed.)
Preparing to unpack .../runc_1.0.0~rc95-0ubuntu1~20.04.2_amd64.deb ...
Unpacking runc (1.0.0~rc95-0ubuntu1~20.04.2) ...
Setting up runc (1.0.0~rc95-0ubuntu1~20.04.2) ...
Processing triggers for man-db (2.9.1-1) ...
```

Verify the version of the installed runc package:

```
student@ubuntu:~$ runc --version

runc version 1.0.0~rc95-0ubuntu1~20.04.2
spec: 1.0.2-dev
go: go1.13.8
libseccomp: 2.5.1
```

Lab 4.2. Install containerd

Although containerd is not intended to be used directly by users, we will still explore the installation method of the containerd package on Ubuntu. In a future lab exercise we will install crictl, a Beta CLI tool designed to allow users to interact with simple runtimes such as containerd.

First, update the system and run all required upgrades:

```
student@ubuntu:~$ sudo apt update

Hit:1 http://us-central1.gce.archive.ubuntu.com/ubuntu focal InRelease
Get:2 http://us-central1.gce.archive.ubuntu.com/ubuntu focal-updates InRelease
[114 kB]
Get:3 http://us-central1.gce.archive.ubuntu.com/ubuntu focal-backports
InRelease [101 kB]
Get:4 http://security.ubuntu.com/ubuntu focal-security InRelease [114 kB]
Get:5 http://us-central1.gce.archive.ubuntu.com/ubuntu focal-updates/main
amd64 Packages [1127 kB]
...


student@ubuntu:~$ sudo apt upgrade -y

Reading package lists... Done
Building dependency tree
Reading state information... Done
Calculating upgrade... Done
...
```

Now it is safe to install the containerd package:

```
student@ubuntu:~$ sudo apt install -y containerd
```

```
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following package was automatically installed and is no longer required:
  libnuma1
Use 'sudo apt autoremove' to remove it.
The following NEW packages will be installed:
  containerd
0 upgraded, 1 newly installed, 0 to remove and 0 not upgraded.
Need to get 32.9 MB of archives.
After this operation, 150 MB of additional disk space will be used.
Get:1 http://us-central1.gce.archive.ubuntu.com/ubuntu focal-updates/main
amd64 containerd amd64 1.5.2-0ubuntu1~20.04.2 [32.9 MB]
Fetched 32.9 MB in 1s (42.1 MB/s)
Selecting previously unselected package containerd.
(Reading database ... 126781 files and directories currently installed.)
Preparing to unpack .../containerd_1.5.2-0ubuntu1~20.04.2_amd64.deb ...
Unpacking containerd (1.5.2-0ubuntu1~20.04.2) ...
Setting up containerd (1.5.2-0ubuntu1~20.04.2) ...
Created symlink /etc/systemd/system/multi-user.target.wants/containerd.service
→ /lib/systemd/system/con
tainerd.service.
Processing triggers for man-db (2.9.1-1) ...
```

Verify the version of the installed containerd package:

```
student@ubuntu:~$ containerd --version

containerd github.com/containerd/containerd 1.5.2-0ubuntu1~20.04.2
```

As containerd runs as a daemon on the Ubuntu host, we can manage it with systemd:

```
student@ubuntu:~$ sudo systemctl status containerd.service

● containerd.service - containerd container runtime
   Loaded: loaded (/lib/systemd/system/containerd.service; enabled; vendor
preset: enabled)
   Active: active (running) since Wed 2021-07-28 22:28:01 UTC; 4min 54s ago
     Docs: https://containerd.io
  Process: 216870 ExecStartPre=/sbin/modprobe overlay (code=exited,
status=0/SUCCESS)
   Main PID: 216871 (containerd)
      Tasks: 10
     Memory: 52.7M
     CGroup: /system.slice/containerd.service
```

```
             └─216871 /usr/bin/containerd

Jul 28 22:28:01 gce-vm containerd[216871]:
time="2021-07-28T22:28:01.360346558Z" level=info msg="loading plugin
\"io.containerd.grpc.v1.introspection\"..." type=io.containerd.grpc.v1
Jul 28 22:28:01 gce-vm containerd[216871]:
time="2021-07-28T22:28:01.361537665Z" level=info msg=serving...
address=/run/containerd/containerd.sock.ttrpc
...
```

Lab 4.3. Install Docker Engine

Docker Engine has many installation options. While for Mac and Windows there is the Docker Desktop, for Linux distributions there are repositories including software packages of the docker components. Let's install on Ubuntu the Docker Engine from the official Docker repository.

To avoid any possible incompatibilities, it requires several steps for system cleanup, repository setup, and for package installation. Let's remove all related previously installed packages - docker, containerd, and runc (if any):

```
student@ubuntu:~$ sudo apt remove docker docker-engine docker.io containerd
runc
```

Update packages:

```
student@ubuntu:~$ sudo apt update

Hit:1 http://us-central1.gce.archive.ubuntu.com/ubuntu focal InRelease
Get:2 http://us-central1.gce.archive.ubuntu.com/ubuntu focal-updates InRelease
[114 kB]
Get:3 http://us-central1.gce.archive.ubuntu.com/ubuntu focal-backports
InRelease [101 kB]
Get:4 http://security.ubuntu.com/ubuntu focal-security InRelease [114 kB]
Get:5 http://us-central1.gce.archive.ubuntu.com/ubuntu focal-updates/main
amd64 Packages [1127 kB]
...
```

Install packages required for HTTPS repository:

```
student@ubuntu:~$ sudo apt install -y apt-transport-https ca-certificates curl
gnupg lsb-release
```

```
Reading package lists... Done
Building dependency tree
Reading state information... Done
lsb-release is already the newest version (11.1.0ubuntu2).
lsb-release set to manually installed.
ca-certificates is already the newest version (20210119~20.04.1).
ca-certificates set to manually installed.
curl is already the newest version (7.68.0-1ubuntu2.6).
curl set to manually installed.
gnupg is already the newest version (2.2.19-3ubuntu2.1).
gnupg set to manually installed.
The following package was automatically installed and is no longer required:
  libnuma1
Use 'sudo apt autoremove' to remove it.
The following NEW packages will be installed:
  apt-transport-https
0 upgraded, 1 newly installed, 0 to remove and 0 not upgraded.
Need to get 4680 B of archives.
After this operation, 162 kB of additional disk space will be used.
Get:1 http://us-central1.gce.archive.ubuntu.com/ubuntu focal-updates/universe
amd64 apt-transport-https all 2.0.6 [4680 B]
Fetched 4680 B in 0s (98.2 kB/s)
Selecting previously unselected package apt-transport-https.
(Reading database ... 126810 files and directories currently installed.)
Preparing to unpack .../apt-transport-https_2.0.6_all.deb ...
Unpacking apt-transport-https (2.0.6) ...
Setting up apt-transport-https (2.0.6) ...
```

Add Docker GPG key:

```
student@ubuntu:~$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg |
sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg
```

Add the stable repository:

```
student@ubuntu:~$ echo "deb [arch=amd64
signed-by=/usr/share/keyrings/docker-archive-keyring.gpg]
https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" | sudo tee
/etc/apt/sources.list.d/docker.list > /dev/null
```

Update packages:

```
student@ubuntu:~$ sudo apt update
```

Install the latest version of Docker Engine:

```
student@ubuntu:~$ sudo apt install -y docker-ce docker-ce-cli containerd.io

Reading package lists... Done
Building dependency tree
Reading state information... Done
The following package was automatically installed and is no longer required:
  libnuma1
Use 'sudo apt autoremove' to remove it.
The following additional packages will be installed:
  docker-ce-rootless-extras docker-scan-plugin pigz slirp4netns
Suggested packages:
  aufs-tools cgroupfs-mount | cgroup-lite
The following packages will be REMOVED:
  containerd runc
The following NEW packages will be installed:
  containerd.io docker-ce docker-ce-cli docker-ce-rootless-extras
docker-scan-plugin pigz
  slirp4netns
 ...
```

Display the version of the Docker Engine components. We may notice that Docker does not use the latest releases of containerd and runc, installed in earlier exercises:

```
student@ubuntu:~$ sudo docker version

Client: Docker Engine - Community
 Version:           20.10.7
 API version:       1.41
 Go version:        go1.13.15
 Git commit:        f0df350
 Built:             Wed Jun  2 11:56:38 2021
 OS/Arch:           linux/amd64
 Context:           default
 Experimental:      true

Server: Docker Engine - Community
 Engine:
  Version:          20.10.7
  API version:      1.41 (minimum version 1.12)
  Go version:       go1.13.15
  Git commit:       b0f5bc3
  Built:            Wed Jun  2 11:54:50 2021
  OS/Arch:          linux/amd64
```

```
 Experimental:      false
containerd:
 Version:           1.4.8
 GitCommit:         7eba5930496d9bbe375fdf71603e610ad737d2b2
runc:
 Version:           1.0.0
 GitCommit:         v1.0.0-0-g84113ee
docker-init:
 Version:           0.19.0
 GitCommit:         de40ad0
```

Verify the installation by running a test image:

```
student@ubuntu:~$ sudo docker run hello-world

Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
b8dfde127a29: Pull complete
Digest:
sha256:df5f5184104426b65967e016ff2ac0bfcd44ad7899ca3bbcf8e44e4461491a9e
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

We can manage the Docker daemon with systemd:

```
student@ubuntu:~$ sudo systemctl status docker.service

● docker.service - Docker Application Container Engine
   Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset:
enabled)
   Active: active (running) since Wed 2021-07-28 23:07:50 UTC; 9min ago
TriggeredBy: ● docker.socket
       Docs: https://docs.docker.com
   Main PID: 218203 (dockerd)
      Tasks: 10
     Memory: 83.7M
     CGroup: /system.slice/docker.service
             └─218203 /usr/bin/dockerd -H fd://
--containerd=/run/containerd/containerd.sock

Jul 28 23:07:50 gce-vm dockerd[218203]: time="2021-07-28T23:07:50.198910473Z"
level=warning msg="Your kernel does not support cgroup blkio weight"
Jul 28 23:07:50 gce-vm dockerd[218203]: time="2021-07-28T23:07:50.199105926Z"
level=warning msg="Your kernel does not support cgroup blkio weight_device"
...
```

In order to manage Docker as a non-root user, we need to create a group called **docker** and add our user ID to it:

```
student@ubuntu:~$ sudo groupadd docker

groupadd: group 'docker' already exists
```

If you see a similar output, then the docker group already exists on your Docker host. It is safe to proceed to the next step:

```
student@ubuntu:~$ sudo usermod -aG docker $USER

student@ubuntu:~$ newgrp docker
```

Verify the installation again, this time without **sudo**:

```
student@ubuntu:~$ docker run hello-world
```

Lab 4.4. Install Podman

The Podman runtime for OCI and Docker containers, runs both privileged and unprivileged containers, wrapped in a container pod. The container pod is extensively used by Kubernetes, and it was used by rkt as well. Beginning with Ubuntu 20.10, the Podman package can be installed directly from the official repository. For prior OS releases, such as ours, the Podman package can be installed via the Kubic project.

Let's install Podman. First source the host OS version information, to make it available as environment variables to subsequent commands:

```
student@ubuntu:~$ . /etc/os-release
```

Add the repository to the list of apt sources:

```
student@ubuntu:~$ echo "deb
https://download.opensuse.org/repositories/devel:/kubic:/libcontainers:/stable
/xUbuntu_${VERSION_ID}/ /" | sudo tee
/etc/apt/sources.list.d/devel:kubic:libcontainers:stable.list
```

Retrieve the specific release key:

```
student@ubuntu:~$ curl -L
"https://download.opensuse.org/repositories/devel:/kubic:/libcontainers:/stabl
e/xUbuntu_${VERSION_ID}/Release.key" | sudo apt-key add -

  % Total    % Received % Xferd  Average Speed   Time    Time     Time
Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  1093  100  1093    0     0   2547      0 --:--:-- --:--:-- --:--:--  2547
OK
```

Update package index and upgrade:

```
student@ubuntu:~$ sudo apt update

student@ubuntu:~$ sudo apt upgrade -y
```

Install:

```
student@ubuntu:~$ sudo apt install -y podman

Reading package lists... Done
Building dependency tree
Reading state information... Done
The following package was automatically installed and is no longer required:
  libnuma1
Use 'sudo apt autoremove' to remove it.
The following additional packages will be installed:
  catatonit conmon containernetworking-plugins containers-common criu crun
fuse-overlayfs fuse3
  libfuse3-3 libnet1 libnl-3-200 libprotobuf-c1 libprotobuf17 libyajl2
podman-machine-cni
  podman-plugins python3-protobuf
...
```

Once installed, let's verify the version:

```
student@ubuntu:~$ podman --version

podman version 3.2.3
```

Lab 4.5. Install CRI-O

Although CRI-O is not intended to be used directly by users, we will still explore the installation method of the CRI-O package on Ubuntu, followed by the installation of crictl, a Beta CLI designed to allow users to interact with simple container runtimes such as CRI-O and containerd.

First, update the system and run all required upgrades:

```
student@ubuntu:~$ sudo apt update
```

As root, set the **OS** variable to the appropriate value that reflects the current operating system (Ubuntu 20.04), and the **VERSION** variable to match the desired Kubernetes version (1.21):

```
student@ubuntu:~$ sudo -i

root@ubuntu:~# OS=xUbuntu_20.04

root@ubuntu:~# VERSION=1.21

root@ubuntu:~# VERSION_ID=20.04
```

Add new repositories with the following four commands, run as root, then exit:

```
root@ubuntu:~# echo "deb
https://download.opensuse.org/repositories/devel:/kubic:/libcontainers:/stable
/$OS/ /" > /etc/apt/sources.list.d/devel:kubic:libcontainers:stable.list

root@ubuntu:~# echo "deb
http://download.opensuse.org/repositories/devel:/kubic:/libcontainers:/stable:
/cri-o:/$VERSION/$OS/ /" >
/etc/apt/sources.list.d/devel:kubic:libcontainers:stable:cri-o:$VERSION.list

root@ubuntu:~# curl -L
https://download.opensuse.org/repositories/devel:kubic:libcontainers:stable:cr
i-o:$VERSION/$OS/Release.key | apt-key add -
```

```
root@ubuntu:~# curl -L
https://download.opensuse.org/repositories/devel:/kubic:/libcontainers:/stable
/$OS/Release.key | apt-key add -

root@ubuntu:~# exit
```

Update package index:

```
student@ubuntu:~$ sudo apt update
```

Install CRI-O packages:

```
student@ubuntu:~$ sudo apt install -y cri-o cri-o-runc

Reading package lists... Done
Building dependency tree
Reading state information... Done
The following package was automatically installed and is no longer required:
  libnuma1
Use 'sudo apt autoremove' to remove it.
The following NEW packages will be installed:
  cri-o cri-o-runc
0 upgraded, 2 newly installed, 0 to remove and 0 not upgraded.
Need to get 22.9 MB of archives.
After this operation, 108 MB of additional disk space will be used.
Get:1
http://download.opensuse.org/repositories/devel:/kubic:/libcontainers:/stable:
/cri-o:/1.21/xUbuntu_20.04  cri-o 1.21.2~0 [19.9 MB]
Get:2
https://download.opensuse.org/repositories/devel:/kubic:/libcontainers:/stable
/xUbuntu_20.04  cri-o-runc 1.0.1~0 [3016 kB]
Fetched 22.9 MB in 3s (8322 kB/s)
...
```

Display CRI-O version:

```
student@ubuntu:~$ crio --version

crio version 1.21.2
Version:       1.21.2
GitCommit:     aaefa6b173f79384c6a2a627e5074a7f5b02957f
GitTreeState:  clean
BuildDate:     2021-07-20T20:44:03Z
```

```
GoVersion:      go1.15.2
Compiler:       gc
Platform:       linux/amd64
Linkmode:       dynamic
```

Lab 4.6. Install crictl

Install the desired/latest crictl release (check the release page). First, set the **VERSION** variable to the desired release value:

```
student@ubuntu:~$ VERSION="v1.21.0"

student@ubuntu:~$ curl -L
https://github.com/kubernetes-sigs/cri-tools/releases/download/$VERSION/crictl
-${VERSION}-linux-amd64.tar.gz --output crictl-${VERSION}-linux-amd64.tar.gz

student@ubuntu:~$ sudo tar zxvf crictl-$VERSION-linux-amd64.tar.gz -C
/usr/local/bin

student@ubuntu:~$ rm -f crictl-$VERSION-linux-amd64.tar.gz
```

Display crictl version:

```
student@ubuntu:~$ crictl --version

crictl version v1.21.0
```

Start the CRI-O service and then display crictl runtime version:

```
student@ubuntu:~$ sudo systemctl start crio.service

student@ubuntu:~$ sudo crictl version

Version:  0.1.0
RuntimeName:  cri-o
RuntimeVersion:  1.21.2
RuntimeApiVersion:  v1alpha2
```

The **/etc/crictl.yaml** config file allows us to declare the desired CRI compatible runtime endpoint the CLI is expected to interact with. If the runtime endpoint is not specifically declared, crictl will search by default for **dockershim** (for the Docker runtime), **containerd**, and **crio**. We will use crio for our lab exercises:

```
student@ubuntu:~$ cat /etc/crictl.yaml

runtime-endpoint: "unix:///var/run/crio/crio.sock"
timeout: 0
debug: false
```

Lab 4.7. Install LXD

In an earlier exercise we explored the LXC installation, an Operating System virtualization mechanism that uses the Linux system as a runtime for Linux Containers. LXD, another tool to manage Linux Containers, enhances the LXC experience by replacing the LXC set of tools (lxc-create, lxc-start, lxc-stop, lxc-info, etc.) with a single CLI tool based on a REST API that enables management of remote containers as well. The recommended installation method is via snap, however, snap is known to store large amounts of unnecessary data on our filesystems. With this in mind, we will install LXD directly from the Ubuntu package repository.

Let's start with a system update and upgrade:

```
student@ubuntu:~$ sudo apt update

student@ubuntu:~$ sudo apt upgrade -y
```

Install LXD:

```
student@ubuntu:~$ sudo apt install -y lxd

Reading package lists... Done
Building dependency tree
Reading state information... Done
The following package was automatically installed and is no longer required:
  libnuma1
Use 'sudo apt autoremove' to remove it.
The following NEW packages will be installed:
  lxd
0 upgraded, 1 newly installed, 0 to remove and 0 not upgraded.
Need to get 5444 B of archives.
After this operation, 79.9 kB of additional disk space will be used.
Get:1 http://us-central1.gce.archive.ubuntu.com/ubuntu focal/universe amd64
lxd all 1:0.9 [5444 B]
Fetched 5444 B in 0s (38.7 kB/s)
Preconfiguring packages ...
```

```
Selecting previously unselected package lxd.
(Reading database ... 127566 files and directories currently installed.)
Preparing to unpack .../archives/lxd_1%3a0.9_all.deb ...
Unpacking lxd (1:0.9) ...
Setting up lxd (1:0.9) ...
```

Once installed, we can verify the version of the LXD package:

```
student@ubuntu:~$ lxd version

4.0.7
```

In order to be able to run LXD as a non-root user, our own user ID needs to be a member of the lxd group. We can validate by running the following command where **(lxd)** should be a member of the **groups** list. Your group ID number may be different, such as **116** or **117**:

```
student@ubuntu:~$ id

uid=1001(student) gid=1002(student)
groups=1002(student)...46(plugdev),117(netdev),118(lxd),998(docker),1000(ubunt
u)...
```

If our own user ID is not a member of the lxd group, then we need to add it manually:

```
student@ubuntu:~$ sudo adduser $USER lxd

student@ubuntu:~$ newgrp lxd

student@ubuntu:~$ id
```

Lab 5.1. Setting Up a Private Docker Registry

In addition to public registries, such as Docker Hub and quay.io, Docker is capable of interacting with private registries as well. Let's set up a private registry, by running Docker's own registry container image:

```
student@ubuntu:~$ docker run -d -p 5000:5000 --restart=always --name registry
registry:2

Unable to find image 'registry:2' locally
2: Pulling from library/registry
ddad3d7c1e96: Pull complete
6eda6749503f: Pull complete
363ab70c2143: Pull complete
5b94580856e6: Pull complete
12008541203a: Pull complete
Digest:
sha256:121baf25069a56749f249819e36b386d655ba67116d9c1c6c8594061852de4da
Status: Downloaded newer image for registry:2
892d208a84d1719f0f5cec0e6f9e056f7f27ab3426d6a9ea090d6a52d8a8545c
```

With the registry running in a "registry" container, we mapped the container's port 5000 to the host port 5000 so that we can access it via localhost. The private registry currently does not hold any images. So let's populate it by pulling an image from the public registry Docker Hub, tag it and push it into the private registry, while listing local cache registries to validate:

```
student@ubuntu:~$ docker image pull alpine:3.14

3.14: Pulling from library/alpine
Digest:
sha256:eb3e4e175ba6d212ba1d6e04fc0782916c08e1c9d7b45892e9796141b1d379ae
Status: Downloaded newer image for alpine:3.14
```

```
docker.io/library/alpine:3.14
```

```
student@ubuntu:~$ docker images

REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
alpine        3.14      021b3423115f   10 days ago    5.6MB
registry      2         1fd8e1b0bb7e   4 months ago   26.2MB
hello-world   latest    d1165f221234   5 months ago   13.3kB
```

Tag the newly pulled image:

```
student@ubuntu:~$ docker image tag alpine:3.14 localhost:5000/myalps
```

```
student@ubuntu:~$ docker images

REPOSITORY              TAG       IMAGE ID       CREATED        SIZE
alpine                  3.14      021b3423115f   10 days ago    5.6MB
localhost:5000/myalps   latest    021b3423115f   10 days ago    5.6MB
registry                2         1fd8e1b0bb7e   4 months ago   26.2MB
hello-world             latest    d1165f221234   5 months ago   13.3kB
```

Push the newly tagged image to the private registry:

```
student@ubuntu:~$ docker image push localhost:5000/myalps

Using default tag: latest
The push refers to repository [localhost:5000/myalps]
bc276c40b172: Pushed
latest: digest:
sha256:be9bdc0ef8e96dbc428dc189b31e2e3b05523d96d12ed627c37aa2936653258c size:
528
```

Now we will remove the two cached images **alpine:3.14** and **localhost:5000/myalps**:

```
student@ubuntu:~$ docker image rm alpine:3.14

Untagged: alpine:3.14
Untagged:
alpine@sha256:eb3e4e175ba6d212ba1d6e04fc0782916c08e1c9d7b45892e9796141b1d379ae
```

```
student@ubuntu:~$ docker image rm localhost:5000/myalps

Untagged: localhost:5000/myalps:latest
Untagged:
localhost:5000/myalps@sha256:be9bdc0ef8e96dbc428dc189b31e2e3b05523d96d12ed627c
37aa2936653258c


student@ubuntu:~$ docker images

REPOSITORY    TAG       IMAGE ID       CREATED         SIZE
registry      2         1fd8e1b0bb7e   4 months ago    26.2MB
hello-world   latest    d1165f221234   5 months ago    13.3kB
```

Pull the image from the private registry:

```
student@ubuntu:~$ docker image pull localhost:5000/myalps

Using default tag: latest
latest: Pulling from myalps
Digest:
sha256:be9bdc0ef8e96dbc428dc189b31e2e3b05523d96d12ed627c37aa2936653258c
Status: Downloaded newer image for localhost:5000/myalps:latest
localhost:5000/myalps:latest


student@ubuntu:~$ docker images

REPOSITORY              TAG       IMAGE ID       CREATED         SIZE
localhost:5000/myalps   latest    021b3423115f   10 days ago     5.6MB
registry                2         1fd8e1b0bb7e   4 months ago    26.2MB
hello-world             latest    d1165f221234   5 months ago    13.3kB
```

After successfully validating that the private registry can store container images, and it can be used to push and pull images to and from it, lets remove images that are no longer needed:

```
student@ubuntu:~$ docker image rm -f registry:2

Untagged: registry:2
```

```
Untagged:
registry@sha256:121baf25069a56749f249819e36b386d655ba67116d9c1c6c8594061852de4
da


student@ubuntu:~$ docker image rm localhost:5000/myalps

Untagged: localhost:5000/myalps:latest
Untagged:
localhost:5000/myalps@sha256:be9bdc0ef8e96dbc428dc189b31e2e3b05523d96d12ed627c
37aa2936653258c
```

Lab 5.2. Image Operations with Docker

Since we have configured Docker to run as **non-root**, we can now run docker commands as the **student** user. Let's search the Docker Hub registry for container images, using **nginx** as a search term. Many search results will be displayed, but only one "official" image, which we will use in subsequent steps:

```
student@ubuntu:~$ docker search nginx

NAME                                 DESCRIPTION
STARS     OFFICIAL   AUTOMATED
nginx                                Official build of Nginx.
15303     [OK]
jwilder/nginx-proxy                  Automated Nginx reverse proxy for docker
con…   2059                  [OK]
richarvey/nginx-php-fpm              Container running Nginx + PHP-FPM capable
of…   815                   [OK]
jc21/nginx-proxy-manager             Docker container for managing Nginx proxy
ho…   232
linuxserver/nginx                    An Nginx container, brought to you by
LinuxS…   150
tiangolo/nginx-rtmp                  Docker image with Nginx using the
nginx-rtmp…   138                     [OK]
jlesage/nginx-proxy-manager          Docker container for Nginx Proxy Manager
131                   [OK]
alfg/nginx-rtmp                      NGINX, nginx-rtmp-module and FFmpeg from
sou…   105                   [OK]
nginxdemos/hello                     NGINX webserver that serves a simple page
co…   70                    [OK]
...
```

Let's pull the official **nginx** image from the Docker Hub:

```
student@ubuntu:~$ docker image pull nginx

Using default tag: latest
```

```
latest: Pulling from library/nginx
33847f680f63: Pull complete
dbb907d5159d: Pull complete
8a268f30c42a: Pull complete
b10cf527a02d: Pull complete
c90b090c213b: Pull complete
1f41b2f2bf94: Pull complete
Digest:
sha256:8f335768880da6baf72b70c701002b45f4932acae8d574dedfddaf967fc3ac90
Status: Downloaded newer image for nginx:latest
docker.io/library/nginx:latest
```

~~If we had a private registry, such as **myregistry.com**, listening on port **5000**, then we could possibly pull an image from the private registry, by specifying the registry name, port number, image name (**alpine**) and tag (**latest**). The following command is provided only for illustrational purposes:~~

```
student@ubuntu:~$ sudo docker image pull myregistry.com:5000/alpine:latest
```

List images cached in the local repository. The **nginx** image was just pulled, while the **hello-world** image was pulled in a prior lab exercise when we validated the docker installation by running the test image:

```
student@ubuntu:~$ docker image ls

REPOSITORY    TAG       IMAGE ID       CREATED         SIZE
nginx         latest    08b152afcfae   3 weeks ago     133MB
hello-world   latest    d1165f221234   5 months ago    13.3kB
```

List images and their digests. A digest is a hash associated with the content of each layer part of the image. Digests ensure each layer's integrity. The image ID is another hash, derived from the JSON configuration file of the image. The output has been truncated for readability:

```
student@ubuntu:~$ docker image ls --digests

REPOSITORY    TAG       DIGEST            IMAGE ID       CREATED         SIZE
nginx         latest    sha256:8f335...   08b152afcfae   3 weeks ago     133MB
hello-world   latest    sha256:df5f5...   d1165f221234   5 months ago    13.3kB
```

Below we see the full digest of the **nginx** image:

```
sha256:8f335768880da6baf72b70c701002b45f4932acae8d574dedfddaf967fc3ac90
```

Push an image to Docker Hub. Assuming the existence of a **lfstudent** account on Docker Hub, a user can login from the CLI and then push an image into the registry to be shared with other users. The following commands are provided only for illustrational purposes:

```
student@ubuntu:~$ docker login

Login with your Docker ID to push and pull images from Docker Hub. If you
don't
have a Docker ID, head over to https://hub.docker.com to create one.
Username: lfstudent
Password: ********
Login Succeeded


student@ubuntu:~$ sudo docker image push lfstudent/alpine:training

The push refers to a repository [docker.io/lfstudent/alpine]
724d404d96ef: Pushed
60ab55d3379d: Mounted from library/alpine
training: digest:
sha256:fcc29a8a772bed232fc3026f9ea5df745e57ea4c99b2306f997036510d4be0f9 size:
735
```

~~Push an image to a private registry. The following command is provided only for illustrational purposes:~~

```
~~student@ubuntu:~$ sudo docker image push myregistry.com:5000/alpine:training~~
```

Let's pull another image from Docker Hub, the official **alpine** image, and tag it:

```
student@ubuntu:~$ docker image search alpine

NAME                                      DESCRIPTION
STARS     OFFICIAL   AUTOMATED
alpine                                    A minimal Docker image based on Alpine
Linux…    7748       [OK]
mhart/alpine-node                         Minimal Node.js built on Alpine Linux
483
anapsix/alpine-java                       Oracle Java 8 (and 7) with GLIBC 2.28
over A…   472                  [OK]
...


student@ubuntu:~$ docker image pull alpine
```

```
Using default tag: latest
latest: Pulling from library/alpine
29291e31a76a: Pull complete
Digest:
sha256:eb3e4e175ba6d212ba1d6e04fc0782916c08e1c9d7b45892e9796141b1d379ae
Status: Downloaded newer image for alpine:latest
docker.io/library/alpine:latest
```

```
student@ubuntu:~$ docker image ls

REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
alpine        latest    021b3423115f   6 days ago     5.6MB
nginx         latest    08b152afcfae   3 weeks ago    133MB
hello-world   latest    d1165f221234   5 months ago   13.3kB
```

```
student@ubuntu:~$ docker image tag alpine myalps:v1
```

```
student@ubuntu:~$ docker image ls

REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
alpine        latest    021b3423115f   6 days ago     5.6MB
myalps        v1        021b3423115f   6 days ago     5.6MB
nginx         latest    08b152afcfae   3 weeks ago    133MB
hello-world   latest    d1165f221234   5 months ago   13.3kB
```

If we examine closely the output, we notice that the **tag** command generated another image entry **myalps:v1**, referencing the same image ID as the original **alpine:latest** image. Both the image ID and size values are identical.

Let's display the **alpine** image details. Between the image ID and its digest, we find both the **alpine** and **myalps** tags. Try running the **inspect** command with the **myalps:v1** image and compare the two outputs:

```
student@ubuntu:~$ docker image inspect alpine

[
    {
        "Id":
"sha256:021b3423115ff662225e83d7e2606475217de7b55fde83ce3447a54019a77aa2",
        "RepoTags": [
            "alpine:latest",
            "myalps:v1"
        ],
        "RepoDigests": [
```

```
"alpine@sha256:eb3e4e175ba6d212ba1d6e04fc0782916c08e1c9d7b45892e9796141b1d379a
e"
        ],
        "Parent": "",
        "Comment": "",
        "Created": "2021-08-06T17:19:45.183996158Z",
        "Container":
"72bdff81d8bf6dfb5d83e6cbdd59564e419e3a97931ce4e31dd3f215eae4914c",
        "ContainerConfig": {
            "Hostname": "72bdff81d8bf",
            "Domainname": "",
            "User": "",
            "AttachStdin": false,
            "AttachStdout": false,
            "AttachStderr": false,
            "Tty": false,
            "OpenStdin": false,
            "StdinOnce": false,
            "Env": [
"PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
            ],
            "Cmd": [
                "/bin/sh",
                "-c",
                "#(nop) ",
                "CMD [\"/bin/sh\"]"
            ],
            "Image":
"sha256:dc0d5c673ee2846463a00ec42d78344d3d5f37ac23c0723da4cd0051c16f4ef4",
            "Volumes": null,
            "WorkingDir": "",
            "Entrypoint": null,
            "OnBuild": null,
            "Labels": {}
        },
        "DockerVersion": "20.10.7",
        "Author": "",
        "Config": {
            "Hostname": "",
            "Domainname": "",
            "User": "",
            "AttachStdin": false,
            "AttachStdout": false,
            "AttachStderr": false,
            "Tty": false,
            "OpenStdin": false,
            "StdinOnce": false,
```

```
            "Env": [
"PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
            ],
            "Cmd": [
                "/bin/sh"
            ],
            "Image":
"sha256:dc0d5c673ee2846463a00ec42d78344d3d5f37ac23c0723da4cd0051c16f4ef4",
            "Volumes": null,
            "WorkingDir": "",
            "Entrypoint": null,
            "OnBuild": null,
            "Labels": null
        },
        "Architecture": "amd64",
        "Os": "linux",
        "Size": 5595292,
        "VirtualSize": 5595292,
        "GraphDriver": {
            "Data": {
                "MergedDir":
"/var/lib/docker/overlay2/6bbabbf0f44c4f7955180c54680093f02d67da20463d903e5981
96b6cc35503b/merged",
                "UpperDir":
"/var/lib/docker/overlay2/6bbabbf0f44c4f7955180c54680093f02d67da20463d903e5981
96b6cc35503b/diff",
                "WorkDir":
"/var/lib/docker/overlay2/6bbabbf0f44c4f7955180c54680093f02d67da20463d903e5981
96b6cc35503b/work"
            },
            "Name": "overlay2"
        },
        "RootFS": {
            "Type": "layers",
            "Layers": [
"sha256:bc276c40b172b1c5467277d36db5308a203a48262d5f278766cf083947d05466"
            ]
        },
        "Metadata": {
            "LastTagTime": "2021-08-12T20:34:52.630348089Z"
        }
    }
]
```


Let's remove a cached image from the local repository (using the **--force** option), then list available images with **images** this time instead of **image ls**:

```
student@ubuntu:~$ docker image rm -f alpine

Untagged: alpine:latest
Untagged:
alpine@sha256:eb3e4e175ba6d212ba1d6e04fc0782916c08e1c9d7b45892e9796141b1d379ae


student@ubuntu:~$ docker images

REPOSITORY    TAG       IMAGE ID       CREATED         SIZE
myalps        v1        021b3423115f   6 days ago      5.6MB
nginx         latest    08b152afcfae   3 weeks ago     133MB
hello-world   latest    d1165f221234   5 months ago    13.3kB
```

Lab 5.3. Image Operations with Podman

For Docker users transitioning to Podman and its command line structure should be quite seamless. The typical recommendation is to set an alias such as **alias docker=podman**. With this in mind, commands that we have run earlier with **docker**, can be easily replicated with **podman**.

Let's search for the official **nginx** image and pull it to the local cache, then list to validate its presence. When performing a search, by default podman searchse both the **docker.io** and **quay.io** registries:

```
student@ubuntu:~$ podman search nginx

INDEX       NAME
DESCRIPTION                                     STARS       OFFICIAL
AUTOMATED
docker.io   docker.io/library/nginx
Official build of Nginx.                        15303       [OK]
docker.io   docker.io/jwilder/nginx-proxy
Automated Nginx reverse proxy for docker con...  2059                    [OK]
docker.io   docker.io/nginxinc/nginx-unprivileged
Unprivileged NGINX Dockerfiles                  46
...


student@ubuntu:~$ podman image pull docker.io/library/nginx

Trying to pull docker.io/library/nginx:latest...
Getting image source signatures
Copying blob 8a268f30c42a done
Copying blob b10cf527a02d done
Copying blob 1f41b2f2bf94 done
Copying blob dbb907d5159d done
Copying blob 33847f680f63 done
Copying blob c90b090c213b done
```

```
Copying config 08b152afcf done
Writing manifest to image destination
Storing signatures
08b152afcfae220e9709f00767054b824361c742ea03a9fe936271ba520a0a4b
```

```
student@ubuntu:~$ podman image ls

REPOSITORY               TAG         IMAGE ID      CREATED      SIZE
docker.io/library/nginx  latest      08b152afcfae  3 weeks ago  137 MB
```

```
student@ubuntu:~$ podman image ls --digests

REPOSITORY              TAG    DIGEST          IMAGE ID    CREATED      SIZE
docker.io/library/nginx latest sha256:8f335... 08b152a...  3 weeks ago  137 MB
```

The full digest:

```
sha256:8f335768880da6baf72b70c701002b45f4932acae8d574dedfddaf967fc3ac90
```

Let's search for the official **alpine** image and pull it to the local cache, then list to validate its presence. Inspect the image, and compare the output with the earlier **docker image inspect** output:

```
student@ubuntu:~$ podman search alpine

INDEX       NAME                                          DESCRIPTION
STARS       OFFICIAL    AUTOMATED
docker.io   docker.io/library/alpine                      A minimal Docker
image based on Alpine Linux...  7748        [OK]
docker.io   docker.io/alpine/git                          A  simple git
container running in alpine li...  189                  [OK]
...
```

```
student@ubuntu:~$ podman image pull docker.io/library/alpine

Trying to pull docker.io/library/alpine:latest...
Getting image source signatures
Copying blob 29291e31a76a done
Copying config 021b342311 done
Writing manifest to image destination
Storing signatures
021b3423115ff662225e83d7e2606475217de7b55fde83ce3447a54019a77aa2
```

```
student@ubuntu:~$ podman image ls

REPOSITORY                TAG         IMAGE ID      CREATED      SIZE
docker.io/library/alpine  latest      021b3423115f  6 days ago   5.87 MB
docker.io/library/nginx   latest      08b152afcfae  3 weeks ago  137 MB


student@ubuntu:~$ podman image tag alpine myalps:v2

podman image ls
REPOSITORY                TAG         IMAGE ID      CREATED      SIZE
docker.io/library/alpine  latest      021b3423115f  6 days ago   5.87 MB
localhost/myalps          v2          021b3423115f  6 days ago   5.87 MB
docker.io/library/nginx   latest      08b152afcfae  3 weeks ago  137 MB


student@ubuntu:~$ podman image inspect alpine

[
    {
        "Id":
"021b3423115ff662225e83d7e2606475217de7b55fde83ce3447a54019a77aa2",
        "Digest":
"sha256:eb3e4e175ba6d212ba1d6e04fc0782916c08e1c9d7b45892e9796141b1d379ae",
        "RepoTags": [
            "docker.io/library/alpine:latest",
            "localhost/myalps:v2"
        ],
        "RepoDigests": [
"docker.io/library/alpine@sha256:be9bdc0ef8e96dbc428dc189b31e2e3b05523d96d12ed
627c37aa2936653258c",
"docker.io/library/alpine@sha256:eb3e4e175ba6d212ba1d6e04fc0782916c08e1c9d7b45
892e9796141b1d379ae",
"localhost/myalps@sha256:be9bdc0ef8e96dbc428dc189b31e2e3b05523d96d12ed627c37aa
2936653258c",
"localhost/myalps@sha256:eb3e4e175ba6d212ba1d6e04fc0782916c08e1c9d7b45892e9796
141b1d379ae"
...
    }
]


student@ubuntu:~$ podman image rm -f alpine

Untagged: docker.io/library/alpine:latest
```

```
student@ubuntu:~$ podman images

REPOSITORY                TAG         IMAGE ID      CREATED      SIZE
localhost/myalps          v2          021b3423115f  6 days ago   5.87 MB
docker.io/library/nginx   latest      08b152afcfae  3 weeks ago  137 MB
```

Lab 5.4. Image Operations with crictl

Crictl, the CLI tool designed for CRI compatible runtimes such as CRI-O, containerd, and even Docker, supports a limited set operations on container images. Some of the commands are similar in syntax and effect to commands encountered earlier with Docker and Podman.

Crictl expects to run as **root**, therefore we will run **sudo crictl**. Otherwise, it may display the following error message:

```
FATA[0002] connect: connect endpoint 'unix:///var/run/crio/crio.sock', make
sure you are running as root and the endpoint has been started: context
deadline exceeded
```

We notice that there is no **search** command, meaning that the user is expected to know the name of the desired image that is to be pulled from the registry:

```
student@ubuntu:~$ sudo crictl pull nginx

Image is up to date for
docker.io/library/nginx@sha256:3f13b4376446cf92b0cb9a5c46ba75d57c41f627c4edb8b
635fa47386ea29e20
```

With the first image pulled, let's list images available in the local cache:

```
student@ubuntu:~$ sudo crictl images

IMAGE                      TAG                 IMAGE ID            SIZE
docker.io/library/nginx    latest              08b152afcfae2       137MB
```

Let's pull a second image from the registry:

```
student@ubuntu:~$ sudo crictl pull alpine
```

```
Image is up to date for
docker.io/library/alpine@sha256:be9bdc0ef8e96dbc428dc189b31e2e3b05523d96d12ed6
27c37aa2936653258c
```

With the second image pulled, let's list images available in the local cache:

```
student@ubuntu:~$ sudo crictl images

IMAGE                      TAG                 IMAGE ID            SIZE
docker.io/library/alpine   latest              021b3423115ff       5.87MB
docker.io/library/nginx    latest              08b152afcfae2       137MB
```

Let's inspect the second image from the local cache. We may notice a slight difference in the output of the **crictl inspecti** from the earlier **inspect** outputs of **docker** and **podman**:

```
student@ubuntu:~$ sudo crictl inspecti alpine

{
  "status": {
    "id": "021b3423115ff662225e83d7e2606475217de7b55fde83ce3447a54019a77aa2",
    "repoTags": [
      "docker.io/library/alpine:latest"
    ],
    "repoDigests": [
"docker.io/library/alpine@sha256:be9bdc0ef8e96dbc428dc189b31e2e3b05523d96d12ed
627c37aa2936653258c",
"docker.io/library/alpine@sha256:eb3e4e175ba6d212ba1d6e04fc0782916c08e1c9d7b45
892e9796141b1d379ae"
    ],
    "size": "5869637",
    "uid": null,
    "username": "",
    "spec": null
  },
  "info": {
    "imageSpec": {
      "created": "2021-08-06T17:19:45.183996158Z",
      "architecture": "amd64",
      "os": "linux",
      "config": {
        "Env": [
          "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
```

```
        ],
        "Cmd": [
          "/bin/sh"
        ]
      },
      "rootfs": {
        "type": "layers",
        "diff_ids": [
"sha256:bc276c40b172b1c5467277d36db5308a203a48262d5f278766cf083947d05466"
        ]
      },
      "history": [
        {
          "created": "2021-08-06T17:19:45.007652184Z",
          "created_by": "/bin/sh -c #(nop) ADD
file:34eb5c40aa00028921a224d1764ae1b1f3ef710d191e4dfc7df55e0594aa7217 in / "
        },
        {
          "created": "2021-08-06T17:19:45.183996158Z",
          "created_by": "/bin/sh -c #(nop)  CMD [\"/bin/sh\"]",
          "empty_layer": true
        }
      ]
    }
  }
}
```

Let's remove one of the images from the local cache:

```
student@ubuntu:~$ sudo crictl rmi alpine

Deleted: docker.io/library/alpine:latest
```

List images again, to confirm the removal of the **alpine** image:

```
student@ubuntu:~$ sudo crictl images

IMAGE                      TAG                 IMAGE ID            SIZE
docker.io/library/nginx    latest              08b152afcfae2       137MB
```

Lab 6.1. Container Operations with runc

Prior to being able to perform container operations with runc, we need to create a container in an OCI bundle format. We will use a busybox Docker container to export its filesystem in a tar archive, and use the extracted filesystem at as the rootfs for the runc container:

```
student@ubuntu:~$ mkdir -p runc-container/rootfs

student@ubuntu:~$ docker container export \
  $(docker container create busybox) \
  > busybox.tar

Unable to find image 'busybox:latest' locally
latest: Pulling from library/busybox
8ec32b265e94: Pulling fs layer
8ec32b265e94: Verifying Checksum
8ec32b265e94: Download complete
8ec32b265e94: Pull complete
Digest:
sha256:b37dd066f59a4961024cf4bed74cae5e68ac26b48807292bd12198afa3ecb778
Status: Downloaded newer image for busybox:latest


student@ubuntu:~$ tar -C runc-container/rootfs/ -xf busybox.tar

student@ubuntu:~$ cd runc-container/rootfs/

student@ubuntu:~/runc-container/rootfs$ ls

bin  dev  etc  home  proc  root  sys  tmp  usr  var
```

Aside from rootfs, runc requires a spec configuration file to start a container. Runc allows us to create a sample spec file:

```
student@ubuntu:~/runc-container/rootfs$ cd ..
```

```
student@ubuntu:~/runc-container$ runc spec

student@ubuntu:~/runc-container$ ls

config.json  rootfs
```

Display the content of the config.json file. Observe some of the sections in the file, such as **process**, **root**, and **namespaces**. The process section specifies a shell process that will run in a terminal as root (uid 0, gid 0). The root directory of the container is mapped, in a read-only mode, to the rootfs generated earlier. Namespaces lists all the namespaces that the container needs to have for the isolation of pid, network, ipc, hostname, and mount.

```
student@ubuntu:~/runc-container$ cat config.json

{
      "ociVersion": "1.0.2-dev",
      "process": {
            "terminal": true,
            "user": {
                  "uid": 0,
                  "gid": 0
            },
            "args": [
                  "sh"
            ],
            "env": [
"PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
                  "TERM=xterm"
            ],
            "cwd": "/",
            "capabilities": {
                  "bounding": [
                        "CAP_AUDIT_WRITE",
                        "CAP_KILL",
                        "CAP_NET_BIND_SERVICE"
                  ],
                  "effective": [
                        "CAP_AUDIT_WRITE",
                        "CAP_KILL",
                        "CAP_NET_BIND_SERVICE"
                  ],
                  "inheritable": [
                        "CAP_AUDIT_WRITE",
                        "CAP_KILL",
                        "CAP_NET_BIND_SERVICE"
```

```
                ],
                "permitted": [
                        "CAP_AUDIT_WRITE",
                        "CAP_KILL",
                        "CAP_NET_BIND_SERVICE"
                ],
                "ambient": [
                        "CAP_AUDIT_WRITE",
                        "CAP_KILL",
                        "CAP_NET_BIND_SERVICE"
                ]
        },
        "rlimits": [
                {
                        "type": "RLIMIT_NOFILE",
                        "hard": 1024,
                        "soft": 1024
                }
        ],
        "noNewPrivileges": true
},
"root": {
        "path": "rootfs",
        "readonly": true
},
"hostname": "runc",
"mounts": [
        {
                "destination": "/proc",
                "type": "proc",
                "source": "proc"
        },
        {
                "destination": "/dev",
                "type": "tmpfs",
                "source": "tmpfs",
                "options": [
                        "nosuid",
                        "strictatime",
                        "mode=755",
                        "size=65536k"
                ]
        },
        {
                "destination": "/dev/pts",
                "type": "devpts",
                "source": "devpts",
```

```
            "options": [
                "nosuid",
                "noexec",
                "newinstance",
                "ptmxmode=0666",
                "mode=0620",
                "gid=5"
            ]
        },
        {
            "destination": "/dev/shm",
            "type": "tmpfs",
            "source": "shm",
            "options": [
                "nosuid",
                "noexec",
                "nodev",
                "mode=1777",
                "size=65536k"
            ]
        },
        {
            "destination": "/dev/mqueue",
            "type": "mqueue",
            "source": "mqueue",
            "options": [
                "nosuid",
                "noexec",
                "nodev"
            ]
        },
        {
            "destination": "/sys",
            "type": "sysfs",
            "source": "sysfs",
            "options": [
                "nosuid",
                "noexec",
                "nodev",
                "ro"
            ]
        },
        {
            "destination": "/sys/fs/cgroup",
            "type": "cgroup",
            "source": "cgroup",
            "options": [
```

```
                    "nosuid",
                    "noexec",
                    "nodev",
                    "relatime",
                    "ro"
                ]
        }
    ],
    "linux": {
        "resources": {
            "devices": [
                {
                    "allow": false,
                    "access": "rwm"
                }
            ]
        },
        "namespaces": [
            {
                "type": "pid"
            },
            {
                "type": "network"
            },
            {
                "type": "ipc"
            },
            {
                "type": "uts"
            },
            {
                "type": "mount"
            }
        ],
        "maskedPaths": [
            "/proc/acpi",
            "/proc/asound",
            "/proc/kcore",
            "/proc/keys",
            "/proc/latency_stats",
            "/proc/timer_list",
            "/proc/timer_stats",
            "/proc/sched_debug",
            "/sys/firmware",
            "/proc/scsi"
        ],
        "readonlyPaths": [
```

```
                "/proc/bus",
                "/proc/fs",
                "/proc/irq",
                "/proc/sys",
                "/proc/sysrq-trigger"
            ]
        }
}
```

Now we are ready to run the container. Make sure to leave this terminal window as it is, with the running shell from the busybox container:

```
student@ubuntu:~/runc-container$ sudo runc run busybox

/ #
```

Open a second terminal, log in to your VM instance and list containers:

```
student@ubuntu:~$ sudo runc list

ID        PID     STATUS     BUNDLE                          CREATED
                            OWNER
busybox   18297   running    /home/student/runc-container
2021-08-10T08:52:58.030568873Z    root
```

Also from the second terminal, list the processes running inside the busybox container:

```
student@ubuntu:~$ sudo runc ps busybox

UID          PID    PPID  C STIME TTY          TIME CMD
root       18297  18287  0 08:52 pts/0    00:00:00 sh
```

Also from the second terminal, list the events of the busybox container. The command will keep outputting events until we stop the stream with **Ctrl + C**:

```
student@ubuntu:~$ sudo runc events busybox

{"type":"stats","id":"busybox","data":{"cpu":{"usage":{"total":21572876,"percp
u":[21572876],"percpu_kernel":[3559192],"percpu_user":[18013684],"kernel":0,"u
ser":20000000},"throttling":{}},"cpuset":{"cpus":[0],"cpu_exclusive":0,"mems":
[0],"mem_hardwall":0,"mem_exclusive":0,"memory_migrate":0,"memory_spread_page"
:0,"memory_spread_slab":0,"memory_pressure":0,"sched_load_balance":1,"sched_re
lax_domain_level":-1},"memory":{"usage":{"limit":9223372036854771712,"usage":6
```

```
06208,"max":6836224,"failcnt":0},"swap":{"limit":9223372036854771712,"usage":6
06208,"max":6836224,"failcnt":0},"kernel":
...
...
^C
```

Runc allows for a container to be paused and then resumed. From the second terminal list containers and issue the pause command, then list again to confirm the paused status. Return to the first terminal and try to type a command at the shell prompt – type **date**. No command will be registered or executed because the container is paused. Now return to the second terminal and resume the container, listing containers again to confirm running status. Return to the first terminal to see that the **date** command typed before is now displayed, together with the expected output.

```
student@ubuntu:~$ sudo runc list

ID       PID     STATUS     BUNDLE                          CREATED
                            OWNER
busybox  18297   running    /home/student/runc-container
2021-08-10T08:52:58.030568873Z    root


student@ubuntu:~$ sudo runc pause busybox

student@ubuntu:~$ sudo runc list

ID       PID     STATUS     BUNDLE                          CREATED
                            OWNER
busybox  18297   paused     /home/student/runc-container
2021-08-10T08:52:58.030568873Z    root


student@ubuntu:~$ sudo runc resume busybox

student@ubuntu:~$ sudo runc list

ID       PID     STATUS     BUNDLE                          CREATED
                            OWNER
busybox  18297   running    /home/student/runc-container
2021-08-10T08:52:58.030568873Z    root
```

At the first terminal window we can validate that the **date** command was executed:

```
/ # date
```

```
Tue Aug 10 12:24:44 UTC 2021
```

The container status can be displayed, again, from the second terminal window:

```
student@ubuntu:~$ sudo runc state busybox

{
  "ociVersion": "1.0.2-dev",
  "id": "busybox",
  "pid": 18297,
  "status": "running",
  "bundle": "/home/student/runc-container",
  "rootfs": "/home/student/runc-container/rootfs",
  "created": "2021-08-10T08:52:58.030568873Z",
  "owner": ""
}
```

There are a few methods to delete this container. From the second terminal, because it is running we need to supply the **-f** option to **force** the delete command. If the container were stopped, then **-f** would not be necessary. Another method would be to exit out of the shell running in the first terminal window, which would terminate the shell process running in the busybox container, and as a result the busybox container will be removed as well. We will run the **delete -f** command from the second terminal:

```
student@ubuntu:~$ sudo runc delete -f busybox

student@ubuntu:~$ sudo runc list

ID          PID         STATUS      BUNDLE      CREATED     OWNER
```

On the first terminal, the prompt has already returned to the **student@ubuntu**, since the runc container was deleted. No command is expected below, this is only shows the current prompt:

```
/ #
/ # student@ubuntu:~/runc-container$
```

Lab 6.2. Container Operations with Docker

Pull an image from the Docker Hub registry to the local repository, to start exploring container operations supported by Docker:

```
student@ubuntu:~$ docker image pull alpine

Using default tag: latest
latest: Pulling from library/alpine
Digest:
sha256:eb3e4e175ba6d212ba1d6e04fc0782916c08e1c9d7b45892e9796141b1d379ae
Status: Downloaded newer image for alpine:latest
docker.io/library/alpine:latest
```

List images available in the local repository. You may have other images displayed, but the output only focuses on the image we've just pulled:

```
student@ubuntu:~$ docker image ls
```

Or:

```
student@ubuntu:~$ docker images

REPOSITORY      TAG         IMAGE ID           CREATED         SIZE
alpine          latest      021b3423115f       2 weeks ago     5.6MB
...
```

Create a container from the image available in the local repository. This command does not start the container, it only creates it. Since we are not providing a name for the container, the runtime will assign a random name:

```
student@ubuntu:~$ docker container create -it alpine sh

2d16a24d2eed8154ce06a8db49c3f684b691e3704cdc244d3e4147beddf8b54a
```

Start the created container, using a partial container ID. The container will start the **sh** program, as we provided it as a **COMMAND** argument:

```
student@ubuntu:~$ docker container start 2d1

2d1
```

List running containers. Observe the random name assigned to the container (**eager_dubinsky** in the example below):

```
student@ubuntu:~$ docker container ls

CONTAINER ID   IMAGE     COMMAND     CREATED          STATUS
PORTS     NAMES
2d16a24d2eed   alpine    "sh"        2 minutes ago    Up 43 seconds
          eager_dubinsky
...
```

Running a container instead of creating & starting. The **-t** option allocates a pseudo-TTY, and the **-i** option keeps the STDIN open in interactive mode. Both **-i** and **-t** options can be combined into a **-it** or **-ti** notation, all with the same effect. The **--name** option allows the myalpine name to be assigned to the running container, instead of allowing the runtime to pick a random name for it.

```
student@ubuntu:~$ docker container run -it --name myalpine alpine sh

/ #
```

Detaching from a running container ensures the container remains running. By pressing the **Ctrl + p + Ctrl + q** key combination in the terminal of a running container:

```
/ # Ctrl + p + Ctrl + q
student@ubuntu:~$
```

The container is being detached, yet still running. Other methods to close the shell process in the terminal window would terminate the container as well. We can confirm the detached/running container by listing the running containers:

```
student@ubuntu:~$ docker container ls

CONTAINER ID   IMAGE   COMMAND  CREATED         STATUS     PORTS  NAMES
3c0d4b432a18   alpine  "sh"     3 minutes ago   Up 3 minutes      myalpine
2d16a24d2eed   alpine  "sh"     10 minutes ago  Up 7 minutes      eager_dubinsky
...
```

We can attach a running container. As a result we receive a shell into the container::

```
student@ubuntu:~$ docker container attach myalpine

/ #
```

We can run a container in the background with **-d** option, in which case we receive an output with the container ID. This example will run an infinite while-loop inside the container:

```
student@ubuntu:~$ docker container run -d alpine sh -c 'while [ 1 ]; do echo
"hello world from container"; sleep 1; done'

6c00df96a35d405b729d97d12006247d841f7a59eaccd45a3682b75f66d7b968
```

Display container logs:

```
student@ubuntu:~$ docker container logs 6c0

hello world from container
hello world from container
hello world from container
hello world from container
...
```

Stop a running container:

```
student@ubuntu:~$ docker container stop 6c0

6c0
```

List all containers (running and stopped):

```
student@ubuntu:~$ docker container ls -a
```

```
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS
PORTS       NAMES
6c00df96a35d   alpine    "sh -c 'while [ 1 ];…"   4 minutes ago    Exited (137)
About a minute ago
      gifted_perlman
...
```

Start a stopped container:

```
student@ubuntu:~$ docker container start 6c0

6c0
```

```
student@ubuntu:~$ docker container ls -a

CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS
PORTS       NAMES
6c00df96a35d   alpine    "sh -c 'while [ 1 ];…"   6 minutes ago    Up 33 seconds
            gifted_perlman
...
```

Restarting a container. This command stops and then starts a container, but it does not change the container ID or its name:

```
student@ubuntu:~$ docker container restart 6c0

6c0
```

Pausing and resuming a container:

```
student@ubuntu:~$ docker container pause 6c0

6c0
```

```
student@ubuntu:~$ docker container ls -a

CONTAINER ID  IMAGE    COMMAND                  CREATED          STATUS
                  PORTS      NAMES
6c00df96a35d  alpine   "sh -c 'while [ 1 ];…"  15 minutes ago   Up 2 minutes
(Paused)                   gifted_perlman
```

```
...

student@ubuntu:~$ docker container unpause 6c0

6c0

student@ubuntu:~$ docker container ls -a

CONTAINER ID   IMAGE    COMMAND                  CREATED          STATUS
PORTS       NAMES
6c00df96a35d   alpine   "sh -c 'while [ 1 ];…"   17 minutes ago   Up 4 minutes
            gifted_perlman
…
```

Renaming a running container. Let's rename the **6c00df96a35d** container, from **gifted_perlman** to **hello_world_loop**:

```
student@ubuntu:~$ docker container rename gifted_perlman hello_world_loop

student@ubuntu:~$ docker container ls -a

CONTAINER ID   IMAGE    COMMAND                  CREATED          STATUS
PORTS       NAMES
6c00df96a35d   alpine   "sh -c 'while [ 1 ];…"   19 minutes ago   Up 6 minutes
            hello_world_loop
...
```

Deleting or removing a container. There are two separate options available to remove containers. The default command that removes stopped containers, and a force option to remove running containers:

```
student@ubuntu:~$ docker container stop hello_world_loop

hello_world_loop

student@ubuntu:~$ docker container rm hello_world_loop

hello_world_loop

student@ubuntu:~$ docker container ls -a
```

```
CONTAINER ID   IMAGE     COMMAND   CREATED          STATUS     PORTS   NAMES
3c0d4b432a18   alpine    "sh"      47 minutes ago   Up 47 minutes      myalpine
...

student@ubuntu:~$ docker container rm -f myalpine

myalpine

student@ubuntu:~$ docker container ls -a

CONTAINER ID   IMAGE     COMMAND   CREATED          STATUS     PORTS   NAMES
2d16a24d2eed   alpine    "sh"      54 minutes ago   Up 51 minutes      eager_dubinsky
...
```

Automatically remove a container upon its exit. In order to automate the removal process, to avoid repetitive tasks to manually find terminated containers and manually remove them, we can automate the removal process by passing the **--rm** option when we run the container. As a result, we will no longer see terminated containers in their stopped state:

```
student@ubuntu:~$ docker container run --rm --name auto_rm alpine ping -c 3
google.com

PING google.com (142.250.136.113): 56 data bytes
64 bytes from 142.250.136.113: seq=0 ttl=114 time=1.254 ms
64 bytes from 142.250.136.113: seq=1 ttl=114 time=1.499 ms
64 bytes from 142.250.136.113: seq=2 ttl=114 time=1.268 ms

--- google.com ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 1.254/1.340/1.499 ms

student@ubuntu:~$ docker container ls -a

CONTAINER ID   IMAGE     COMMAND   CREATED          STATUS     PORTS   NAMES
2d16a24d2eed   alpine    "sh"      56 minutes ago   Up 53 minutes      eager_dubinsky
...
```

Setting the hostname of a container. Unless we explicitly set the hostname of a container with **-h** option, by default, at runtime, the container hostname is set to the container ID:

```
student@ubuntu:~$ docker container run -h alpine-host -it --rm alpine sh
```

```
/ # hostname
alpine-host
/ # exit
```

Set the current working directory with **-w** option, of a container:

```
student@ubuntu:~$ docker container run -it -w /tmp/mypath --rm alpine sh

/tmp/mypath # pwd
/tmp/mypath
/tmp/mypath # exit
```

Set an environment variable of a container and assign a value to it. We are setting the WEB_HOST environment variable of the container, and assign an IP address to it:

```
student@ubuntu:~$ docker container run -it --env "WEB_HOST=172.168.1.1" --rm
alpine sh

/ # env
HOSTNAME=9b9f7a458286
SHLVL=1
HOME=/root
TERM=xterm
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
PWD=/
WEB_HOST=172.168.1.1
/ # exit
```

Set the ulimit of a container. **Ulimit** is a command line tool to manage resource limits for users. It returns current limits for the user, but it can also set such resource limits. Let's display all the default limits on a new alpine container:

```
student@ubuntu:~$ docker container run -it --rm alpine sh

/ # ulimit -a
core file size (blocks)         (-c) unlimited
data seg size (kb)              (-d) unlimited
scheduling priority             (-e) 0
file size (blocks)              (-f) unlimited
pending signals                 (-i) 2264
max locked memory (kb)          (-l) 64
max memory size (kb)            (-m) unlimited
open files                      (-n) 1048576
POSIX message queues (bytes)    (-q) 819200
```

```
real-time priority              (-r) 0
stack size (kb)                 (-s) 8192
cpu time (seconds)              (-t) unlimited
max user processes              (-u) unlimited
virtual memory (kb)             (-v) unlimited
file locks                      (-x) unlimited
/ # exit
```

By default, user processes are unlimited. Let's try to limit the max user processes. By setting a limit we restrict the number of processes this container can create:

```
student@ubuntu:~$ docker container run -it --ulimit nproc=10 --rm alpine sh

/ # ulimit -a
core file size (blocks)         (-c) unlimited
data seg size (kb)              (-d) unlimited
scheduling priority             (-e) 0
file size (blocks)              (-f) unlimited
pending signals                 (-i) 2264
max locked memory (kb)          (-l) 64
max memory size (kb)            (-m) unlimited
open files                      (-n) 1048576
POSIX message queues (bytes)    (-q) 819200
real-time priority              (-r) 0
stack size (kb)                 (-s) 8192
cpu time (seconds)              (-t) unlimited
max user processes              (-u) 10
virtual memory (kb)             (-v) unlimited
file locks                      (-x) unlimited
/ # exit
```

Display all the details of a container, such as hostname, IP address, attached volumes, image, and network configuration:

```
student@ubuntu:~$ docker container ls

CONTAINER ID   IMAGE    COMMAND   CREATED         STATUS    PORTS   NAMES
2d16a24d2eed   alpine   "sh"      58 minutes ago  Up 55 minutes   eager_dubinsky
...


student@ubuntu:~$ docker container inspect 2d16a24d2eed

[
    {
```

```
        "Id":
"2d16a24d2eed8154ce06a8db49c3f684b691e3704cdc244d3e4147beddf8b54a",
        "Created": "2021-08-10T12:59:40.469499338Z",
        "Path": "sh",
        "Args": [],
        "State": {
            "Status": "running",
            "Running": true,
            "Paused": false,
            "Restarting": false,
            "OOMKilled": false,
            "Dead": false,
            "Pid": 18712,
            "ExitCode": 0,
            "Error": "",
            "StartedAt": "2021-08-10T13:02:43.570606305Z",
            "FinishedAt": "0001-01-01T00:00:00Z"
        },
        "Image":
"sha256:021b3423115ff662225e83d7e2606475217de7b55fde83ce3447a54019a77aa2",
        "ResolvConfPath":
"/var/lib/docker/containers/2d16a24d2eed8154ce06a8db49c3f684b691e3704cdc244d3e
4147beddf8b54a/resolv.conf",
        "HostnamePath":
"/var/lib/docker/containers/2d16a24d2eed8154ce06a8db49c3f684b691e3704cdc244d3e
4147beddf8b54a/hostname",
        "HostsPath":
"/var/lib/docker/containers/2d16a24d2eed8154ce06a8db49c3f684b691e3704cdc244d3e
4147beddf8b54a/hosts",
        "LogPath":
"/var/lib/docker/containers/2d16a24d2eed8154ce06a8db49c3f684b691e3704cdc244d3e
4147beddf8b54a/2d16a24d2eed8154ce06a8db49c3f684b691e3704cdc244d3e4147beddf8b54
a-json.log",
        "Name": "/eager_dubinsky",
        "RestartCount": 0,
        "Driver": "overlay2",
        "Platform": "linux",
        "MountLabel": "",
        "ProcessLabel": "",
        "AppArmorProfile": "docker-default",
        "ExecIDs": null,
        "HostConfig": {
            "Binds": null,
            "ContainerIDFile": "",
            "LogConfig": {
                "Type": "json-file",
                "Config": {}
```

```
},
"NetworkMode": "default",
"PortBindings": {},
"RestartPolicy": {
    "Name": "no",
    "MaximumRetryCount": 0
},
"AutoRemove": false,
"VolumeDriver": "",
"VolumesFrom": null,
"CapAdd": null,
"CapDrop": null,
"CgroupnsMode": "host",
"Dns": [],
"DnsOptions": [],
"DnsSearch": [],
"ExtraHosts": null,
"GroupAdd": null,
"IpcMode": "private",
"Cgroup": "",
"Links": null,
"OomScoreAdj": 0,
"PidMode": "",
"Privileged": false,
"PublishAllPorts": false,
"ReadonlyRootfs": false,
"SecurityOpt": null,
"UTSMode": "",
"UsernsMode": "",
"ShmSize": 67108864,
"Runtime": "runc",
"ConsoleSize": [
    0,
    0
],
"Isolation": "",
"CpuShares": 0,
"Memory": 0,
"NanoCpus": 0,
"CgroupParent": "",
"BlkioWeight": 0,
"BlkioWeightDevice": [],
"BlkioDeviceReadBps": null,
"BlkioDeviceWriteBps": null,
"BlkioDeviceReadIOps": null,
"BlkioDeviceWriteIOps": null,
"CpuPeriod": 0,
```

```
            "CpuQuota": 0,
            "CpuRealtimePeriod": 0,
            "CpuRealtimeRuntime": 0,
            "CpusetCpus": "",
            "CpusetMems": "",
            "Devices": [],
            "DeviceCgroupRules": null,
            "DeviceRequests": null,
            "KernelMemory": 0,
            "KernelMemoryTCP": 0,
            "MemoryReservation": 0,
            "MemorySwap": 0,
            "MemorySwappiness": null,
            "OomKillDisable": false,
            "PidsLimit": null,
            "Ulimits": null,
            "CpuCount": 0,
            "CpuPercent": 0,
            "IOMaximumIOps": 0,
            "IOMaximumBandwidth": 0,
            "MaskedPaths": [
                "/proc/asound",
                "/proc/acpi",
                "/proc/kcore",
                "/proc/keys",
                "/proc/latency_stats",
                "/proc/timer_list",
                "/proc/timer_stats",
                "/proc/sched_debug",
                "/proc/scsi",
                "/sys/firmware"
            ],
            "ReadonlyPaths": [
                "/proc/bus",
                "/proc/fs",
                "/proc/irq",
                "/proc/sys",
                "/proc/sysrq-trigger"
            ]
        },
        "GraphDriver": {
            "Data": {
                "LowerDir":
"/var/lib/docker/overlay2/42127082d467bb3b5221fa5a70fe5d86aa568cbf922efa357197
6a74785325a6-init/diff:/var/lib/docker/overlay2/6bbabbf0f44c4f7955180c54680093
f02d67da20463d903e598196b6cc35503b/diff",
```

```
                "MergedDir":
"/var/lib/docker/overlay2/42127082d467bb3b5221fa5a70fe5d86aa568cbf922efa357197
6a74785325a6/merged",
                "UpperDir":
"/var/lib/docker/overlay2/42127082d467bb3b5221fa5a70fe5d86aa568cbf922efa357197
6a74785325a6/diff",
                "WorkDir":
"/var/lib/docker/overlay2/42127082d467bb3b5221fa5a70fe5d86aa568cbf922efa357197
6a74785325a6/work"
            },
            "Name": "overlay2"
        },
        "Mounts": [],
        "Config": {
            "Hostname": "2d16a24d2eed",
            "Domainname": "",
            "User": "",
            "AttachStdin": true,
            "AttachStdout": true,
            "AttachStderr": true,
            "Tty": true,
            "OpenStdin": true,
            "StdinOnce": true,
            "Env": [

"PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
            ],
            "Cmd": [
                "sh"
            ],
            "Image": "alpine",
            "Volumes": null,
            "WorkingDir": "",
            "Entrypoint": null,
            "OnBuild": null,
            "Labels": {}
        },
        "NetworkSettings": {
            "Bridge": "",
            "SandboxID":
"8f7de6ca875a3d47545aef1dba02572d0527a756b4ea31274365134c4a7850aa",
            "HairpinMode": false,
            "LinkLocalIPv6Address": "",
            "LinkLocalIPv6PrefixLen": 0,
            "Ports": {},
            "SandboxKey": "/var/run/docker/netns/8f7de6ca875a",
            "SecondaryIPAddresses": null,
```

```
            "SecondaryIPv6Addresses": null,
            "EndpointID":
"36ea2cafb9d7a3f13873e511438f9bd4a507eeb411ac132f5c2ef63099aef6c1",
            "Gateway": "172.17.0.1",
            "GlobalIPv6Address": "",
            "GlobalIPv6PrefixLen": 0,
            "IPAddress": "172.17.0.3",
            "IPPrefixLen": 16,
            "IPv6Gateway": "",
            "MacAddress": "02:42:ac:11:00:03",
            "Networks": {
                "bridge": {
                    "IPAMConfig": null,
                    "Links": null,
                    "Aliases": null,
                    "NetworkID":
"08ef3eb633e6688ded6109d6edad5d930f49a14d484becd82519d354b1222dc4",
                    "EndpointID":
"36ea2cafb9d7a3f13873e511438f9bd4a507eeb411ac132f5c2ef63099aef6c1",
                    "Gateway": "172.17.0.1",
                    "IPAddress": "172.17.0.3",
                    "IPPrefixLen": 16,
                    "IPv6Gateway": "",
                    "GlobalIPv6Address": "",
                    "GlobalIPv6PrefixLen": 0,
                    "MacAddress": "02:42:ac:11:00:03",
                    "DriverOpts": null
                }
            }
        }
    }
]
```

Restrict the host CPU(s) that are allowed to execute a container. We can set a single CPU, or a range of CPUs that are allowed to execute the container. In the previous full output of the inspect command, there were no such restrictions. In this exercise let's restrict the container to be executed only by CPU "0":

```
student@ubuntu:~$ docker container run -d --name cpu-set --cpuset-cpus="0"
alpine top

0504b3bb7ddf9943ee207396fdce92b48adc5d37e6f78d2a0903ddd32985783f
```

Let's verify the new setting by inspecting the container:

```
student@ubuntu:~$ docker container inspect cpu-set | grep -i cpuset
```

```
            "CpusetCpus": "0",
            "CpusetMems": "",
```

Now it is safe to remove the container:

```
student@ubuntu:~$ docker container rm -f cpu-set

cpu-set
```

We can also set the amount of memory of a container. By default, from the previous full output of the inspect command, the value is set to “0”. If the VM is provisioned on a local hypervisor such as Virtualbox, you may see a warning message that can be ignored. Otherwise the command will only return the container ID. Let’s reset that value:

```
student@ubuntu:~$ docker container run -d --name memory --memory "200m" alpine
top

WARNING: Your kernel does not support swap limit capabilities or the cgroup is
not mounted. Memory limited without swap.
86ef309433bba0aafe73fed5dcf97f483243da70fd5bb4a45f2c959a6a41bdfa


student@ubuntu:~$ docker container inspect memory | grep -i mem

        "Name": "/memory",
            "Memory": 209715200,
            "CpusetMems": "",
            ...
```

Now let’s remove the container:

```
student@ubuntu:~$ docker container rm -f memory

memory
```

Create a new process inside a running container, a feature very useful for debugging. Let’s execute a new process inside a container by running a command that retrieves and lists the IP address of the container. As soon as the command finishes, the newly forked process also gets terminated:

```
student@ubuntu:~$ docker container ls

CONTAINER ID   IMAGE    COMMAND   CREATED         STATUS    PORTS   NAMES
2d16a24d2eed   alpine   "sh"      1 hour ago      Up 1 hour         eager_dubinsky
...
```

```
student@ubuntu:~$ docker container exec eager_dubinsky ip a

1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
7: eth0@if8: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
state UP
    link/ether 02:42:ac:11:00:03 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.3/16 brd 172.17.255.255 scope global eth0
       valid_lft forever preferred_lft forever
```

Set the restart policy of a container. An always restart policy restarts the container every time it fails. An on-failure policy, however, allows us to control the number of restarts of the container as a result of several failures - set to 3 in the example below:

```
student@ubuntu:~$ docker container run -d --restart=always --name web-always
nginx

04ba201f0c62e1b14f71dab71c00879e7886fd4592bf5e889f5fae5da63f6343


student@ubuntu:~$ docker container run -d --restart=on-failure:3 --name
web-on-failure nginx

8fd91ca8c73355f13eca1ed92bc3d96e3b7f522034e33fc4f015cf4443e5c976
```

We can copy files between the host system and a running container. This example will overwrite the index.html file of the nginx webserver running in a container, and the verification step will include the display of the container IP and finally a curl command to display the new web page served by the webserve:

```
student@ubuntu:~$ echo Welcome to Container Fundamentals! > host-file

student@ubuntu:~$ docker container cp host-file
web-on-failure:/usr/share/nginx/html/index.html

student@ubuntu:~$ docker container inspect --format='{{range
.NetworkSettings.Networks}}{{.IPAddress}}{{end}}' web-on-failure

172.17.0.7
```

```
student@ubuntu:~$ curl 172.17.0.7

Welcome to Container Fundamentals!
```

Labeling a container:

```
student@ubuntu:~$ docker container run -d --label env=dev nginx

82e9bd095d83455c7cf9aa0166c703afbc9a57887e213f58d9a2c1610ad87c04
```

Filtering container lists. We can filter containers by specifying conditions, to control and limit the output only to the desired objects. In this example let's use the label created previously as a filter:

```
student@ubuntu:~$ docker container ls --filter label=env=dev

CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS               NAMES
82e9bd095d83        nginx               "/docker-entrypoint.…"   6 minutes ago
Up 6 minutes        80/tcp              dazzling_northcutt
```

Remove/delete all (running and stopped) containers with one command. But first, let's explore the **-q** option of the **ls** command that lists only the container IDs:

```
student@ubuntu:~$ docker container ls -q

82e9bd095d83
8fd91ca8c733
04ba201f0c62
b366d6cfcb74
9e09b2cf6ff3
b1f1f19b1735
124b392d4bf2
```

```
student@ubuntu:~$ docker container rm -f `docker container ls -q`

82e9bd095d83
8fd91ca8c733
04ba201f0c62
b366d6cfcb74
9e09b2cf6ff3
b1f1f19b1735
124b392d4bf2
```

Change the default executable that runs at container startup. If defined, a default executable runs when a container starts. A nginx container starts with **/usr/sbin/nginx -g daemon off**. We can change it by passing another command with possible arguments when running the container. Let's start a container from the nginx image, but with a running shell instead:

```
student@ubuntu:~$ docker container run -it nginx sh

#
```

Privileged containers. In privileged mode, containers gain permissions to access devices on the host. By default, it is disabled and containers run in un-privileged mode. Let's demonstrate privileges by running two containers in un-privileged and privileged mode respectively, while attempting to change host network settings, that is to create a simple alias to a network device:

```
student@ubuntu:~$ docker container run -it --net=host alpine sh

/ # ifconfig ens4:0 192.168.2.1 up
ifconfig: SIOCSIFADDR: Operation not permitted
/ #


student@ubuntu:~$ docker container run -it --net=host --privileged alpine sh

/ # ifconfig ens4:0 192.168.2.1 up
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: ens4: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1460 qdisc fq_codel state UP
qlen 1000
    link/ether 42:01:0a:80:00:05 brd ff:ff:ff:ff:ff:ff
    inet 10.128.0.5/32 scope global dynamic ens4
       valid_lft 1844sec preferred_lft 1844sec
    inet 192.168.2.1/24 brd 192.168.2.255 scope global ens4:0
       valid_lft forever preferred_lft forever
    inet6 fe80::4001:aff:fe80:5/64 scope link
       valid_lft forever preferred_lft forever
3: lxcbr0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state
DOWN qlen 1000
    link/ether 00:16:3e:00:00:00 brd ff:ff:ff:ff:ff:ff
    inet 10.0.3.1/24 scope global lxcbr0
       valid_lft forever preferred_lft forever
```

```
    inet6 fe80::216:3eff:fe00:0/64 scope link
       valid_lft forever preferred_lft forever
8: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state
DOWN
    link/ether 02:42:93:77:3b:cd brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:93ff:fe77:3bcd/64 scope link
       valid_lft forever preferred_lft forever
/ #
```

Remove all stopped containers with **prune**, typing **y** (yes) as confirmation when prompted:

```
student@ubuntu:~$ docker container ls

CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES


student@ubuntu:~$ docker container ls -a

CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS                      PORTS               NAMES
3f639a5fceae        alpine              "sh"                6 minutes ago
Exited (0) 11 seconds ago                       gracious_moser
9a050a479a9d        alpine              "sh"                7 minutes ago
Exited (1) 6 minutes ago                        serene_chatelet
fd26346e8a95        alpine              "sh"                17 hours ago
Exited (0) 17 hours ago                         sweet_booth
32f7643ef139        busybox             "sh"                19 hours ago
Created                                         recursing_ganguly
8a6c65920f97        hello-world         "/hello"            3 days ago
Exited (0) 3 days ago                           nice_clarke


student@ubuntu:~$ docker container prune

WARNING! This will remove all stopped containers.
Are you sure you want to continue? [y/N] y
Deleted Containers:
3f639a5fceaeb8e58cb291b18985071db67920c1e21699440f0db0e4b9c9e14e
9a050a479a9db355e9b6759811b7248906618ef647f26be5a60572187747cc53
fd26346e8a9515d12d91e8127b4639e07f00f3b62ea03511ef264e263f2139b2
32f7643ef139852ef4dc8c20db526e8cd7eb12c499af51c3dbeae8ccd56e74ad
8a6c65920f9789e0e83a7185c735b0f94e7b18a45e2cf626a1a950647f17afb3
```

```
Total reclaimed space: 118B

student@ubuntu:~$ docker container ls -a
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
```

Lab 6.3. Container Operations with Podman

Let's create a container, which will also trigger an image **fetch** if the desired image is not found in the local image cache. The container will remain in a created state until we explicitly **start** the container, when it becomes a running container. Since we are not providing a name for the new container, the runtime will randomly pick a name for it (**eloquent_rhodes** in the example below). Then we list containers to see whether the container is running:

```
student@ubuntu:~$ podman container create -it alpine sh

Resolved "alpine" as an alias
(/etc/containers/registries.conf.d/000-shortnames.conf)
Trying to pull docker.io/library/alpine:latest...
Getting image source signatures
Copying blob 29291e31a76a skipped: already exists
Copying config 021b342311 done
Writing manifest to image destination
Storing signatures
e715cf93100545bb6749ce3e990cc7b9d96c0ad7b6599c23725f478f9ca57154


student@ubuntu:~$ podman container ls

CONTAINER ID  IMAGE     COMMAND     CREATED     STATUS      PORTS       NAMES



student@ubuntu:~$ podman container ls -a

CONTAINER ID  IMAGE                            COMMAND    CREATED
STATUS      PORTS       NAMES
e715cf931005  docker.io/library/alpine:latest  sh         3 minutes ago
Created                 eloquent_rhodes
```

Now let's start the container and list containers again:

```
student@ubuntu:~$ podman container start e71

e71


student@ubuntu:~$ podman container ls

CONTAINER ID  IMAGE                            COMMAND    CREATED         STATUS
            PORTS       NAMES
e715cf931005  docker.io/library/alpine:latest  sh         4 minutes ago   Up 10
seconds ago               eloquent_rhodes
```

Let's attach to the running container, and validate its hostname, which by default is the container ID. List the container's processes and detach from the container with the **Ctrl + p + Ctrl + q** combination:

```
student@ubuntu:~$ podman container attach e71

/ # hostname
e715cf931005
/ # ps
PID   USER     TIME  COMMAND
    1 root      0:00 sh
    3 root      0:00 ps
/ # Ctrl + p + Ctrl + q
```

A container can be inspected to list its configuration details:

```
student@ubuntu:~$ podman container inspect  alpine sh

[
    {
        "Id":
"e715cf93100545bb6749ce3e990cc7b9d96c0ad7b6599c23725f478f9ca57154",
        "Created": "2021-08-10T19:33:50.271562025Z",
        "Path": "sh",
        "Args": [
            "sh"
        ],
        "State": {
```

```
            "OciVersion": "1.0.2-dev",
            "Status": "running",
            "Running": true,
            "Paused": false,
            "Restarting": false,
            "OOMKilled": false,
            "Dead": false,
            "Pid": 23110,
            "ConmonPid": 23107,
            "ExitCode": 0,
            "Error": "",
            "StartedAt": "2021-08-10T19:38:37.089357793Z",
            "FinishedAt": "0001-01-01T00:00:00Z",
            "Healthcheck": {
                "Status": "",
                "FailingStreak": 0,
                "Log": null
            }
        },
        "Image":
"021b3423115ff662225e83d7e2606475217de7b55fde83ce3447a54019a77aa2",
        "ImageName": "docker.io/library/alpine:latest",
        "Rootfs": "",
        "Pod": "",
...
```

A simplified method to run a container is the **run** command. It incorporates a **fetch** if needed, a **create** and a **start**. Let's run a container and manually set its hostname with **-h**. The container will be removed/deleted automatically once it exits **--rm**:

```
student@ubuntu:~$ podman container run -h alpine-host -it --rm alpine sh

/ # hostname
alpine-host
/ # exit
```

Let's run a container with an infinite while-loop, output its logs, then stop and start the container:

```
student@ubuntu:~$ podman container run -d alpine sh -c 'while [ 1 ]; do echo
"hello world from container"; sleep 1; done'
```

```
28590f6b972f0897d234db05e873258ee0a0676e3f199ca6a024ed51cef34007


student@ubuntu:~$ podman container logs 285

hello world from container
hello world from container
hello world from container
hello world from container
...


student@ubuntu:~$ podman container ls

CONTAINER ID  IMAGE                                COMMAND               CREATED
              STATUS          PORTS       NAMES
28590f6b972f  docker.io/library/alpine:latest  sh -c while [ 1 ]...  34
seconds ago   Up 35 seconds ago           eager_neumann
...


student@ubuntu:~$ podman container stop 285

285



student@ubuntu:~$ podman container ls

CONTAINER ID  IMAGE     COMMAND     CREATED       STATUS     PORTS       NAMES
...



student@ubuntu:~$ podman container start 285

285



student@ubuntu:~$ podman container ls

CONTAINER ID  IMAGE                                COMMAND               CREATED
            STATUS          PORTS       NAMES
```

```
28590f6b972f  docker.io/library/alpine:latest  sh -c while [ 1 ]...  About a
minute ago  Up 3 seconds ago            eager_neumann
...
```

Let's run a container that exits after a set number of ping executions, and it is automatically removed once exited:

```
student@ubuntu:~$ podman container run --rm --name auto_rm alpine ping -c 3
google.com

PING google.com (142.250.152.139): 56 data bytes
64 bytes from 142.250.152.139: seq=0 ttl=42 time=1.425 ms
64 bytes from 142.250.152.139: seq=1 ttl=42 time=1.241 ms
64 bytes from 142.250.152.139: seq=2 ttl=42 time=1.051 ms

--- google.com ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 1.051/1.239/1.425 ms
```

Now let's force the removal of a running container with the **-f** option:

```
student@ubuntu:~$ podman container rm -f 285

28590f6b972f0897d234db05e873258ee0a0676e3f199ca6a024ed51cef34007
```

Exec allows for commands to be run inside a running container. Running a shell in the container allows users to directly interact with the container environment. Exec can run non-interactive mode and in interactive terminal mode with the **-ti** options. We can also run installers, validators, display the environment or a set of permissions from the container. Let's copy content from the host system into a running container, and validate the content:

```
student@ubuntu:~$ echo Welcome to Container Fundamentals! > host-file

student@ubuntu:~$ podman container run -d --name host-content nginx

65fa90715e592502e5723bac7c9f3e380d08745a45df136b0352be37ec30e037


student@ubuntu:~$ podman container cp host-file
host-content:/usr/share/nginx/html/index.html
```

```
student@ubuntu:~$ podman container exec host-content cat
/usr/share/nginx/html/index.html

Welcome to Container Fundamentals!


student@ubuntu:~$ podman container exec -ti host-content sh

# cat /usr/share/nginx/html/index.html
Welcome to Container Fundamentals!
# exit
```

Let's remove all non-running containers with **prune**:

```
student@ubuntu:~$ podman container ls -aq

361b397125c8
65fa90715e59
f691cc036ca6
6353ae36f5fb
8f6e9a9663c2
8f1bfb121512
feb404020c56


student@ubuntu:~$ podman container prune

WARNING! This will remove all non running containers.
Are you sure you want to continue? [y/N] y
361b397125c88be524f439ef320078f40a196a33befde61cceddf2a9da77db17
6353ae36f5fb878fb849dc2515a28da413e7513513b2075fe15573aedf04b20b
65fa90715e592502e5723bac7c9f3e380d08745a45df136b0352be37ec30e037
8f1bfb12151256df65771b1e962a39ecb0b116fa752bc2a5ebfc033406b6be9c
8f6e9a9663c2627362b85fdc7cfe739c0b9375ec2b861e58aba455e2857fdff4
f691cc036ca6bfe6507b68e893f8661bc836e8e686d1b0e85e7ee388edc1a540
feb404020c560e758e6bcf6a7d93e4b2574067f02efe5398f7209fe3b0b4a3b4


student@ubuntu:~$ podman container ls -aq
```

Lab 6.4. Container Operations with crictl

Performing container operations with **crictl** is not as straightforward as we have seen earlier with docker and podman. Both docker and podman are user oriented and offer commands that are intuitive, easy to customize and adapt, while abstracting runtime operations, so that the user's focus is on the container configuration and not on the runtime itself. Therefore, we can simply **create** or **run** a container with docker and podman, and the runtime operations are automated under the hood, ...

Runtimes such as containerd and CRI-O do not come with dedicated command line utilities, as they have been designed to be used in complex environments, by container orchestration tools equipped with sophisticated interfaces to interact with them. Without dedicated command line utilities, it becomes difficult to troubleshoot runtime related issues if users cannot interact with the runtime itself, even at a higher level.

The solution is **crictl**, a high level command line utility, that can be customized to use either containerd, CRI-O, or Docker as its backend runtime. Although it seems to be as intuitive as the earlier tools, with supported container **create** and **run** commands, among others, crictl uses a more complex approach when running containers, by requiring the user to also define an isolated sandbox for the container to be run. This sandbox is called a **pod**, a concept widely used by the Kubernetes container orchestrator, Podman, and even the retired rkt.

This lab exercise aims to introduce us to specific steps required by crictl in conjunction with CRI-O runtime to run containers, exec, stop, and remove. As both projects are still maturing, some of the steps presented here may require extra care to avoid possible runtime loads that may prevent some features from behaving as expected.

Keep in mind that the crictl command needs to run with **sudo**, and the crio service needs to be running as well. In case the crio service is not running or its features stop working and the runtime becomes unresponsive, make sure to start or restart the crio service with **sudo systemctl start crio.service**, or **restart** if required.

Let's first configure the pod sandbox, and the container. The container will run a busybox image, which crictl will pull from the registry if not found in the local image cache. The busybox container will run a

sleep command for **600** seconds, keeping the container in a running state for 10 minutes before its state changes to exited.

```
student@ubuntu:~$ vim pod-config.json

{
    "metadata": {
        "name": "busybox-sandbox",
        "namespace": "default",
        "attempt": 1,
        "uid": "hdishd83djaidwnduwk28bcsb"
    },
    "log_directory": "/tmp",
    "linux": {
    }
}
```

```
student@ubuntu:~$ vim container-config.json

{
  "metadata": {
      "name": "busybox"
  },
  "image":{
      "image": "busybox"
  },
  "command": [
      "sleep"
  ],
  "args": [
      "600"
  ],
  "log_path":"busybox.0.log",
  "linux": {
  }
}
```

The **run** command takes as arguments the two config files, and it returns the container ID upon its start. The **ps** command displays containers in running state, the container ID, pod ID, image, age, and the container name.

```
student@ubuntu:~$ sudo crictl run container-config.json pod-config.json

342ba50c7230433e79e0553c3c0c60ea7c42be2fb6cf9d1779303a91c80d9781
```

```
student@ubuntu:~$ sudo crictl ps

CONTAINER           IMAGE               CREATED             STATE
NAME                ATTEMPT             POD ID
342ba50c72304       busybox             11 seconds ago      Running
busybox             0                   cee737728cc0f
```

The exec command allows us to run commands inside the container in a non-interactive fashion (**ps** and then **hostname**) and then interactive via a terminal. We will notice that by default, the hostname of the container is the pod ID, and not the container ID as we have seen with Docker and Podman.

```
student@ubuntu:~$ sudo crictl exec 342 ps

PID   USER     TIME  COMMAND
    1 root      0:00 /pause
    6 root      0:00 sleep 600
   11 root      0:00 ps
```

```
student@ubuntu:~$ sudo crictl exec 342 hostname

cee737728cc0
```

```
student@ubuntu:~$ sudo crictl exec -ti 342 sh

/ # ps
PID   USER     TIME  COMMAND
    1 root      0:00 /pause
    6 root      0:00 sleep 600
   26 root      0:00 sh
   31 root      0:00 ps
/ # hostname
cee737728cc0
/ # exit
```

We can display the container's details, such as image information, runtime, command with arguments, environment, capabilities, mount points, permissions, and resources:

```
student@ubuntu:~$ sudo crictl inspect 342

{
  "status": {
    "id": "342ba50c7230433e79e0553c3c0c60ea7c42be2fb6cf9d1779303a91c80d9781",
    "metadata": {
      "attempt": 0,
      "name": "busybox"
    },
    "state": "CONTAINER_EXITED",
    "createdAt": "2021-08-10T22:19:33.186094318Z",
    "startedAt": "2021-08-10T22:19:33.21895633Z",
    "finishedAt": "2021-08-10T22:29:33.223215249Z",
    "exitCode": 0,
    "image": {
      "annotations": {},
      "image": "docker.io/library/busybox:latest"
    },
    "imageRef":
"docker.io/library/busybox@sha256:b37dd066f59a4961024cf4bed74cae5e68ac26b48807
292bd12198afa3ecb778",
    "reason": "Completed",
    "message": "",
    "labels": {},
    "annotations": {},
    "mounts": [],
    "logPath": "/tmp/busybox.0.log"
  },
  "info": {
    "sandboxID":
"cee737728cc0fc9fa77f7b91a1d8e1949463f396318fb0641704b0cfd2817d95",
    "pid": 0,
    "runtimeSpec": {
      "ociVersion": "1.0.2-dev",
      "process": {
        "user": {
          "uid": 0,
          "gid": 0,
          "additionalGids": [
            10
          ]
        },
        "args": [
          "sleep",
```

```
          "600"
        ],
...
```

We can also display pod's details, such as its status, IP address, pause container image used by the runtime to set up and configure the pod sandbox resources, runtime, environment, capabilities, and mounts:

```
student@ubuntu:~$ sudo crictl inspectp cee

{
  "status": {
    "id": "cee737728cc0fc9fa77f7b91a1d8e1949463f396318fb0641704b0cfd2817d95",
    "metadata": {
      "attempt": 1,
      "name": "busybox-sandbox",
      "namespace": "default",
      "uid": "hdishd83djaidwnduwk28bcsb"
    },
    "state": "SANDBOX_READY",
    "createdAt": "2021-08-10T22:19:30.439911829Z",
    "network": {
      "additionalIps": [
        {
          "ip": "1100:200::f"
        }
      ],
      "ip": "10.85.0.15"
    },
...
```

Prior to continuing with the following steps, we need to remove the container and the pod to reclaim resources. Before their removal we need to ensure that the container and the pod are stopped respectively. First we will stop and remove the container, then we will do the same with the pod, listing resources for validation.

```
student@ubuntu:~$ sudo crictl stop 342

342
```

```
student@ubuntu:~$ sudo crictl ps -a
```

```
CONTAINER           IMAGE               CREATED             STATE
NAME                ATTEMPT             POD ID
342ba50c72304       busybox             5 minutes ago       Exited
busybox             0                   cee737728cc0
```

```
student@ubuntu:~$ sudo crictl rm 342

342
```

```
student@ubuntu:~$ sudo crictl pods

POD ID              CREATED             STATE               NAME
NAMESPACE           ATTEMPT             RUNTIME
cee737728cc0        5 minutes ago       Ready               busybox-sandbox
default             1                   (default)
```

```
student@ubuntu:~$ sudo crictl stopp cee

Stopped sandbox cee
```

```
student@ubuntu:~$ sudo crictl rmp cee

Removed sandbox cee
```

```
student@ubuntu:~$ sudo crictl pods

POD ID              CREATED             STATE               NAME
NAMESPACE           ATTEMPT             RUNTIME
```

Let's reconfigure the pod sandbox, and the container. The container will run a busybox image, which crictl will pull from the registry if not found in the local image cache. The busybox container will run a single **echo** command, keeping the container in a running state only for a brief moment, then transitioning in an exited state.

```
student@ubuntu:~$ vim pod-config.json

{
    "metadata": {
        "name": "busybox1-sandbox",
        "namespace": "default",
        "attempt": 1,
        "uid": "hdishd83djaidwnduwk28bcsb"
    },
    "log_directory": "/tmp",
    "linux": {
    }
}
```

```
student@ubuntu:~$ vim container-config.json

{
  "metadata": {
      "name": "busybox1"
  },
  "image":{
      "image": "busybox"
  },
  "command": [
      "echo"
  ],
  "args": [
      "Container Fundamentals"
  ],
  "log_path":"busybox.1.log",
  "linux": {
  }
}
```

The **run** command takes as arguments the two config files, and it returns the container ID upon its start. The **ps -a** command displays containers in all states, the container ID, pod ID, image, age, and the container name.

```
student@ubuntu:~$ sudo crictl run container-config.json pod-config.json

c78627d2b77b639c4b57391c3a4b9ec3de45479098409b35776893d5016c58fa
```

```
student@ubuntu:~$ sudo crictl ps -a

CONTAINER           IMAGE               CREATED             STATE
NAME                ATTEMPT             POD ID
c78627d2b77b6       busybox             19 seconds ago      Exited
busybox1            0                   3a5c27fcdabb2
```

The **logs** command will show the output generated by the echo command.

```
student@ubuntu:~$ sudo crictl logs c78

Container Fundamentals
```

We can again remove the container and the pod to reclaim resources. An easier method to remove containers and pods is to remove the pod directly, with the **-f** option to **force** the removal of active pods. At the end we will list resources for validation:

```
student@ubuntu:~$ sudo crictl ps -a

CONTAINER           IMAGE               CREATED             STATE
NAME                ATTEMPT             POD ID
c78627d2b77b6       busybox             4 minutes ago       Exited
busybox1            0                   3a5c27fcdabb2


student@ubuntu:~$ sudo crictl pods

POD ID              CREATED             STATE               NAME
NAMESPACE           ATTEMPT             RUNTIME
3a5c27fcdabb2       4 minutes ago       Ready               busybox1-sandbox
default             1                   (default)


student@ubuntu:~$ sudo crictl rmp -f 3a5

Stopped sandbox 3a5
Removed sandbox 3a5
```

```
student@ubuntu:~$ sudo crictl pods

POD ID              CREATED             STATE               NAME
NAMESPACE           ATTEMPT             RUNTIME


student@ubuntu:~$ sudo crictl ps -a

CONTAINER           IMAGE               CREATED             STATE
NAME                ATTEMPT             POD ID
```

The container can also be configured with a command and no arguments, or a command followed by a more complex set of arguments. The following example with the **top** command and no args will just keep the container in a running state, and the **while-loop** example will run an infinite loop that displays some text, the container hostname and a timestamp every 10 seconds, and will also keep the container in a running state:

```
student@ubuntu:~$ vim container-config.json

{
...
  "command": [
      "top"
  ],
...
}


student@ubuntu:~$ vim container-config.json

{
...
  "command": [
      "sh"
  ],
  "args": [
      "-c", "while [ 1 ]; do echo Container Fundamentals $(hostname) $(date);
sleep 10; done"
  ],
...
}
```

Lab 7.1. Building Docker Images

Build an Image from a running Container

Let's run a container first, and apply some changes by saving data on its filesystem. After verifying the changes, the output of the date command, we can then create a new image out of the modified container:

```
student@ubuntu:~$ docker container run -ti --name myalpine alpine sh

Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine
a0d0a0d46f8b: Pull complete
Digest:
sha256:e1c082e3d3c45cccac829840a25941e679c25d438cc8412c2fa221cf1a824e6a
Status: Downloaded newer image for alpine:latest
/ # ls
bin    etc    lib    mnt    proc   run    srv    tmp    var
dev    home   media  opt    root   sbin   sys    usr
/ # date > /data
/ # ls
bin    dev    home   media  opt    root   sbin   sys    usr
data   etc    lib    mnt    proc   run    srv    tmp    var
/ # cat /data
Thu Aug  5 21:08:59 UTC 2021
/ # <Ctrl p + Ctrl q>


student@ubuntu:~$ docker container ls

CONTAINER ID   IMAGE    COMMAND   CREATED         STATUS       PORTS
NAMES
854500197d3a   alpine   "sh"      24 minutes ago Up 24 minutes
myalpine
```

```
...
```

```
student@ubuntu:~$ docker container diff myalpine

A /data
C /root
A /root/.ash_history
```

```
student@ubuntu:~$ docker container commit myalpine lfstudent/alpine:training

sha256:d59d3be4db94be933305cc8fc5d4a66fbc3ee33e7488262e2238281660d76701
```

```
student@ubuntu:~$ docker image ls

REPOSITORY          TAG         IMAGE ID       CREATED          SIZE
lfstudent/alpine    training    d59d3be4db94   43 seconds ago   5.6MB
alpine              latest      14119a10abf4   13 days ago      5.6MB
...
```

For verification, we may now create a container out of the new image, and verify the existence of the data produced earlier by the date command:

```
student@ubuntu:~$ docker container run -ti lfstudent/alpine:training

/ # cat /data
Thu Aug  5 21:08:59 UTC 2021
/ # <Ctrl p + Ctrl q>
```

```
student@ubuntu:~$ docker container ls

CONTAINER ID   IMAGE                         COMMAND                  CREATED
STATUS              PORTS                                  NAMES
f618cd9660b2   lfstudent/alpine:training   "sh"                     About a
minute ago   Up About a minute
crazy_jepsen
854500197d3a   alpine                        "sh"                     40 minutes
ago       Up 40 minutes
myalpine
```

Export a Container filesystem to a tar archive

The filesystem of a running container can be exported as a tar archive. Subsequently, a new image can be created from the tar archive.

```
student@ubuntu:~$ docker container ls

CONTAINER ID   IMAGE                          COMMAND                  CREATED
STATUS              PORTS                                      NAMES
f618cd9660b2   lfstudent/alpine:training   "sh"                     About a
minute ago   Up About a minute
crazy_jepsen
...


student@ubuntu:~$ docker container export f618cd9660b2 > lfstudent_alpine.tar


student@ubuntu:~$ ls lfstudent_alpine.tar

lfstudent_alpine.tar
```

Import filesystem from a tar archive into an Image

From an existing tar file, which archives a container filesystem, we can create a new image:

```
student@ubuntu:~$ ls lfstudent_alpine.tar

lfstudent_alpine.tar


student@ubuntu:~$ docker image import lfstudent_alpine.tar
lfstudent/alpine:latest

sha256:05e432b37b51432c29cf31af5e17d4cdff38c096763de63ed01d8a93ca343177


student@ubuntu:~$ docker image ls

REPOSITORY          TAG         IMAGE ID       CREATED              SIZE
lfstudent/alpine    latest      05e432b37b51   About a minute ago   5.6MB
```

```
lfstudent/alpine    training    d59d3be4db94    13 minutes ago       5.6MB
alpine              latest      14119a10abf4    13 days ago          5.6MB
...
```

As a verification step we may run a container out of the new image, and verify the existence of the data produced earlier by the date command:

```
student@ubuntu:~$ docker container run -ti lfstudent/alpine:latest sh

/ # cat /data
Thu Aug  5 21:08:59 UTC 2021
/ # <Ctrl p + Ctrl q>


student@ubuntu:~$ docker container ls

CONTAINER ID   IMAGE                          COMMAND                  CREATED
STATUS          PORTS                                      NAMES
0dd89dde545d   lfstudent/alpine:latest        "sh"                     31 seconds
ago   Up 28 seconds
elated_spence
f618cd9660b2   lfstudent/alpine:training      "sh"                     14 minutes
ago   Up 14 minutes                                        crazy_jepsen
854500197d3a   alpine                         "sh"                     53 minutes
ago   Up 53 minutes                                        myalpine
...
```

Push an image to Docker Hub

Assuming the existence of a **lfstudent** account on Docker Hub, a user is required to login from the CLI before attempting to push a container image to the registry:

```
student@ubuntu:~$ docker login

Login with your Docker ID to push and pull images from Docker Hub. If you
don't
have a Docker ID, head over to https://hub.docker.com to create one.
Username: lfstudent
Password: ********
Login Succeeded
```

Once logged in to Docker Hub, a user may push an image into the registry to be shared by other users:

```
student@ubuntu:~$ docker image push lfstudent/alpine:training

The push refers to a repository [docker.io/lfstudent/alpine]
724d404d96ef: Pushed
60ab55d3379d: Mounted from library/alpine
training: digest:
sha256:fcc29a8a772bed232fc3026f9ea5df745e57ea4c99b2306f997036510d4be0f9 size:
735
```

Remove unused cached images from local repository

When removing dangling images, make sure to confirm it by typing **y** at the prompt:

```
student@ubuntu:~$ docker image prune

WARNING! This will remove all dangling images.
Are you sure you want to continue? [y/N]
y
Deleted Images:
deleted:
sha256:74688f28366cd966f203b54f175de233846e7b720eb2e0d09fae06a26a939779
Total reclaimed space: 0 B
```

Remove one or more cached images from the local repository

```
student@ubuntu:~$ docker image rm -f alpine:latest nginx:latest

Untagged: alpine:latest
Untagged:
alpine@sha256:dfbd4a3a8ebca874ebd2474f044a0b33600d4523d03b0df76e5c5986cb02d7e8
Untagged: nginx:latest
Untagged:
nginx@sha256:f2d384a6ca8ada733df555be3edc427f2e5f285ebf468aae940843de8cf74645
Deleted:
sha256:cc1b614067128cd2f5cdafb258b0a4dd25760f14562bcce516c13f760c3b79c4
...
```

Lab 7.2. Building a Docker Image with Dockerfile

Another method to create and share a Docker containerized application is through a Dockerfile. For a while, this method was Docker specific. However, other runtimes and image building tools have adopted this method to build and distribute images. With Dockerfile, instead of sharing the container image through a registry, we share a script with steps to create the image. The Dockerfile represents a reproducible method allowing the creation of identical images on any platform supporting Docker.

Let's examine a redacted example of Dockerfile of the latest (as of the time of this writing) nginx container image from Docker Hub:

```
...
#
FROM debian:buster-slim

LABEL maintainer="NGINX Docker Maintainers <docker-maint@nginx.com>"

ENV NGINX_VERSION   1.21.3
ENV NJS_VERSION     0.6.2
ENV PKG_RELEASE     1~buster

RUN set -x \
# create nginx user/group first, to be consistent throughout docker variants
    && addgroup --system --gid 101 nginx \
    && adduser --system --disabled-login --ingroup nginx --no-create-home
--home /nonexistent --gecos "nginx user" --shell /bin/false --uid 101 nginx \
    && apt-get update \
    && apt-get install --no-install-recommends --no-install-suggests -y gnupg1
...

# forward request and error logs to docker log collector
RUN ln -sf /dev/stdout /var/log/nginx/access.log \
    && ln -sf /dev/stderr /var/log/nginx/error.log
```

```
EXPOSE 80

STOPSIGNAL SIGQUIT

CMD ["nginx", "-g", "daemon off;"]
```

The file includes reserved keywords such as FROM, LABEL, ENV, RUN, EXPOSE, COPY, STOPSIGNAL, and CMD, that are instructions followed by sets of arguments. These instructions are read by the Docker daemon when the docker build command is issued from the Docker client CLI, to build the container images as specified by the Dockerfile. By default, the build process looks for a file called Dockerfile inside the context folder. We can also use custom configuration files as long as the build process refers to them via the **-f** option.

Let's attempt to create our own Dockerfile (that will include only the FROM and RUN instructions shown below) in a custom application subdirectory, which would be treated as the context of our build:

```
student@ubuntu:~$ mkdir myapp

student@ubuntu:~$ cd myapp/

student@ubuntu:~/myapp$ vim Dockerfile

FROM alpine
RUN date > data


student@ubuntu:~/myapp$ cat Dockerfile

FROM alpine
RUN date > data
```

Let's build the image, based on the Dockerfile we had just created:

```
student@ubuntu:~/myapp$ docker image build -t lfstudent/alpine:dockerfile .

Sending build context to Docker daemon  2.048kB
Step 1/2 : FROM alpine
 ---> 14119a10abf4
Step 2/2 : RUN date > data
 ---> Running in 7e181ab14a1f
Removing intermediate container 7e181ab14a1f
 ---> d90e2253123f
Successfully built d90e2253123f
Successfully tagged lfstudent/alpine:dockerfile
```

This build command instructs the docker daemon to use the current directory as context, and use the Dockerfile found at the top of the context, while tagging the image with the provided name and tag. The context is archived into a tarball and sent to the Docker daemon running on the Docker host (which could be local or remote). Our Dockerfile includes only two instructions (FROM and RUN), each corresponding to a step in the build process. Step 1 instructs the daemon to use the alpine container image as a base image, which in subsequent step(s) is customized to alter its behavior. Step 2 instructs the daemon to run a command that alters the base image filesystem. Step 2 runs the date command and saves some data onto the writable layer of the filesystem. At the end of the build process, once both steps 1 and 2 have completed, the newly built image is a modified alpine image, altered with some additional data.

```
student@ubuntu:~/myapp$ docker image ls

REPOSITORY         TAG           IMAGE ID          CREATED              SIZE
lfstudent/alpine   dockerfile    d90e2253123f      2 minutes ago        5.6MB
```

The same build could have been achieved by running the build command with a reference to the myapp directory which includes the Dockerfile:

```
student@ubuntu:~$ pwd

/home/student


student@ubuntu:~$ docker image build -t lfstudent/alpine:dockerfile myapp
```

Image caching and build times

By default, Docker caches prior build steps to achieve faster future builds from the same base image. Create a new directory **myapp2** and populate it with a Dockerfile of a simplified nginx image definition (presented below). Then let's output the time required by the initial build:

```
student@ubuntu:~/myapp$ cd


student@ubuntu:~$ mkdir myapp2


student@ubuntu:~$ cd myapp2
```

```
student@ubuntu:~/myapp2$ vim Dockerfile

#
# nginx simple Dockerfile
#

# Pull base image.
FROM ubuntu

# Install Nginx.
RUN \
  apt-get update && \
  apt-get install -y nginx && \
  rm -rf /var/lib/apt/lists/* && \
  echo "\ndaemon off;" >> /etc/nginx/nginx.conf && \
  chown -R www-data:www-data /var/lib/nginx

# Define mountable directories.
VOLUME ["/etc/nginx/sites-enabled", "/etc/nginx/certs", "/etc/nginx/conf.d",
"/var/log/nginx", "/var/www/html"]

# Define working directory.
WORKDIR /etc/nginx

# Define default command.
CMD ["nginx"]

# Expose ports.
EXPOSE 80



student@ubuntu:~/myapp2$ cat Dockerfile

#
# nginx simple Dockerfile
#

# Pull base image.
FROM ubuntu
...
# Define default command.
CMD ["nginx"]

# Expose ports.
EXPOSE 80
```

```
student@ubuntu:~/myapp2$ time docker image build -t lfstudent/nginx:dockerfile .

Sending build context to Docker daemon  11.26kB
Step 1/6 : FROM ubuntu
 ---> fb52e22af1b0
Step 2/6 : RUN   apt-get update &&   apt-get install -y nginx &&   rm -rf
/var/lib/apt/lists/* &&   echo "\ndaemon off;" >> /etc/nginx/nginx.conf &&
chown -R www-data:www-data /var/lib/nginx
 ---> Running in b789c4085250
Get:1 http://security.ubuntu.com/ubuntu focal-security InRelease [114 kB]
Get:2 http://archive.ubuntu.com/ubuntu focal InRelease [265 kB]
...
Setting up nginx (1.18.0-0ubuntu1.2) ...
Processing triggers for libc-bin (2.31-0ubuntu9.2) ...
Removing intermediate container b789c4085250
 ---> 0d24b6f17972
Step 3/6 : VOLUME ["/etc/nginx/sites-enabled", "/etc/nginx/certs",
"/etc/nginx/conf.d", "/var/log/nginx", "/var/www/html"]
 ---> Running in 51dcc8cf310a
Removing intermediate container 51dcc8cf310a
 ---> 61d8e2779000
Step 4/6 : WORKDIR /etc/nginx
 ---> Running in 5dc989e85f61
Removing intermediate container 5dc989e85f61
 ---> 7241451ace93
Step 5/6 : CMD ["nginx"]
 ---> Running in dca1651aa4e3
Removing intermediate container dca1651aa4e3
 ---> 6ee189f83cf2
Step 6/6 : EXPOSE 80
 ---> Running in 76c396a1d9dd
Removing intermediate container 76c396a1d9dd
 ---> 7f065a5bc6b4
Successfully built 7f065a5bc6b4
Successfully tagged lfstudent/nginx:dockerfile

real  1m40.051s
user  0m0.055s
sys   0m0.050s
```

While the initial build took about 1 minute and 40 seconds, let's attempt another build from the same Dockerfile:

```
student@ubuntu:~/myapp2$ time docker image build -t lfstudent/nginx:cached .

Sending build context to Docker daemon  11.26kB
Step 1/6 : FROM ubuntu
 ---> fb52e22af1b0
Step 2/6 : RUN   apt-get update &&   apt-get install -y nginx &&   rm -rf
/var/lib/apt/lists/* &&   echo "\ndaemon off;" >> /etc/nginx/nginx.conf &&
chown -R www-data:www-data /var/lib/nginx
 ---> Using cache
 ---> 0d24b6f17972
Step 3/6 : VOLUME ["/etc/nginx/sites-enabled", "/etc/nginx/certs",
"/etc/nginx/conf.d", "/var/log/nginx", "/var/www/html"]
 ---> Using cache
 ---> 61d8e2779000
Step 4/6 : WORKDIR /etc/nginx
 ---> Using cache
 ---> 7241451ace93
Step 5/6 : CMD ["nginx"]
 ---> Using cache
 ---> 6ee189f83cf2
Step 6/6 : EXPOSE 80
 ---> Using cache
 ---> 7f065a5bc6b4
Successfully built 7f065a5bc6b4
Successfully tagged lfstudent/nginx:cached

real  0m0.958s
user  0m0.027s
sys   0m0.045s
```

The second build took a little less than 1 second, because most build steps were referenced from the cache, with the exception of step 1. Let's modify the Dockerfile, and attempt another build. Edit the Dockerfile and EXPOSE port number 443 as well:

```
student@ubuntu:~/myapp2$ vim Dockerfile

#
# nginx simple Dockerfile
#

# Pull base image.
FROM ubuntu

# Install Nginx.
...
```

```
# Define default command.
CMD ["nginx"]

# Expose ports.
EXPOSE 80
EXPOSE 443
```

```
student@ubuntu:~/myapp2$ time docker image build -t lfstudent/nginx:expose .

Sending build context to Docker daemon  11.26kB
Step 1/7 : FROM ubuntu
 ---> fb52e22af1b0
Step 2/7 : RUN   apt-get update &&   apt-get install -y nginx &&   rm -rf
/var/lib/apt/lists/* &&   echo "\ndaemon off;" >> /etc/nginx/nginx.conf &&
chown -R www-data:www-data /var/lib/nginx
 ---> Using cache
 ---> 0d24b6f17972
Step 3/7 : VOLUME ["/etc/nginx/sites-enabled", "/etc/nginx/certs",
"/etc/nginx/conf.d", "/var/log/nginx", "/var/www/html"]
 ---> Using cache
 ---> 61d8e2779000
Step 4/7 : WORKDIR /etc/nginx
 ---> Using cache
 ---> 7241451ace93
Step 5/7 : CMD ["nginx"]
 ---> Using cache
 ---> 6ee189f83cf2
Step 6/7 : EXPOSE 80
 ---> Using cache
 ---> 7f065a5bc6b4
Step 7/7 : EXPOSE 443
 ---> Running in 71b03f9f6790
Removing intermediate container 71b03f9f6790
 ---> 4370a99e55a8
Successfully built 4370a99e55a8
Successfully tagged lfstudent/nginx:expose

real  0m1.583s
user  0m0.033s
sys   0m0.038s
```

This build took 1.5 seconds, about half a second longer than the previous build which used the cache for every single build step. The latest build took longer because build step 7, representing the EXPOSE 443 instruction, had to be executed instead of using the cache as seen on steps 2 through 6.

Let's rerun the last build, instructing the build to avoid the cache:

```
student@ubuntu:~/myapp2$ time docker image build --no-cache -t
lfstudent/nginx:expose .

Sending build context to Docker daemon  11.26kB
Step 1/7 : FROM ubuntu
 ---> fb52e22af1b0
Step 2/7 : RUN   apt-get update &&   apt-get install -y nginx &&   rm -rf
/var/lib/apt/lists/* &&   echo "\ndaemon off;" >> /etc/nginx/nginx.conf &&
chown -R www-data:www-data /var/lib/nginx
 ---> Running in 9e5a51a8ecbf
Get:1 http://security.ubuntu.com/ubuntu focal-security InRelease [114 kB]
Get:2 http://archive.ubuntu.com/ubuntu focal InRelease [265 kB]
...
Setting up nginx (1.18.0-0ubuntu1.2) ...
Processing triggers for libc-bin (2.31-0ubuntu9.2) ...
Removing intermediate container 9e5a51a8ecbf
 ---> dc28f8699e70
Step 3/7 : VOLUME ["/etc/nginx/sites-enabled", "/etc/nginx/certs",
"/etc/nginx/conf.d", "/var/log/nginx", "/var/www/html"]
 ---> Running in 03ca37b5db7c
Removing intermediate container 03ca37b5db7c
 ---> 9a15cc4a5123
Step 4/7 : WORKDIR /etc/nginx
 ---> Running in 0446e24b645d
Removing intermediate container 0446e24b645d
 ---> edd451d5e8ca
Step 5/7 : CMD ["nginx"]
 ---> Running in be6cc2a1062b
Removing intermediate container be6cc2a1062b
 ---> 77d6a7267059
Step 6/7 : EXPOSE 80
 ---> Running in a267bd06497f
Removing intermediate container a267bd06497f
 ---> 711bfb1f5cfb
Step 7/7 : EXPOSE 443
 ---> Running in e3bfd231c96f
Removing intermediate container e3bfd231c96f
 ---> 6a232798505b
Successfully built 6a232798505b
Successfully tagged lfstudent/nginx:expose

real  1m30.681s
user  0m0.055s
sys   0m0.056s
```

This final build took 1 minute and 30 seconds, which is very close to the first timed build at 1 minute 40 seconds. From the output we see that each build step was executed and not referenced from the cache.

Listing the images we've previously built, we may notice something strange. The last image built is shown with its full repository name and tag (ID 6a232798505b), while the previous build (ID4370a99e55a8) no longer shows the repository and tag. With both builds using the same image name, only the latest image kept the desired name, while the previous image was stripped of its repository name and tag.

```
student@ubuntu:~/myapp2$ docker image ls

REPOSITORY          TAG             IMAGE ID       CREATED          SIZE
lfstudent/nginx     expose          6a232798505b   2 minutes ago    132MB
<none>              <none>          4370a99e55a8   5 minutes ago    132MB
lfstudent/nginx     cached          7f065a5bc6b4   10 minutes ago   132MB
lfstudent/nginx     dockerfile      7f065a5bc6b4   10 minutes ago   132MB
...
```

Such images with <none> as repository name and tag, are known as dangling images. We can filter through the image list to display only such images, and ultimately remove them from the local repository:

```
student@ubuntu:~/myapp2$ docker image ls --quiet -filter=dangling=true

4370a99e55a8


student@ubuntu:~/myapp2$ docker image ls --quiet --filter=dangling=true |
xargs --no-run-if-empty sudo docker image rm

Deleted:
sha256:4370a99e55a8b7e3094b35d8ee67af3e01f5fecbcf232b5135c51865b4cfbcd6


student@ubuntu:~/myapp2$ docker image ls

REPOSITORY          TAG             IMAGE ID       CREATED          SIZE
lfstudent/nginx     expose          6a232798505b   5 minutes ago    132MB
lfstudent/nginx     cached          7f065a5bc6b4   13 minutes ago   132MB
lfstudent/nginx     dockerfile      7f065a5bc6b4   13 minutes ago   132MB
...
```

Lab 7.3. Building Images with Podman

Build an Image from a running Container

Podman has significantly matured as a containerization tool, and it received several improvements aimed at container image builds earlier found in Buildah. Let's run a container first, and apply some changes by saving data on its filesystem. After verifying the changes, the output of the date command, we can then create a new image out of the modified container:

```
student@ubuntu:~$ podman container run -ti --name myalpine alpine sh

/ # ls
bin    etc    lib    mnt    proc   run    srv    tmp    var
dev    home   media  opt    root   sbin   sys    usr
/ # date > /data
/ # ls
bin    dev    home   media  opt    root   sbin   sys    usr
data   etc    lib    mnt    proc   run    srv    tmp    var
/ # cat /data
Thu Aug  5 23:11:34 UTC 2021
/ # <Ctrl p + Ctrl q>


student@ubuntu:~$ podman container ls

CONTAINER ID  IMAGE                                COMMAND     CREATED
STATUS                  PORTS       NAMES
f56ca963d39c  docker.io/library/alpine:latest  sh          About a minute ago
Up About a minute ago               myalpine
...
```

```
student@ubuntu:~$ podman container commit myalpine lfstudent/alpine:training

Getting image source signatures
Copying blob bc276c40b172 skipped: already exists
Copying blob ae54d4edfbaa done
Copying config 7ad7352d97 done
Writing manifest to image destination
Storing signatures
7ad7352d974956039113e61b000e4e4b706c0fd989f99b7beb2d1871e89da45f


student@ubuntu:~$ podman images

REPOSITORY                   TAG         IMAGE ID      CREATED         SIZE
localhost/lfstudent/alpine   training    7ad7352d9749  11 seconds ago  5.87 MB
docker.io/library/alpine     latest      021b3423115f  4 weeks ago     5.87 MB
...
```

For verification, we may now create a container out of the new image, and verify the existence of the data produced earlier by the date command:

```
student@ubuntu:~$ podman container run -ti lfstudent/alpine:training

/ # cat /data
Thu Aug  5 23:11:34 UTC 2021
/ # <Ctrl p + Ctrl q>


student@ubuntu:~$ podman container ls

CONTAINER ID  IMAGE                                 COMMAND     CREATED
STATUS              PORTS       NAMES
f56ca963d39c  docker.io/library/alpine:latest       sh          10 minutes ago
Up 10 minutes ago               myalpine
73c7e4701189  localhost/lfstudent/alpine:training   sh          18 seconds ago
Up 18 seconds ago               serene_meninsky
...
```

Export a Container filesystem to a tar archive

The filesystem of a running container can be exported as a tar archive. Subsequently, a new image can be created from the tar archive.

```
student@ubuntu:~$ podman container export 73c7e4701189 > lfstudent_alpine.tar
```

```
student@ubuntu:~$ ls lfstudent_alpine.tar

lfstudent_alpine.tar
```

Import filesystem from a tar archive into an Image

From an existing tar file, which archives a container filesystem, we can create a new image:

```
student@ubuntu:~$ podman image import lfstudent_alpine.tar
lfstudent/alpine:latest

Getting image source signatures
Copying blob 5da2882d6bd7 done
Copying config 2b59e80ad9 done
Writing manifest to image destination
Storing signatures
sha256:2b59e80ad93b9c4590e37298481450be53db8a87994afeb852bb47dd09956285
```

```
student@ubuntu:~$ podman images

REPOSITORY                  TAG         IMAGE ID      CREATED          SIZE
localhost/lfstudent/alpine  latest      2b59e80ad93b  3 minutes ago    5.87 MB
localhost/lfstudent/alpine  training    7ad7352d9749  39 minutes ago   5.87 MB
...
```

As a verification step we may run a container out of the new image, and verify the existence of the data produced earlier by the date command:

```
student@ubuntu:~$ podman container run -ti lfstudent/alpine:latest sh

/ # cat /data
Thu Aug  5 23:11:34 UTC 2021
/ # <Ctrl p + Ctrl q>
```

```
student@ubuntu:~$ podman container ls

CONTAINER ID  IMAGE                                  COMMAND     CREATED
STATUS                  PORTS       NAMES
f56ca963d39c  docker.io/library/alpine:latest      sh          54 minutes ago
Up 54 minutes ago                   myalpine
73c7e4701189  localhost/lfstudent/alpine:training  sh          44 minutes ago
Up 44 minutes ago                   serene_meninsky
4bf98b3a29a5  localhost/lfstudent/alpine:latest    sh          About a minute
ago  Up About a minute ago              eager_davinci
...
```

Lab 7.4. Build a Podman Image with Containerfile

Another method to create and share a containerized application is through a Containerfile or a Dockerfile. For a while, this method was Docker specific. However, other runtimes and image building tools, such as Podman and Buildah have adopted this method to build and distribute images. With a Containerfile, instead of sharing the container image through a registry, we share a script with steps to create the image. The Containerfile represents a reproducible method allowing the creation of identical images on any platform supporting Podman.

Let's attempt to create our own Containerfile (that will include only the FROM and RUN instructions shown below) in a custom application subdirectory, which would be treated as the context of our build:

```
student@ubuntu:~$ mkdir myapp

student@ubuntu:~$ cd myapp/

student@ubuntu:~/myapp$ vim Containerfile

FROM alpine
RUN date > data


student@ubuntu:~/myapp$ cat Containerfile

FROM alpine
RUN date > data
```

Let's build the image, based on the Containerfile we had just created:

```
student@ubuntu:~/myapp$ podman image build -t lfstudent/alpine:containerfile .

STEP 1: FROM alpine
```

```
STEP 2: RUN date > data
STEP 3: COMMIT lfstudent/alpine:containerfile
--> 063528b26a3
Successfully tagged localhost/lfstudent/alpine:containerfile
063528b26a37fb7882b18b37c9007b22889c0f507257a70ac5e3995912b01d36
```

This build command uses the current directory as context, and uses the Containerfile found at the top of the context, while tagging the image with the provided name and tag. Our Containerfile includes only two instructions (FROM and RUN), each corresponding to a step in the build process. Step 1 instructs the runtime to use the alpine container image as a base image, which in subsequent step(s) is customized to alter its behavior. Step 2 instructs the runtime to run a command that alters the base image filesystem. Step 2 runs the date command and saves some data onto the writable layer of the filesystem. At the end of the build process, once both steps 1 and 2 have completed, the newly built image is a modified alpine image, altered with some additional data.

```
student@ubuntu:~/myapp$ podman images

REPOSITORY                  TAG           IMAGE ID      CREATED
SIZE
localhost/lfstudent/alpine  containerfile  063528b26a37  18 minutes ago
5.87 MB
localhost/lfstudent/alpine  latest         2b59e80ad93b  50 minutes ago
5.87 MB
localhost/lfstudent/alpine  training       7ad7352d9749  About an hour ago
5.87 MB
...
```

The same build could have been achieved by running the build command with a reference to the myapp directory which includes the Containerfile:

```
student@ubuntu:~$ pwd

/home/student


student@ubuntu:~$ podman image build -t lfstudent/alpine:containerfile myapp
```

Image caching and build times

By default, Podman caches prior build steps to achieve faster future builds from the same base image. Create a new directory **myapp2** and populate it with a Containerfile of a simplified nginx image definition (presented below). Then let's output the time required by the initial build:

```
student@ubuntu:~/myapp$ cd


student@ubuntu:~$ mkdir myapp2


student@ubuntu:~$ cd myapp2


student@ubuntu:~/myapp2$ vim Containerfile

#
# nginx simple Containerfile
#

# Pull base image.
FROM ubuntu

# Install Nginx.
RUN \
  apt-get update && \
  apt-get install -y nginx && \
  rm -rf /var/lib/apt/lists/* && \
  echo "\ndaemon off;" >> /etc/nginx/nginx.conf && \
  chown -R www-data:www-data /var/lib/nginx

# Define mountable directories.
VOLUME ["/etc/nginx/sites-enabled", "/etc/nginx/certs", "/etc/nginx/conf.d",
"/var/log/nginx", "/var/www/html"]

# Define working directory.
WORKDIR /etc/nginx

# Define default command.
CMD ["nginx"]

# Expose ports.
EXPOSE 80



student@ubuntu:~/myapp2$ cat Containerfile
```

```
#
# nginx simple Containerfile
#

# Pull base image.
FROM ubuntu
...
# Define default command.
CMD ["nginx"]

# Expose ports.
EXPOSE 80
```

```
student@ubuntu:~/myapp2$ time podman image build -t lfstudent/nginx:containerfile .

STEP 1: FROM ubuntu
Resolved "ubuntu" as an alias
(/etc/containers/registries.conf.d/000-shortnames.conf)
Trying to pull docker.io/library/ubuntu:latest...
Getting image source signatures
Copying blob 35807b77a593 done
Copying config fb52e22af1 done
Writing manifest to image destination
Storing signatures
STEP 2: RUN   apt-get update &&   apt-get install -y nginx &&   rm -rf
/var/lib/apt/lists/* &&   echo "\ndaemon off;" >> /etc/nginx/nginx.conf &&
chown -R www-data:www-data /var/lib/nginx
Get:1 http://security.ubuntu.com/ubuntu focal-security InRelease [114 kB]
Get:2 http://archive.ubuntu.com/ubuntu focal InRelease [265 kB]
...
Setting up nginx (1.18.0-0ubuntu1.2) ...
Processing triggers for libc-bin (2.31-0ubuntu9.2) ...
--> acc5a459d29
STEP 3: VOLUME ["/etc/nginx/sites-enabled", "/etc/nginx/certs",
"/etc/nginx/conf.d", "/var/log/nginx", "/var/www/html"]
--> 48aff599098
STEP 4: WORKDIR /etc/nginx
--> 55f2b9aa344
STEP 5: CMD ["nginx"]
--> e089e4999de
STEP 6: EXPOSE 80
STEP 7: COMMIT lfstudent/nginx:containerfile
--> 16a1238f9b7
Successfully tagged localhost/lfstudent/nginx:containerfile
16a1238f9b7fdf9331710359122cdad9752a1504d8c123399a4916fbe58c3e8d
```

```
real  1m10.244s
user  0m15.575s
sys   0m7.042s
```

While the initial build took about 1 minute and 10 seconds, let's attempt another build from the same Containerfile:

```
student@ubuntu:~/myapp2$ time podman image build -t lfstudent/nginx:cached .

STEP 1: FROM ubuntu
STEP 2: RUN   apt-get update &&   apt-get install -y nginx &&   rm -rf
/var/lib/apt/lists/* &&   echo "\ndaemon off;" >> /etc/nginx/nginx.conf &&
chown -R www-data:www-data /var/lib/nginx
--> Using cache
acc5a459d29a63a124115582c557e358db97f4fd3326db2362f8966399e7e522
--> acc5a459d29
STEP 3: VOLUME ["/etc/nginx/sites-enabled", "/etc/nginx/certs",
"/etc/nginx/conf.d", "/var/log/nginx", "/var/www/html"]
--> Using cache
48aff59909897387e1ba0d10e8fa116517d23ea57676a1c9ea5a925f7221ae95
--> 48aff599098
STEP 4: WORKDIR /etc/nginx
--> Using cache
55f2b9aa344311d3d555f7c1538b7ae2e88ea32d87b3c8697ac248e0050287f9
--> 55f2b9aa344
STEP 5: CMD ["nginx"]
--> Using cache
e089e4999de1570586f73d73b2dd2c98e09db2f1efa73a04c94c43685078983e
--> e089e4999de
STEP 6: EXPOSE 80
--> Using cache
16a1238f9b7fdf9331710359122cdad9752a1504d8c123399a4916fbe58c3e8d
STEP 7: COMMIT lfstudent/nginx:cached
--> 16a1238f9b7
Successfully tagged localhost/lfstudent/nginx:cached
Successfully tagged localhost/lfstudent/nginx:containerfile
16a1238f9b7fdf9331710359122cdad9752a1504d8c123399a4916fbe58c3e8d

real  0m0.601s
user  0m0.112s
sys   0m0.145s
```

The second build took 0.6 seconds, because the build steps were referenced from the cache. Let's modify the Containerfile, and attempt another build. Edit the Containerfile and EXPOSE port number 443 as well:

```
student@ubuntu:~/myapp2$ vim Containerfile

#
# nginx simple Containerfile
#

# Pull base image.
FROM ubuntu

# Install Nginx.
...
# Define default command.
CMD ["nginx"]

# Expose ports.
EXPOSE 80
EXPOSE 443



student@ubuntu:~/myapp2$ time podman image build -t lfstudent/nginx:expose .

STEP 1: FROM ubuntu
STEP 2: RUN   apt-get update &&   apt-get install -y nginx &&   rm -rf
/var/lib/apt/lists/* &&   echo "\ndaemon off;" >> /etc/nginx/nginx.conf &&
chown -R www-data:www-data /var/lib/nginx
--> Using cache acc5a459d29a63a124115582c557e358db97f4fd3326db2362f8966399e7e522
--> acc5a459d29
STEP 3: VOLUME ["/etc/nginx/sites-enabled", "/etc/nginx/certs",
"/etc/nginx/conf.d", "/var/log/nginx", "/var/www/html"]
--> Using cache 48aff59909897387e1ba0d10e8fa116517d23ea57676a1c9ea5a925f7221ae95
--> 48aff599098
STEP 4: WORKDIR /etc/nginx
--> Using cache 55f2b9aa344311d3d555f7c1538b7ae2e88ea32d87b3c8697ac248e0050287f9
--> 55f2b9aa344
STEP 5: CMD ["nginx"]
--> Using cache e089e4999de1570586f73d73b2dd2c98e09db2f1efa73a04c94c43685078983e
--> e089e4999de
STEP 6: EXPOSE 80
--> Using cache 16a1238f9b7fdf9331710359122cdad9752a1504d8c123399a4916fbe58c3e8d
--> 16a1238f9b7
STEP 7: EXPOSE 443
STEP 8: COMMIT lfstudent/nginx:expose
```

```
--> 48d38f5c45b
Successfully tagged localhost/lfstudent/nginx:expose
48d38f5c45bc61b704672e7cfafb3c2a6a159f41e2c567015439965f4a8aa6cc

real  0m0.463s
user  0m0.128s
sys   0m0.147s
```

This build took nearly 0.5 seconds, comparable to the previous build which used the cache for all build steps. The latest build executed step 7, the EXPOSE 443 instruction, instead of using the cache as seen on steps 2 through 6.

Let's rerun the last build, instructing the build to avoid the cache:

```
student@ubuntu:~/myapp2$ time podman image build --no-cache -t
lfstudent/nginx:expose .

STEP 1: FROM ubuntu
STEP 2: RUN   apt-get update &&   apt-get install -y nginx &&   rm -rf
/var/lib/apt/lists/* &&   echo "\ndaemon off;" >> /etc/nginx/nginx.conf &&
chown -R www-data:www-data /var/lib/nginx
Get:1 http://security.ubuntu.com/ubuntu focal-security InRelease [114 kB]
Get:2 http://archive.ubuntu.com/ubuntu focal InRelease [265 kB]
...
Setting up nginx (1.18.0-0ubuntu1.2) ...
Processing triggers for libc-bin (2.31-0ubuntu9.2) ...
--> 3167aa41b25
STEP 3: VOLUME ["/etc/nginx/sites-enabled", "/etc/nginx/certs",
"/etc/nginx/conf.d", "/var/log/nginx", "/var/www/html"]
--> 26d44610fcc
STEP 4: WORKDIR /etc/nginx
--> 847fdfdaae9
STEP 5: CMD ["nginx"]
--> b5d5581ec70
STEP 6: EXPOSE 80
--> 3d659793895
STEP 7: EXPOSE 443
STEP 8: COMMIT lfstudent/nginx:expose
--> d110740f8e3
Successfully tagged localhost/lfstudent/nginx:expose
d110740f8e3e0c52f7f7b4b665981032c0309f7f98d9ddceb1f6b832c4090b29

real  0m49.716s
user  0m10.024s
sys   0m3.215s
```

This final build took close to 50 seconds, which is very close to the first timed build at 1 minute 10 seconds. From the output we see that each build step was executed and not referenced from the cache.

Lab 8.1. Docker Networking

Docker Networking

Working with Docker networks

1. List available networks

Before working with networks and container networking in Docker, let's quickly display the networks available in Docker running on our host system:

```
student@ubuntu:~$ docker network ls

NETWORK ID          NAME                DRIVER              SCOPE
0178f26dea86        bridge              bridge              local
812e1f9599d1        host                host                local
3cadc63d003f        none                null                local
```

Docker makes available three different networks for usage with containers: the default bridge network, and two additional networks used to start and connect a container directly to the host networking stack, or to start a container with no network devices.

2. Display network details

Displaying details of a network can be achieved by referencing the network by its network ID or by its name. Although using either the network ID `0178f26dea86` or the name `bridge` will produce the same output, we can take advantage of the autocomplete feature when typing out the command with the network name instead of the network ID. The output has been slightly edited for readability:

```
student@ubuntu:~$ docker network inspect 0178f26dea86

[
    {
        "Name": "bridge",
        "Id": "0178f26dea86320062538f6fb22857adde9c602ea4edf8750c00c84f1...",
        "Created": "2021-06-10T18:53:02.313768719Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.17.0.0/16",
                    "Gateway": "172.17.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {},
        "Options": {
            "com.docker.network.bridge.default_bridge": "true",
            "com.docker.network.bridge.enable_icc": "true",
            "com.docker.network.bridge.enable_ip_masquerade": "true",
            "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
            "com.docker.network.bridge.name": "docker0",
            "com.docker.network.driver.mtu": "1500"
        },
        "Labels": {}
    }
]
```

By default, the **bridge** network uses the **bridge** driver, and it creates the **172.17.0.0/16** subnet for the IP addresses to be assigned to containers running on this host when attached to the **bridge** network.

3. Create a user-defined network

Let's create a **bridge** network of our own, ensuring we can **attach** running containers to it as well as new containers. The new **mynet** network also uses the **bridge** driver, and it creates the

172.18.0.0/16 subnet for the IP addresses to be assigned to containers running on this host when attached to the **mynet** network:

```
student@ubuntu:~$ docker network create -d bridge --attachable mynet

bdee7801048d562fd3c4d3303ad2fca0f48ceae15d2fe2c96076219bf4a7cb8b


student@ubuntu:~$ docker network ls

NETWORK ID          NAME                DRIVER              SCOPE
0178f26dea86        bridge              bridge              local
812e1f9599d1        host                host                local
bdee7801048d        mynet               bridge              local
3cadc63d003f        none                null                local


student@ubuntu:~$ docker network inspect mynet

[
    {
        "Name": "mynet",
        "Id": "bdee7801048d562fd3c4d3303ad2fca0f48ceae15d2fe2c96076219bf4...",
        "Created": "2021-07-10T19:59:01.756312Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.18.0.0/16",
                    "Gateway": "172.18.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": true,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {},
        "Options": {},
        "Labels": {}
    }
]
```

By contrast, the **mynet** network is attachable while the default **bridge** network is not.

4. List all containers attached to a particular network

Although there is not one command to list all containers connected to a particular network, this is a method of listing those containers:

```
student@ubuntu:~$ docker network inspect <network-name>|grep Name|tail -n
+2|cut -d':' -f2|tr -d ' ,"'
```

5. Remove a particular network or all unused networks

Prior to removing a Docker network, you must ensure that all running containers have been disconnected from it. Let's create a **testnet** network first and then remove it, so that we save the **mynet** network for the following exercises.

```
student@ubuntu:~$ docker network create testnet

fb46a2007c7d7de6084e8ae608a48ad3a9819a4b750fdf9294f1f9452ae8ee9a


student@ubuntu:~$ docker network ls

NETWORK ID          NAME                DRIVER              SCOPE
0178f26dea86        bridge              bridge              local
812e1f9599d1        host                host                local
bdee7801048d        mynet               bridge              local
3cadc63d003f        none                null                local
fb46a2007c7d        testnet             bridge              local


student@ubuntu:~$ docker network rm testnet

testnet


student@ubuntu:~$ docker network ls

NETWORK ID          NAME                DRIVER              SCOPE
0178f26dea86        bridge              bridge              local
812e1f9599d1        host                host                local
bdee7801048d        mynet               bridge              local
3cadc63d003f        none                null                local
```

A network can be removed either by name or by network ID. Before removal we may verify that there are no containers attached to a particular network by displaying the network's details with **`docker network inspect <network-name>`** and observing the **`"Containers": {}`** field. The empty curly brackets confirm that no container is attached to this network. By listing the networks after the creation step and after the removal step we verify that the **`testnet`** network has been created and then removed successfully.

If we have many unused networks (and we confirmed that there are no containers attached to them), and we want to remove all of them at once, we may do so with the following command:

```
student@ubuntu:~$ docker network prune
```

Working with container networking

1. Display the IP address of a running container

Let's run a container with the **`nginx`** container image without specifying a network this time, and then display the container details to observe the network it is attached to by default, the IP address assigned to it, MAC address, etc. The output has been slightly edited for readability:

```
student@ubuntu:~$ docker container run -d --name web nginx

b0025c13ea35a1d0c2a21a1f9af397ab5e6ea7e8c7eca17501c8a316909123a9


student@ubuntu:~$ docker container ls

CONTAINER ID  IMAGE  COMMAND      CREATED         STATUS        PORTS   NAMES
b0025c13ea35  nginx  "/docker-e…" 13 seconds ago Up 8 seconds 80/tcp  web


student@ubuntu:~$ docker container inspect web | more

[
    {
        "Id": "b0025c13ea35a1d0c2a21a1f9af397ab5e6ea7e8c7eca17501c8a316...",
        ...
        "State": {
            "Status": "running",
            "Running": true,
            "Paused": false,
            "Restarting": false,
```

```
            "OOMKilled": false,
            "Dead": false,
            "Pid": 10003,
            "ExitCode": 0,
            "Error": "",
            ...
        },
...
        "NetworkSettings": {
            "Bridge": "",
            "SandboxID": "c7f17408ca95acf90ae334ced9b2cdfcf93dc3ce076c5...",
            "HairpinMode": false,
            "LinkLocalIPv6Address": "",
            "LinkLocalIPv6PrefixLen": 0,
            "Ports": {
                "80/tcp": null
            },
            "SandboxKey": "/var/run/docker/netns/c7f17408ca95",
            "SecondaryIPAddresses": null,
            "SecondaryIPv6Addresses": null,
            "EndpointID": "f8ad764b149489a08e6512ec2194ca95e75a3ff247a9...",
            "Gateway": "172.17.0.1",
            "GlobalIPv6Address": "",
            "GlobalIPv6PrefixLen": 0,
            "IPAddress": "172.17.0.2",
            "IPPrefixLen": 16,
            "IPv6Gateway": "",
            "MacAddress": "02:42:ac:11:00:02",
            "Networks": {
                "bridge": {
                    "IPAMConfig": null,
                    "Links": null,
                    "Aliases": null,
                    "NetworkID": "0178f26dea86320062538f6fb22857adde9c6...",
                    "EndpointID": "f8ad764b149489a08e6512ec2194ca95e75a...",
                    "Gateway": "172.17.0.1",
                    "IPAddress": "172.17.0.2",
                    "IPPrefixLen": 16,
                    "IPv6Gateway": "",
                    "GlobalIPv6Address": "",
                    "GlobalIPv6PrefixLen": 0,
                    "MacAddress": "02:42:ac:11:00:02",
                    "DriverOpts": null
                }
            }
        }
    }
```

```
]
```

To no surprise, the **web** container is running attached to the default **bridge** network (observe the **"NetworkID": "0178f26dea86…"** field) and it received the **172.17.0.2** IP address from the default **172.17.0.0/16** subnet.

When Docker daemon starts, it creates a **docker0** network bridge on the host system. By default, all the containers connect to the **docker0** network bridge. Docker creates a **veth** pair to attach a container to a bridge. One end of the **veth** pair is attached to the bridge while the other end to the container. The bridge side of the veth pair is **vethcbba2f9@if7** while the container end of the veth pair is **eth0** (not displayed here, but accessible from inside the running container):

```
student@ubuntu:~$ ip a

1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: ens4: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1460 qdisc fq_codel state UP
group default qlen 1000
    link/ether 42:01:0a:80:00:03 brd ff:ff:ff:ff:ff:ff
    inet 10.128.0.3/32 scope global dynamic ens4
       valid_lft 2693sec preferred_lft 2693sec
    inet6 fe80::4001:aff:fe80:3/64 scope link
       valid_lft forever preferred_lft forever
3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
group default
    link/ether 02:42:92:80:94:51 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:92ff:fe80:9451/64 scope link
       valid_lft forever preferred_lft forever
4: br-bdee7801048d: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue
state DOWN group default
    link/ether 02:42:e0:44:d1:5e brd ff:ff:ff:ff:ff:ff
    inet 172.18.0.1/16 brd 172.18.255.255 scope global br-bdee7801048d
       valid_lft forever preferred_lft forever
8: vethcbba2f9@if7: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
master docker0 state UP group default
    link/ether c2:c9:21:01:c3:09 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet6 fe80::c0c9:21ff:fe01:c309/64 scope link
       valid_lft forever preferred_lft forever
```

2. Expose a container port on a specific port of the host

When running with default options, a container is not directly accessible from outside the host because for security and privacy reasons it is isolated and shielded from the outside world. However, we can expose a container to the outside world, for when a client needs access to a particular front-end service or a portal service running inside a container. We may publish a container port, or map a port on the host to a container port with a similar mapping notation: **<hostPort>:<containerPort>**.

```
student@ubuntu:~$ docker container run -d --name web1 -p 80:80 nginx

a1a9916f9abba19deb21138c32e3d4c68f6f10e84168027f4acdc2392d07210f


student@ubuntu:~$ docker container ls

CONTAINER ID    IMAGE    ...    STATUS          PORTS                   NAMES
a1a9916f9abb    nginx    ...    Up 8 seconds    0.0.0.0:80->80/tcp      web1
b0025c13ea35    nginx    ...    Up 2 hours      80/tcp                  web
```

Once the **web1** container port **80** has been published, we may access the **nginx** service running inside the container directly over the host IP (regardless of whether private or public IP of you host system):

```
student@ubuntu:~$ curl 10.128.0.3

<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
</html>


student@ubuntu:~$ curl 35.239.38.42

<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
</html>
```

Unfortunately, we may only publish one port of a single container through the port 80 of the host. Even if we attempt to publish another container port through the host port 80, we will be unsuccessful:

```
student@ubuntu:~$ docker container run -d --name web2 -p 80:80 nginx

ab4e1b2d3234af2b0a26e042f2279e46c6f875eac6769e96a4dfb445368fdb31
docker: Error response from daemon: driver failed programming external
connectivity on endpoint web2
```

```
(1a078fbf0a0df30b7ee646c8758ab04aeae1274adf17e2b99808adb4b1f2e73b): Bind for
0.0.0.0:80 failed: port is already allocated.
```

However, we may publish container port 80 through another host port, such as host port 8080:

```
student@ubuntu:~$ docker container run -d --name web3 -p 8080:80 nginx

13a805bb0b2e7ea93fc7ba7662c5c13acbbd2d74f92289b714506c08c6d8b789


student@ubuntu:~$ docker container ls

CONTAINER ID     IMAGE    ...   STATUS           PORTS                    NAMES
13a805bb0b2e     nginx    ...   Up 6 seconds     0.0.0.0:8080->80/tcp     web3
a1a9916f9abb     nginx    ...   Up 21 minutes    0.0.0.0:80->80/tcp       web1
b0025c13ea35     nginx    ...   Up 2 hours       80/tcp                   web
```

Because **curl** targets port **80** by default, we now have to specify the host port **8080** with the **curl** command in order to access the **nginx** service running inside the **web3** container:

```
student@ubuntu:~$ curl 10.128.0.3:8080

<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
</html>


student@ubuntu:~$ curl 35.239.38.42:8080

<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
</html>
```

3. Expose a container port on a random port of the host

In the previous exercise we manually mapped container ports to host ports. When we publish a couple of container ports on the host it may not be such a challenging task. However, when running hundreds, or possibly more containers on one host, keeping track of mapped port numbers could become quite challenging, if not impossible. For such challenging cases, Docker comes to the rescue and offers a very

simple solution - it maps all the ports of a container to random host ports picked from a pool of available host ports. Below, the port **80** of **web4** container is mapped randomly to host port **32768**.

```
student@ubuntu:~$ docker container run -d --name web4 -P nginx

d33bde5197c4a2a6d1c10752151c1a21a53ed234d2811cf7fd9a91cf5373410f


student@ubuntu:~$ docker container ls

CONTAINER ID    IMAGE    ...   STATUS          PORTS                    NAMES
d33bde5197c4    nginx    ...   Up 8 seconds    0.0.0.0:32768->80/tcp    web4
13a805bb0b2e    nginx    ...   Up 16 minutes   0.0.0.0:8080->80/tcp     web3
a1a9916f9abb    nginx    ...   Up 37 minutes   0.0.0.0:80->80/tcp       web1
b0025c13ea35    nginx    ...   Up 2 hours      80/tcp                   web
```

Now we need to specify port **32768** with the **curl** command in order to access the **nginx** service running inside the **web4** container:

```
student@ubuntu:~$ curl 10.128.0.3:32768

<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
</html>


student@ubuntu:~$ curl 35.239.38.42:32768

<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
</html>
```

4. Run a container without a network interface

We may run a container without attaching it to a network, perhaps to attach it later to a particular network or not attach it at all. In the interactive terminal of the container running the **alpine** image we are able to confirm there are no networks the container is attached to:

```
student@ubuntu:~$ docker container run -it --network=none alpine sh
```

```
Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine
e6b0cf9c0882: Pull complete
Digest:
sha256:21716586201556792 40babee0a7714f6509fae66898db422ad803b951257db78
Status: Downloaded newer image for alpine:latest
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
/ # exit
```

5. Share the host network namespace with a container

Similarly we may run a container that shares the host network namespace. In the interactive terminal of the container running the **alpine** image we are able to confirm that it shares the host network namespace by observing in the output the entire network configuration of the host system:

```
student@ubuntu:~$ docker container run -it --network=host alpine sh

/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: ens4: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1460 qdisc fq_codel state UP
qlen 1000
    link/ether 42:01:0a:80:00:03 brd ff:ff:ff:ff:ff:ff
    inet 10.128.0.3/32 scope global dynamic ens4
       valid_lft 1858sec preferred_lft 1858sec
    inet6 fe80::4001:aff:fe80:3/64 scope link
       valid_lft forever preferred_lft forever
3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:92:80:94:51 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:92ff:fe80:9451/64 scope link
       valid_lft forever preferred_lft forever
4: br-bdee7801048d: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue
state DOWN
    link/ether 02:42:e0:44:d1:5e brd ff:ff:ff:ff:ff:ff
    inet 172.18.0.1/16 brd 172.18.255.255 scope global br-bdee7801048d
       valid_lft forever preferred_lft forever
```

```
8: vethcbba2f9@if7: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc
noqueue master docker0 state UP
    link/ether c2:c9:21:01:c3:09 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::c0c9:21ff:fe01:c309/64 scope link
       valid_lft forever preferred_lft forever
10: veth2df7e3b@if9: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc
noqueue master docker0 state UP
    link/ether 06:5c:3b:41:c5:a8 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::45c:3bff:fe41:c5a8/64 scope link
       valid_lft forever preferred_lft forever
14: veth2c2a394@if13: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc
noqueue master docker0 state UP
    link/ether e2:f6:74:83:b7:18 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::e0f6:74ff:fe83:b718/64 scope link
       valid_lft forever preferred_lft forever
16: veth8008340@if15: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc
noqueue master docker0 state UP
    link/ether fe:cc:e6:87:66:8b brd ff:ff:ff:ff:ff:ff
    inet6 fe80::fccc:e6ff:fe87:668b/64 scope link
       valid_lft forever preferred_lft forever
/ # exit
```

In an attempt to verify this on the host system we may run the following command on the host system and compare the outputs. Surprisingly, or not, they are identical:

```
student@ubuntu:~$ ip a

1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: ens4: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1460 qdisc fq_codel state UP
qlen 1000
    link/ether 42:01:0a:80:00:03 brd ff:ff:ff:ff:ff:ff
    inet 10.128.0.3/32 scope global dynamic ens4
       valid_lft 1858sec preferred_lft 1858sec
    inet6 fe80::4001:aff:fe80:3/64 scope link
       valid_lft forever preferred_lft forever
3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:92:80:94:51 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:92ff:fe80:9451/64 scope link
       valid_lft forever preferred_lft forever
4: br-bdee7801048d: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue
state DOWN
    link/ether 02:42:e0:44:d1:5e brd ff:ff:ff:ff:ff:ff
    inet 172.18.0.1/16 brd 172.18.255.255 scope global br-bdee7801048d
       valid_lft forever preferred_lft forever
```

```
...
16: veth8008340@if15: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc
noqueue master docker0 state UP
...
```

6. Share a network namespace among containers

Let's revisit the **web** container, which is still running the **nginx** container image. Let's keep in mind that a container image is minimal in size, therefore it packs a minimal amount of features and libraries required by the container environment to run the image. Having said that, let's try to find the **ip** package in the running container by starting an interactive terminal into the **web** container:

```
student@ubuntu:~$ docker container exec -it web sh

# which ip
#
```

It was not found. However, we could really use the help of the **ip** package to display the IP address of the **web** container. So let's install it, display the container IP address, and then exit the container with the **Ctrl+p Ctrl+q** sequence:

```
# apt update
...
# apt install -y iproute2
...
# which ip
/sbin/ip
# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
7: eth0@if8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
group default
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
       valid_lft forever preferred_lft forever
# ^p^q
```

Let's run a new container and attempt to share the network namespace of the **web** container with this new container. Subsequently we are able to confirm that the new alpine container shares the web container network namespace together with its IP address **172.17.0.2**. This feature is commonly used by containerization tools such as Podman and rkt, and container orchestrators such as Kubernetes, in the implementation of the Pod resource.

```
student@ubuntu:~$ docker container run -it --network=container:web alpine sh

/# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
7: eth0@if8: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
state UP
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
       valid_lft forever preferred_lft forever
/ # ^p^q
```

7. Run a container and attach it to an existing network

In a previous exercise we created a **bridge** network, **mynet**, which was configured with the **172.18.0.0/16** subnet for container IP addresses. Let's run a container and attach it to the mynet network. Then let's display the container IP address:

```
student@ubuntu:~$ docker container run -it --network=mynet alpine sh

/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
23: eth0@if24: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
state UP
    link/ether 02:42:ac:12:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.18.0.2/16 brd 172.18.255.255 scope global eth0
       valid_lft forever preferred_lft forever
```

The new container was assigned the IP address **172.18.0.2** from the **mynet** network, confirming that this container was attached to the existing **mynet** network.

8. Attach a running container to an existing network

There may be times when an already running container needs to be attached to a network, such as when the network latency needs to be minimized between two communicating services, or to allow two services to talk to each other. Let's run a new container from the **alpine** container image. As expected, it will automatically connect to the default **bridge** network. Then let's manually attach the new **alpine** container to the existing **mynet** network and verify that its IP address was assigned out of the

172.18.0.0/16 subnet of the **mynet** network, while it also holds an IP from the **172.17.0.0/16** subnet of the default **bridge** network:

```
student@ubuntu:~$ docker container run -it --name alpine alpine sh

/ # ^p^q


student@ubuntu:~$ docker network connect mynet alpine


student@ubuntu:~$ docker container exec -it alpine sh

/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
31: eth0@if32: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
state UP
    link/ether 02:42:ac:11:00:06 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.6/16 brd 172.17.255.255 scope global eth0
       valid_lft forever preferred_lft forever
33: eth1@if34: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
state UP
    link/ether 02:42:ac:12:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.18.0.2/16 brd 172.18.255.255 scope global eth1
       valid_lft forever preferred_lft forever
/ #
```

9. Detach a running container from a network

Detaching a running container from a network is quite simple. Let's display the details of the **alpine** container, showing that it is connected to two networks: the default **bridge** network and the user-defined **mynet** network:

```
student@ubuntu:~$ docker container inspect alpine

...
       "NetworkSettings": {
            "Bridge": "",
            ...
            "Networks": {
                "bridge": {
                    "IPAMConfig": null,
                    "Links": null,
                    "Aliases": null,
                    "NetworkID": "0178f26dea86320062538f6fb22857...",
```

```
                    "EndpointID": "6020ed89d0846279f58f3b4b03098...",
                    "Gateway": "172.17.0.1",
                    "IPAddress": "172.17.0.6",
                    "IPPrefixLen": 16,
                    "IPv6Gateway": "",
                    "GlobalIPv6Address": "",
                    "GlobalIPv6PrefixLen": 0,
                    "MacAddress": "02:42:ac:11:00:06",
                    "DriverOpts": null
                },
                "mynet": {
                    "IPAMConfig": {},
                    "Links": null,
                    "Aliases": [
                        "fb81e99a7b7c"
                    ],
                    "NetworkID": "bdee7801048d562fd3c4d3303ad2fc...",
                    "EndpointID": "c9fcac5ee9584ade0b14df1f1f977...",
                    "Gateway": "172.18.0.1",
                    "IPAddress": "172.18.0.2",
                    "IPPrefixLen": 16,
                    "IPv6Gateway": "",
                    "GlobalIPv6Address": "",
                    "GlobalIPv6PrefixLen": 0,
                    "MacAddress": "02:42:ac:12:00:02",
                    "DriverOpts": null
                }
            }
        }
    }
]
```

Let's detach the **alpine** container from the **mynet** network, and confirm by displaying the container details again:

```
student@ubuntu:~$ docker network disconnect mynet alpine

student@ubuntu:~$ docker container inspect alpine

...
        "NetworkSettings": {
            "Bridge": "",
            ...
            "Networks": {
                "bridge": {
                    "IPAMConfig": null,
                    "Links": null,
                    "Aliases": null,
                    "NetworkID": "0178f26dea86320062538f6fb22857...",
                    "EndpointID": "6020ed89d0846279f58f3b4b03098...",
```

```
                    "Gateway": "172.17.0.1",
                    "IPAddress": "172.17.0.6",
                    "IPPrefixLen": 16,
                    "IPv6Gateway": "",
                    "GlobalIPv6Address": "",
                    "GlobalIPv6PrefixLen": 0,
                    "MacAddress": "02:42:ac:11:00:06",
                    "DriverOpts": null
                }
            }
        }
    }
]
```

Lab 8.2. Podman Networking

Working with Podman networks

1. Create and list a default network

In contrast with other containerization frameworks, Podman does not build a default set of networks for its containers. It becomes the users' responsibility to create desired networks for containers deployed with Podman. The easiest method to initialize a CNI compatible container network with Podman is to simply create a network, without any flags or options. The output provides the path to the configuration file of the newly created network:

```
student@ubuntu:~$ podman network create

/home/student/.config/cni/net.d/cni-podman0.conflist
```

Exploring the file reveals information about the network's configuration, such as the CNI version implemented, network type – **bridge**, its name – **cni-podman0**, an IP subnet range and its gateway.

```
student@ubuntu:~$ cat /home/student/.config/cni/net.d/cni-podman0.conflist

{
   "cniVersion": "0.4.0",
   "name": "cni-podman0",
   "plugins": [
      {
         "type": "bridge",
         "bridge": "cni-podman0",
         "isGateway": true,
         "ipMasq": true,
         "hairpinMode": true,
         "ipam": {
            "type": "host-local",
            "routes": [
               {
```

```
                "dst": "0.0.0.0/0"
             }
          ],
          "ranges": [
             [
                {
                   "subnet": "10.88.2.0/24",
                   "gateway": "10.88.2.1"
                }
             ]
          ]
       }
    },
    {
       "type": "portmap",
       "capabilities": {
          "portMappings": true
       }
    },
    {
       "type": "firewall",
       "backend": ""
    },
    {
       "type": "tuning"
    },
    {
       "type": "dnsname",
       "domainName": "dns.podman",
       "capabilities": {
          "aliases": true
       }
    }
 ]
}
```

Let's list networks display network details using Podman:

```
student@ubuntu:~$ podman network ls

NETWORK ID    NAME         VERSION     PLUGINS
39e9c7a64c68  cni-podman0  0.4.0       bridge,portmap,firewall,tuning,dnsname


student@ubuntu:~$ podman network inspect cni-podman0

[
    {
        "cniVersion": "0.4.0",
        "name": "cni-podman0",
        "plugins": [
```

```
            {
                "bridge": "cni-podman0",
                "hairpinMode": true,
                "ipMasq": true,
                "ipam": {
                    "ranges": [
                        [
                            {
                                "gateway": "10.88.2.1",
                                "subnet": "10.88.2.0/24"
                            }
                        ]
                    ],
                    "routes": [
                        {
                            "dst": "0.0.0.0/0"
                        }
                    ],
                    "type": "host-local"
                },
                "isGateway": true,
                "type": "bridge"
              ...
]
```

2. Create and list a custom network

A custom bridge CNI network can be created by specifying the desired subnet, a gateway, and even an IP range of the network's subnet:

```
student@ubuntu:~$ podman network create --subnet 10.99.3.0/24 --gateway
10.99.3.3 mynet

/home/student/.config/cni/net.d/mynet.conflistls
```

Let's list networks and inspect the newly created **mynet** network:

```
student@ubuntu:~$ podman network ls

NETWORK ID    NAME         VERSION     PLUGINS
39e9c7a64c68  cni-podman0  0.4.0       bridge,portmap,firewall,tuning,dnsname
11c844f95e28  mynet        0.4.0       bridge,portmap,firewall,tuning,dnsname


student@ubuntu:~$ podman network inspect mynet
```

```
[
    {
        "cniVersion": "0.4.0",
        "name": "mynet",
        "plugins": [
            {
                "bridge": "cni-podman1",
                "hairpinMode": true,
                "ipMasq": true,
                "ipam": {
                    "ranges": [
                        [
                            {
                                "gateway": "10.99.3.3",
                                "subnet": "10.99.3.0/24"
                            }
                        ]
                    ],
                    "routes": [
                        {
                            "dst": "0.0.0.0/0"
                        }
                    ],
                    "type": "host-local"
                },
                "isGateway": true,
                "type": "bridge"
            },
...
]
```

Working with container networking

1. Run a container attached to an existing network

Let's run a container with the **nginx** container image attached to the **mynet** network. The container will be assigned a random IP address from **mynet** network's subnet, revealed by the inspect command:

```
student@ubuntu:~$ podman container run -d --name mywebmynet --net mynet nginx

fe068734ed6a74d35761b6364527b5cee6038e846bd063b72c1f064aa1bb17e9
```

Let's inspect the container, and observe its IP address `10.99.3.1` from **mynet** network's subnet `10.99.3.0/24`:

```
student@ubuntu:~$ podman container inspect mywebmynet

[
    {
        "Id": "fe068734ed6a74d35761b6364527b5cee6038e846bd063b72c1f0...",
...
        "Mounts": [],
        "Dependencies": [],
        "NetworkSettings": {
            "EndpointID": "",
            "Gateway": "",
            "IPAddress": "",
            "IPPrefixLen": 0,
            "IPv6Gateway": "",
            "GlobalIPv6Address": "",
            "GlobalIPv6PrefixLen": 0,
            "MacAddress": "",
            "Bridge": "",
            "SandboxID": "",
            "HairpinMode": false,
            "LinkLocalIPv6Address": "",
            "LinkLocalIPv6PrefixLen": 0,
            "Ports": {},
            "SandboxKey": "/run/user/1000/netns/cni-2d290989-7419-2afa...",
            "Networks": {
                "mynet": {
                    "EndpointID": "",
                    "Gateway": "10.99.3.3",
                    "IPAddress": "10.99.3.1",
                    "IPPrefixLen": 24,
                    "IPv6Gateway": "",
                    "GlobalIPv6Address": "",
                    "GlobalIPv6PrefixLen": 0,
                    "MacAddress": "c6:36:8b:bc:a9:5c",
                    "NetworkID": "mynet",
                    "DriverOpts": null,
                    "IPAMConfig": null,
                    "Links": null
                }
            }
        },
...
            "NetworkMode": "bridge",
```

```
            "PortBindings": {},
            "RestartPolicy": {
                "Name": "",
                "MaximumRetryCount": 0
            },
...
    }
]
```

2. Run a container attached to another container's network namespace

Let's run a container with the **nginx** container image attached to the network namespace of the **mywebmynet** container. The newly created container will also use the same IP address that was initially assigned to the **mywebmynet** container from the **mynet** network's subnet, revealed by the inspect command:

```
student@ubuntu:~$ podman container run -d --name mywebnamespace --net
container:mywebmynet nginx

2fccb5a85278933f4deea910f6ff6262506986b5b7a6737f4f944b79f5fba596
```

Let's inspect the container, and observe its IP address **10.99.3.1** from **mynet** network's subnet **10.99.3.0/24**, that was originally assigned to the **mywebmynet** container and now shared by the two containers:

```
student@ubuntu:~$ podman container inspect mywebnamespace

[
    {
        "Id": "2fccb5a85278933f4deea910f6ff6262506986b5b7a6737f4f944b79...",
...
        "Name": "mywebnamespace",
        "RestartCount": 0,
        "Driver": "overlay",
        "MountLabel": "",
        "ProcessLabel": "",
...
        "NetworkSettings": {
...
            "Networks": {
```

```
                "mynet": {
                    "EndpointID": "",
                    "Gateway": "10.99.3.3",
                    "IPAddress": "10.99.3.1",
                    "IPPrefixLen": 24,
                    "IPv6Gateway": "",
                    "GlobalIPv6Address": "",
                    "GlobalIPv6PrefixLen": 0,
                    "MacAddress": "c6:36:8b:bc:a9:5c",
                    "NetworkID": "mynet",
                    "DriverOpts": null,
                    "IPAMConfig": null,
                    "Links": null
                }
            }
        },
...
            "NetworkMode": "container:fe068734ed6a74d35761b6364527b5cee...",
            "PortBindings": {},
            "RestartPolicy": {
                "Name": "",
                "MaximumRetryCount": 0
            },
...
]
```

3. Disconnect a running container from one network and connect it to another network

Let's disconnect the **mywebmynet** container from the **mynet** network, and attach it to the **cni-podman0** network, created earlier. The first set of container inspects reveal the **mynet** network and the IP from its subnet, while the second set of inspects will reveal the **cni-podman0** network attached to the container with a new IP address as well:

```
student@ubuntu:~$ podman container inspect mywebmynet | grep -i networkid
                    "NetworkID": "mynet",

student@ubuntu:~$ podman container inspect mywebmynet | grep -i ipaddress
            "IPAddress": "",
                    "IPAddress": "10.99.3.1",
```

```
student@ubuntu:~$ podman network disconnect mynet mywebmynet

student@ubuntu:~$ podman container inspect mywebmynet | grep -i networkid

student@ubuntu:~$ podman network connect cni-podman0 mywebmynet

student@ubuntu:~$ podman container inspect mywebmynet | grep -i networkid

                    "NetworkID": "cni-podman0",


student@ubuntu:~$ podman container inspect mywebmynet | grep -i ipaddress

            "IPAddress": "",
                    "IPAddress": "10.88.2.3",
```

In an earlier step we shared the **mywebmynet** container's network namespace with the **mywebnamespace** container, and inspecting the **mywebnamespace** container revealed an IP address from the **mynet** network subnet **10.99.3.0/24**. After we swapped networks on the **mywebmynet** container, we can validate that this process has automatically updated the IP address of the **mywebnamespace** container as well, because it is still sharing the **mywebmynet** container's namespace:

```
student@ubuntu:~$ podman container inspect mywebnamespace | grep -i ipaddress

            "IPAddress": "",
                    "IPAddress": "10.88.2.3",
```

Lab 9.1. Manage Storage with Docker

1. Run a container with a mounted volume

To manage volumes for containers, we may use the **--mount** or the **-v** options. Let's run the **alpine** container image in a **cvol** container while mounting a volume under the container's **/data** mount point. Passing the **-i** and **-t** options and the **sh** command allow us to interact with the container's environment from a terminal:

```
student@ubuntu:~$ docker container run -ti --mount target=/data --name cvol
alpine sh

/ # ls
bin    dev    home   media  opt    root   sbin   sys    usr
data   etc    lib    mnt    proc   run    srv    tmp    var
/ # cd /data/
/data # ls
/data # touch file1
/data # ls
file1
```

As a verification step, we created a **file1** file inside the **/data** directory of the **cvol** container.

Similarly, the volume could have been mounted inside the container with the **-v** option:

```
student@ubuntu:~$ docker container run -ti -v /data --name cvol alpine sh
```

2. Run a container with a mounted named volume (named volume created on the fly)

Let's run the same **alpine** container image in a **cmountvol** container while mounting a volume named **mountvol** under the container's **/data** mount point:

```
student@ubuntu:~$ docker container run -ti --mount
source=mountvol,target=/data --name cmountvol alpine sh

/ # cd /data/
/data # ls
/data # touch mount-file1
/data # ls
mount-file1
```

As a verification step, we created a **mount-file1** file inside the **/data** directory of the **cmountvol** container.

Similarly, the volume could have been mounted inside the container with the **-v** option:

```
student@ubuntu:~$ docker container run -ti -v mountvol:/data --name cmountvol
alpine sh
```

3. Display container details to view mounted volume properties

By displaying the container's details, inside the **Mounts** section we find detailed information about mounted volumes, such as source, destination, and access mode. The output below has been redacted for readability:

```
student@ubuntu:~$ docker container ls

CONTAINER ID   IMAGE    COMMAND CREATED         STATUS      PORTS   NAMES
f360a5ff317a   alpine   "sh"    2 minutes ago   Up 2 minutes        cmountvol
aead1c30cd3d   alpine   "sh"    4 minutes ago   Up 4 minutes cvol


student@ubuntu:~$ docker container inspect cvol | more

...
        "Mounts": [
            {
                "Type": "volume",
                "Name": "50875bf11582aaacb98e29063...",
                "Source": "/var/lib/docker/volumes/50875bf1.../_data",
```

```
                "Destination": "/data",
                "Driver": "local",
                "Mode": "z",
                "RW": true,
                "Propagation": ""
            }
        ],
...
```

The full volume name is **50875bf11582aaacb98e290630421ad9e71657b4957717f63848082e1cdc3624**, the full path to the source volume on the host system is **/var/lib/docker/volumes/50875bf11582aaacb98e290630421ad9e71657b4957717f63848082e1cdc3624/_data**, and the target mount point on the container is **/data**. This volume is mounted in Read-Write mode **("RW":true)**. By navigating to the source on the host system, we are able to find the verification file **file1** created inside the container's **/data** directory:

```
student@ubuntu:~$ sudo ls /var/lib/docker/volumes/50875bf11582aaac.../_data/

file1
```

By default, all the local volumes are saved under the **/var/lib/docker/volumes** directory of the host system.

4. List volumes

Let's list the volumes created and mounted so far:

```
student@ubuntu:~$ docker volume ls

DRIVER          VOLUME NAME
local           50875bf11582aaacb98e290630421ad9e71657b4957717f63...
local           mountvol
...
```

5. Create a named volume

Previously we allowed Docker to create volumes on our behalf at container runtime. However, Docker provides us with the capability to create our own volume, and then decide whether we want to mount it in a container.

Let's create a volume and then list the available volumes:

```
student@ubuntu:~$ docker volume create --name myvol
```

```
myvol
```

```
student@ubuntu:~$ docker volume ls

DRIVER          VOLUME NAME
local           50875bf11582aaacb98e290630421ad9e71657b4957717f63...
local           mountvol
local           myvol
...
```

6. Run a container with a mounted named volume (pre-created volume)

With the **myvol** volume available, we decide to mount it in a container and then create a file and add some content to it:

```
student@ubuntu:~$ docker container run -ti --mount source=myvol,target=/data
--name cmyvol alpine sh

/ # cd /data/
/data # echo "Docker volumes" > learning.txt
/data # ls
learning.txt
/data # cat learning.txt
Docker volumes
/data # Ctrl+p Ctrl+q
```

```
student@ubuntu:~$ cat /var/lib/docker/volumes/myvol/_data/learning.txt

Docker volumes
```

As a verification step, not only that we created a **learning.txt** file with some content on the mounted volume in the container, but also we verified the existence of the file together with its content from the host system by navigating to the source path of the volume **/var/lib/docker/volumes/myvol/_data/**.

7. Remove a volume

If the volume is created with the container, such as the case of the **cvol** container, we can remove the volume together with the container:

```
student@ubuntu:~$ docker container rm -f -v cvol

cvol
```

The `-v` option instructs Docker to remove the volume together with the container.

Volumes may be removed separately as well, provided they have been unmounted/released from the container that used them. The first attempt to remove the **`myvol`** volume fails, with the error message displaying the ID of the container still mounting the volume. We then have to stop and remove the container to unmount/release the volume we intend to remove:

```
student@ubuntu:~$ docker volume ls

DRIVER          VOLUME NAME
local           mountvol
local           myvol
...


student@ubuntu:~$ docker volume rm myvol

Error response from daemon: remove myvol: volume is in use -
[62870ebe162299620a92d7fd280d46f54f9337b420f8c072d8ce892f86ff2ff2]


student@ubuntu:~$ docker container ls

CONTAINER ID   IMAGE   COMMAND  CREATED          STATUS      PORTS     NAMES
62870ebe1622   alpine  "sh"     9 minutes ago    Up 8 minutes          cmyvol
f360a5ff317a   alpine  "sh"     23 minutes ago   Up 23 minutes         cmountvol


student@ubuntu:~$ docker container rm -f cmyvol

cmyvol


student@ubuntu:~$ docker volume rm myvol

myvol


student@ubuntu:~$ docker container ls

CONTAINER ID   IMAGE   COMMAND  CREATED          STATUS      PORTS     NAMES
f360a5ff317a   alpine  "sh"     24 minutes ago   Up 24 minutes         cmountvol


student@ubuntu:~$ docker volume ls

DRIVER          VOLUME NAME
local           mountvol
```

8. Mount a host directory inside a container in bind mode

We may use a host directory as a shared storage location between the host system and a container, by mounting a host directory inside the container. This can be achieved by specifying an additional parameter to declare it a bind type of mount. A bind type of mount, however, does not produce a new volume entry in the volumes list, and no source path in the default **/var/lib/docker/volumes/** directory either.

NOTE: After the **touch bind-file** command we want to detach from the container while also keeping it running so we can return to it. Use the **Ctrl+p Ctrl+q** keys combination to detach from a running container without terminating it at the same time (presented as **^p^q** below).

```
student@ubuntu:~$ sudo mkdir /mnt/shared


student@ubuntu:~$ docker container run -ti --mount
type=bind,source=/mnt/shared,target=/data --name csharedvol alpine sh

/ # cd /data/
/data # ls
/data # touch bind-file
/data # ^p^q


student@ubuntu:~$ cd /mnt/shared/


student@ubuntu:/mnt/shared$ ls

bind-file


student@ubuntu:/mnt/shared$ sudo bash -c 'echo "text from host" > bind-file'


student@ubuntu:/mnt/shared$ docker attach csharedvol

/data # ls
bind-file
/data # cat bind-file
text from host
```

After our verification step where we created the **bind-file** file from the container in the mounted directory, we are able to navigate to it and add content to the shared file from the host system, and finally return to the running container with the **attach** command (after ensuring we did not terminate the container by detaching from it earlier with **Ctrl+p Ctrl+q** keys combination) to read the shared content from inside the container.

9. Mount a Read-Only host directory inside a container in bind mode

In this scenario we provide an additional parameter for the Read-Only option to the bind mount type. During the verification steps, we are not allowed to create new content on the mounted volume from the container, although we attempt to create a new file and to add extra content to the existing file. In both cases we are reminded that the container has Read-Only access to the shared mounted volume, which allows the container to list files in the directory and read the contents of the file:

```
student@ubuntu:~$ mkdir /tmp/ro


student@ubuntu:~$ echo "host file" > /tmp/ro/host-ro-file


student@ubuntu:~$ docker container run -ti --mount
type=bind,source=/tmp/ro,target=/data,readonly --name crovol alpine sh

/ # cd /data/
/data # ls
host-ro-file
/data # touch container-file
touch: container-file: Read-only file system
/data # ls
host-ro-file
/data # cat host-ro-file
host file
/data # echo "hello from container" >> host-ro-file
sh: can't create host-ro-file: Read-only file system
/data # ls
host-ro-file
/data # cat host-ro-file
host file
```

10. Display host system disk usage by Docker

In order to store container image data, running container data, and data from volumes and other types of storage layer data, Docker makes use of the host system's storage. We may list the availability of free storage resources of the host system:

```
student@ubuntu:~$ docker system df

TYPE                TOTAL      ACTIVE      SIZE          RECLAIMABLE
Images              1          1           5.591MB       0B (0%)
Containers          4          4           73B           0B (0%)
Local Volumes       3          3           0B            0B
```

```
Build Cache          0          0            0B            0B
```

11. How to remove all unused volumes?

We may remove all the unused/unmounted/released volumes from docker, all with one command:

```
student@ubuntu:~$ docker volume prune

WARNING! This will remove all local volumes not used by at least one
container.
Are you sure you want to continue? [y/N] y
Deleted Volumes:
mountvol
20e639412902d69f40c5be3bc722b26ad3ebaaba3f35488a5799b024f930f22b
a0fab9c54ae01dc2d08c5ea0c2ba4de654fcb512c9526404c380cb2dca623b6a

Total reclaimed space: 0B


student@ubuntu:~$ docker volume ls

DRIVER              VOLUME NAME
```

Lab 9.2. Manage Storage with Podman

1. Run a container with a mounted volume

To manage volumes for containers, we may use the **--mount** or the **-v** options. Let's run the **alpine** container image in a **cvol** container while creating and mounting a volume with **-v** option under the container's **/data** mount point. Passing the **-i** and **-t** options and the **sh** command allow us to interact with the container's environment from a terminal:

```
student@ubuntu:~$ podman container run -i -t -v /data --name cvol alpine sh

/ # cd /data/
/data # ls
/data # touch file1
/data # ls
file1
```

As a verification step, we created a **file1** file inside the **/data** directory of the **cvol** container.

2. Run a container with a mounted named volume (named volume created on the fly)

Let's run the same **alpine** container image in a **cmountvol** container while creating and mounting a volume named **mountvol** under the container's **/data** mount point:

```
student@ubuntu:~$ podman container run -i -t -v mountvol:/data --name
cmountvol alpine sh

/ # cd /data/
/data # ls
/data # touch mount-file1
/data # ls
mount-file1
```

As a verification step, we created a **mount-file1** file inside the **/data** directory of the **cmountvol** container.

3. Display container details to view mounted volume properties or display volume details

By displaying the container's details, inside the **Mounts** section we find detailed information about mounted volumes, such as source, destination, and access mode. The output below has been redacted for readability:

```
student@ubuntu:~$ podman container ls

CONTAINER ID  IMAGE                                COMMAND     CREATED
STATUS                PORTS       NAMES
...
3056a4697493  docker.io/library/alpine:latest      sh          11 minutes ago
Up 11 minutes ago                 cvol
...


student@ubuntu:~$ podman container inspect cvol | more

...
        "Mounts": [
            {
                "Type": "volume",
                "Name": "617b6dea725aa7ad39e96357ba320963326253e6312136...",
                "Source": "/home/student/.local/share/containers/...",
                "Destination": "/data",
                "Driver": "local",
                "Mode": "",
                "Options": [
                    "nosuid",
                    "nodev",
                    "rbind"
                ],
                "RW": true,
                "Propagation": "rprivate"
            }
        ],
...
```

The full volume name is **617b6dea725aa7ad39e96357ba320963326253e6312136d8df74d9c9fdad8f00**, the full path to the source volume on the host system is **/home/student/.local/share/containers/storage/volumes/617b6dea725aa7ad39e96357ba320963326253e6312136d8df74d9c9fdad8f00/_data**, and the target mount point on the container is **/data**. This volume is mounted in Read-Write mode **("RW":true)**. By navigating to the source on the host system, we are able to find the verification file **file1** created inside the container's **/data** directory:

```
student@ubuntu:~$ ls
/home/student/.local/share/containers/storage/volumes/617b6dea725aa7ad39e96357
ba320963326253e6312136d8df74d9c9fdad8f00/_data/

file1
```

By default, all the local volumes are saved under **/home/student/.local/share/containers/storage/volumes** directory of the host system.

Separately, a volume's details can be displayed as well:

```
student@ubuntu:~$ podman volume inspect 617

[
    {
        "Name": "617b6dea725aa7ad39e96357ba320963326253e6312136d8df74d9...",
        "Driver": "local",
        "Mountpoint": "/home/student/.local/share/containers/storage/vo...",
        ...
    }
]
```

4. Create a named volume

Podman provides us with the capability to create our own volume, and then decide whether we want to mount it in a container.

Let's create a volume and then list the available volumes:

```
student@ubuntu:~$ podman volume create myvol

myvol
```

```
student@ubuntu:~$ podman volume ls

DRIVER      VOLUME NAME
local       617b6dea725aa7ad39e96357ba320963326253e6312136d8df74d9c9fdad8f00
local       mountvol
local       myvol
```

5. Run a container with a mounted named volume (pre-created volume)

With the **myvol** volume available, we decide to mount it in a container and then create a file and add some content to it:

```
student@ubuntu:~$ podman container run -i -t --mount
type=volume,src=myvol,target=/data --name cmyvol alpine sh

/ # cd /data/
/data # echo "Podman volumes" > learning.txt
/data # ls
learning.txt
/data # cat learning.txt
Podman volumes
/data # Ctrl+p Ctrl+q


student@ubuntu:~$ cat
/home/student/.local/share/containers/storage/volumes/myvol/_data/learning.txt

Podman volumes
```

As a verification step, not only that we created a **learning.txt** file with some content on the mounted volume in the container, but also we verified the existence of the file together with its content from the host system by navigating to the source path of the volume **/home/student/.local/share/containers/storage/volumes/myvol/_data/**.

6. Remove a volume

If the volume is created with the container, such as the case of the **cvol** container, we can remove the volume together with the container:

```
student@ubuntu:~$ podman container rm -f -v cvol
```

```
3056a469749320d83221868bebbcec99d1893763041c8dc14e32febaf864b9cc
```

The `-v` option instructs Podman to remove the volume together with the container.

Volumes may be removed separately as well, provided they have been unmounted/released from the container that used them. The first attempt to remove the **myvol** volume fails, with the error message displaying the ID of the container still mounting the volume. We then have to stop and remove the container to unmount/release the volume we intend to remove:

```
student@ubuntu:~$ podman volume ls

DRIVER          VOLUME NAME
local           mountvol
local           myvol


student@ubuntu:~$ podman volume rm myvol

Error: volume myvol is being used by the following container(s):
91731bb83bf030ddf07e5ab03ace4ae88419efc3967be0fdf68514740ffae00b: volume is
being used


student@ubuntu:~$ podman container ls

CONTAINER ID  IMAGE                                COMMAND     CREATED
STATUS                  PORTS       NAMES
...
91731bb83bf0  docker.io/library/alpine:latest      sh          About an hour
ago  Up About an hour ago              cmyvol
...


student@ubuntu:~$ podman container rm -f cmyvol

91731bb83bf030ddf07e5ab03ace4ae88419efc3967be0fdf68514740ffae00b


student@ubuntu:~$ podman volume rm myvol

myvol


student@ubuntu:~$ podman volume ls

DRIVER          VOLUME NAME
local           mountvol
```

7. Mount a host directory inside a container in bind mode (rootless and rooted)

We may use a host directory as a shared storage location between the host system and a container, by mounting a host directory inside the container. This can be achieved by specifying an additional parameter to declare it a bind type of mount, which does not produce a new volume entry in the volumes list.

NOTE: After the **touch bind-file** command we want to detach from the container while also keeping it running so we can return to it. Use the **Ctrl+p Ctrl+q** keys combination to detach from a running container without terminating it at the same time (presented as **^p^q** below).

In the following example the newly created **shared-bind** host directory is owned by **root**, therefore the attempt to create a **bind-file** in it from a rootless container will fail. For a successful **bind-file** creation we need to run a rooted container, in order to ensure the container has the needed permissions to write in a directory owned by the host **root**. However, a rootless host directory such as **/tmp/shared-bind**, can be mounted and accessed by a rootless container.

```
student@ubuntu:~$ sudo mkdir /mnt/shared-bind

student@ubuntu:~$ podman container run -i -t --mount
type=bind,src=/mnt/shared-bind,target=/data --name csharedvol alpine sh

/ # cd /data/
/data # ls
/data # touch bind-file
touch: bind-file: Permission denied
/data # ^p^q


student@ubuntu:~$ sudo podman container run -i -t --mount
type=bind,src=/mnt/shared-bind,target=/data --name csharedvolr alpine sh

/ # cd /data/
/data # ls
/data # touch bind-file
/data # ^p^q


student@ubuntu:~$ cd /mnt/shared-bind/

student@ubuntu:/mnt/shared-bind$ ls

bind-file
```

```
student@ubuntu:/mnt/shared-bind$ sudo bash -c 'echo "root text from host" >
bind-file'

student@ubuntu:/mnt/shared-bind$ sudo podman attach csharedvolr

/data # ls
bind-file
/data # cat bind-file
root text from host
```

After our verification step where we created the **bind-file** file from the rooted container in the mounted directory, we are able to navigate to it and add content to the shared file from the host system, and finally return to the running rooted container with the **attach** command (after ensuring we did not terminate the container by detaching from it earlier with **Ctrl+p Ctrl+q** keys combination) to read the shared content from inside the container.

8. Mount a Read-Only host directory inside a container in bind mode (rootless)

In this scenario we provide an additional parameter for the Read-Only option to the bind mount type. During the verification steps, we are not allowed to create new content on the mounted volume from the container, although we attempt to create a new file and to add extra content to the existing file. In both cases we are reminded that the container has Read-Only access to the shared mounted volume, which allows the container to list files in the directory and read the contents of the file:

```
student@ubuntu:~$ mkdir /tmp/ro

student@ubuntu:~$ echo "host file" > /tmp/ro/host-ro-file

student@ubuntu:~$ podman container run -i -t --mount
type=bind,src=/tmp/ro,target=/data,ro --name crovol alpine sh

/ # cd data/
/data # ls
host-ro-file
/data # touch container-file
touch: container-file: Read-only file system
/data # ls
host-ro-file
/data # cat host-ro-file
host file
/data # echo "hello from container" >> host-ro-file
sh: can't create host-ro-file: Read-only file system
/data # ls
host-ro-file
/data # cat host-ro-file
host file
```

9. Display host system disk usage by Podman

In order to store container image data, running container data, and data from volumes and other types of storage layer data, Podman makes use of the host system's storage. We may list the availability of free storage resources of the host system:

```
student@ubuntu:~$ podman system df

TYPE            TOTAL       ACTIVE      SIZE        RECLAIMABLE
Images          20          4           2.014GB     1.991GB (0%)
Containers      6           6           233B        0B (0%)
Local Volumes   0           0           0B          0B (0%)
```

10. How to remove all unused volumes?

We may remove all the unused/unmounted/released volumes, all with one command:

```
student@ubuntu:~$ podman volume prune

WARNING! This will remove all volumes not used by at least one container. The
following volumes will be removed:
No dangling volumes found
```