Red Hat
Training and
Certification

DO188 - Introduction to Containers with Podman

Travis Michette

Version 1.2

Table of Contents

Introduction

Repositories for this Course

Main Repository

The DO188 Demo repository contains the PDF of the instructor notes, and various pre-built demos for the chapters. Demos in this repository are meant to setup and configure the environment automatically to provide consistent demos from course-to-course. Each of the various chapter directories contains one or more playbooks for demonstration or for setting up and configuring the environment for running demonstrations from the various **Demo Repositories**,

- **DO188 Demo**: https://github.com/tmichett/DO467_Demo
- **DO188 (Private Notes)**: https://github.com/tmichett/DO467_Notes

The **DO188** Notes repository contains the Asciidoc code for the PDF as well as where the examples get developed. Both of these repositories are part of Jenkins workflows. The primary workflow task monitors the **NOTES** repository for changes. Upon changes, a new PDF is built. The PDF, along with the **Demos** directory are then promoted to the **DEMO** repository and published to the **main** branch publicly for everyone to access (including students).

Demo Setup/Preparing to Teach

The primary demo uses all playbooks to setup an Organizations, Users, Teams, Projects, Credentials, Roles, Job Templates, and finally a Job Workflow Template for approvals.

1. Create Github directory

```
[student@workstation ~]$ mkdir Github ; cd Github
```

2. Clone Repository

```
[student@workstation Github]$
Cloning into 'DO188_Demo'...
remote: Enumerating objects: 118, done.
remote: Counting objects: 100% (118/118), done.
remote: Compressing objects: 100% (75/75), done.
remote: Total 118 (delta 39), reused 88 (delta 12), pack-reused 0
Receiving objects: 100% (118/118), 734.20 KiB | 5.65 MiB/s, done.
Resolving deltas: 100% (39/39), done.
```

3. Change to **DO188_Demo** Directory and the Demo Setup directory

```
[student@workstation Github]$ cd DO188_Demo/Demos
```

QUAY Registry Password

There is a Quay repository installed on the **Registry** VM and the credentials are listed below.

https://registry.ocp4.example.com:8443/repository/

- **Username**: developer
- **Password**: developer

OCP Credentials and Passwords

https://console-openshift-console.apps.ocp4.example.com

Administrator

- **Username**: admin
- **Password**: redhat

Developer

- **Username**: developer
- **Password**: developer

1. Introduction and Overview of Containers

1.1. Introduction to Containers

1.1.1. Describing Containers

- **Container**: A process running from an image containing all required runtime dependencies and components and is self-contained and independent of host operating system and libraries.

A container engine creates a union filesystem which merges the layers of an immutable container image adding a thin readable/writable layer of ephemeral storage which is removed when the container is removed.

Containers use the following Linux components:

- **CGroups** - Partitioning process provided by linux kernel allowing limiting of resources
- **Kernel Namespaces** - Location provided by Linux kernel isolating specific resources from being visible to the entire system and processes. Namespace resources include: network interfaces, process IDs, mount points, IPC resources, and system hostname resources.
- **Secomp**: Limits how processes use system calls and provides a way to whitelist system calls.
- **SELinux**: Security Enhanced Linux providing mandatory access controls for processes. SELinux protects processes from each other and ensures these processes run in a confined space. SELinux has booleans controlling system-level items, File-Contexts, Network Contexts, and Process-Contexts that all work in conjunction to protect the system.

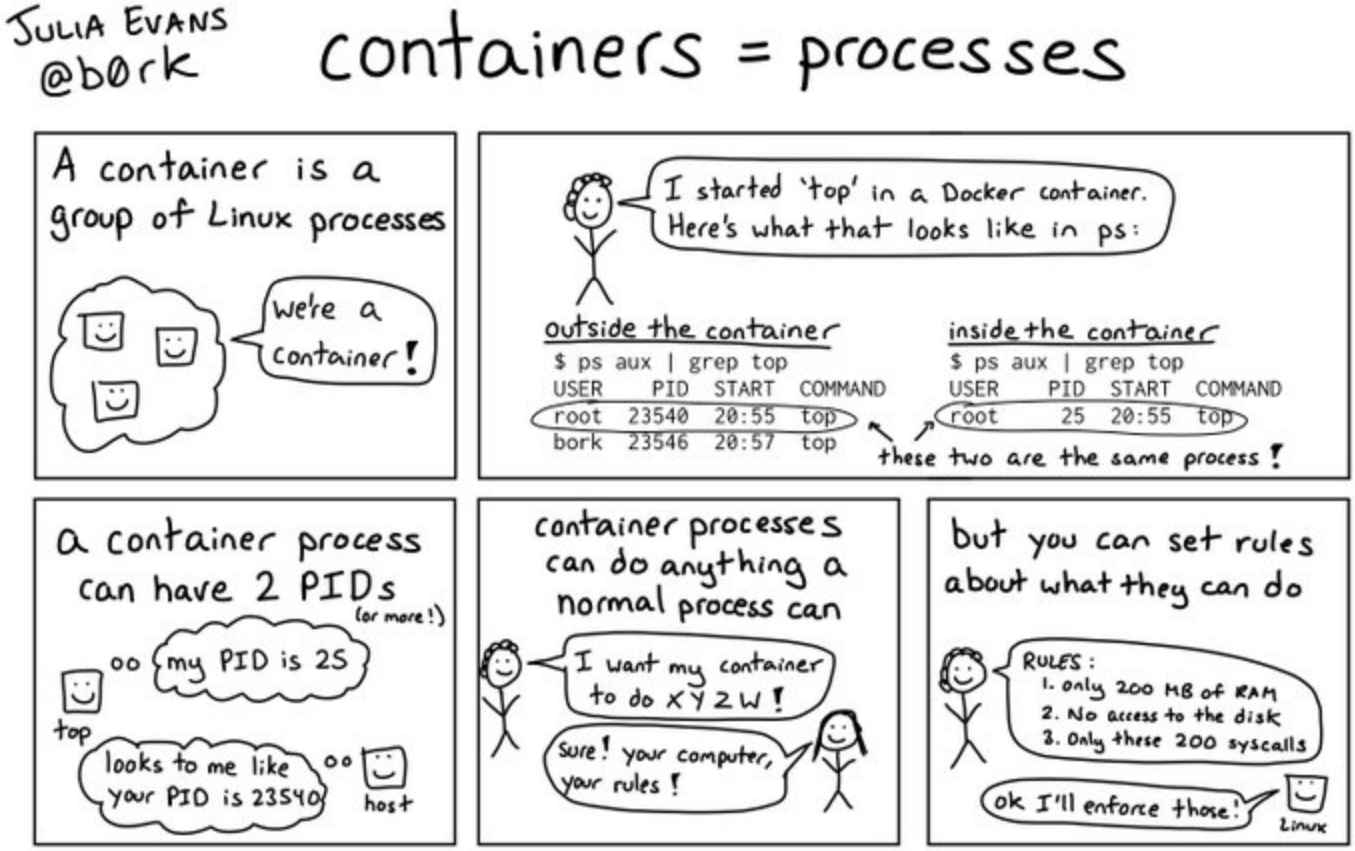

Figure 1. Containers are Processes - image from Julia Evans (@b0rk)

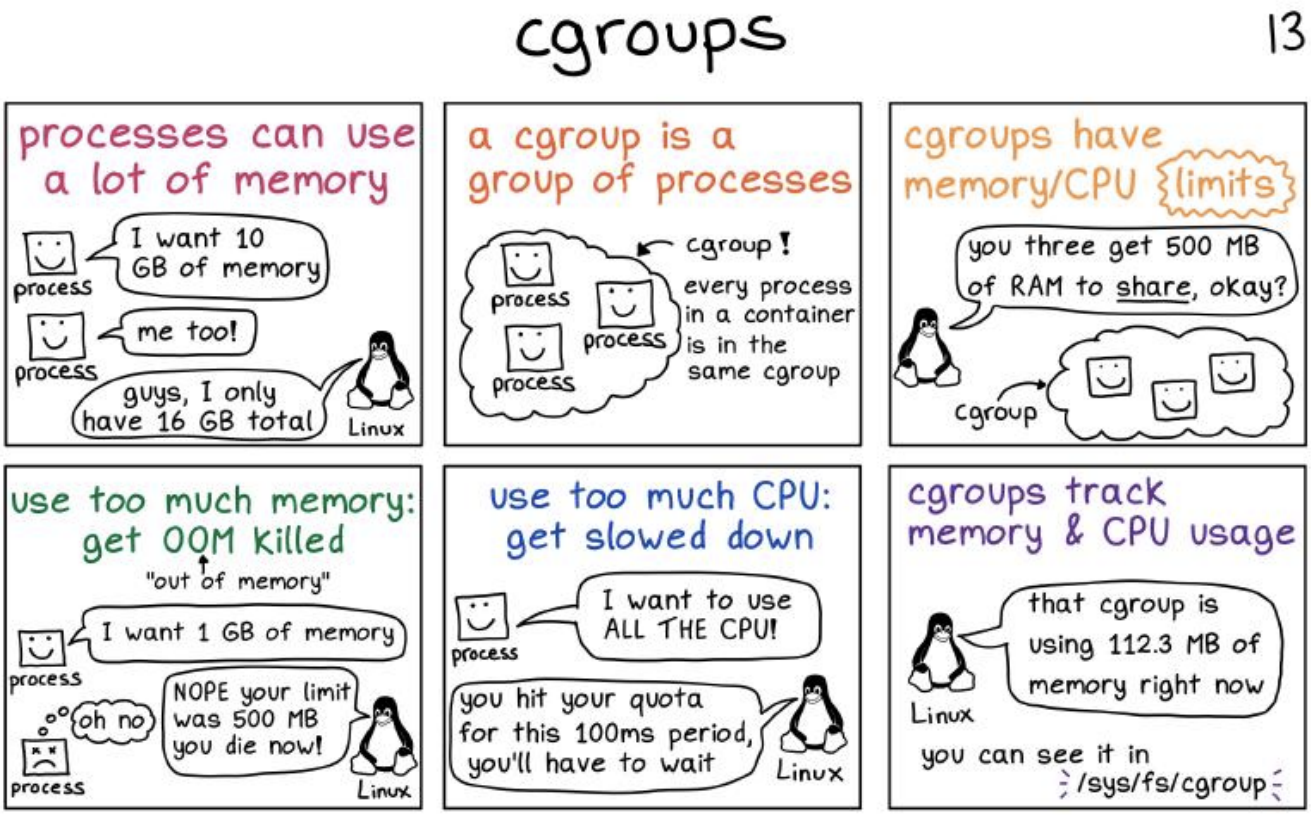

Figure 2. CGroup Overview - image from Julia Evans (@b0rk)

Julia Evans @b0rk

Containers aren't magic

These 15 lines of bash will start a container running the fish shell. Try it!
(download this script at bit.ly/containers-arent-magic)

```
wget bit.ly/fish-container -O fish.tar       # 1. download the image
mkdir container-root; cd container-root      #
tar -xf ../fish.tar                          # 2. unpack image into a directory
cgroup_id="cgroup_$(shuf -i 1000-2000 -n 1)" # 3. generate random cgroup name
cgcreate -g "cpu,cpuacct,memory:$cgroup_id"  # 4. make a cgroup &
cgset -r cpu.shares=512 "$cgroup_id"         #    set CPU/memory limits
cgset -r memory.limit_in_bytes=1000000000 \  #
      "$cgroup_id"                           #
cgexec -g "cpu,cpuacct,memory:$cgroup_id" \  # 5. use the cgroup
    unshare -fmuipn --mount-proc \           # 6. make + use some namespaces
    chroot "$PWD" \                          # 7. change root directory
    /bin/sh -c "                             #
        /bin/mount -t proc proc /proc &&     # 8. use the right /proc
        hostname container-fun-times &&      # 9. change the hostname
        /usr/bin/fish"                       # 10. finally, start fish!
```

Figure 3. Containers aren't Magic - image from Julia Evans (@b0rk)

Today, containers have evolved to use container images based on the Open Container Initiative (OCI) standards and specifications (https://opencontainers.org/).

Demo of **Containers Aren't Magic**

Here it is possible to perform a demo of the containers aren't magic to show how the Linux components work that make up containers without using Podman or Docker.

The files and script are available here: https://github.com/tmichett/OCP_Demos/tree/main/Containers_Arent_Magic

The is an excellent demo for courses with Admins, YMMV with a developer course.

Installing CGroup Tools

The default environment doesn't have CGroup tools installed so the **Containers aren't Magic** demo will fail. You MUST install CGroup tools prior to running the demo.

Listing 1. Installing CGroup Tools

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

Also, based on the system setup, in order to properly create the CGroups and run the items, this must be done as a privileged user on the system.

Listing 2. Running the Container Demo as a Privileged User

```
[student@workstation Containers_Arent_Magic]$ sudo
./containers-arent-magic.sh
```

Images versus Instances

Generally speaking, containers are divided into two components:

- **Container Images**: Image containing the data needed to run a container including all libraries and application components for the container.
- **Container Instance**: A running container based on a container image.

Containers

Generally speaking, when someone refers to a container, it is a **container instance**. A single container image can be used to create multiple **container instances**. Each container instance has the container image data which is ***immutable*** and one additional unique read/write layer added to the container which is ephemeral and will be removed from the system when the container is removed.

A running container is always based on a container image which is a "read-only" image and filesystem. The container runtime will add an overlay filesystem to the container when it starts up to provide a

temporary read/write layer to the container.

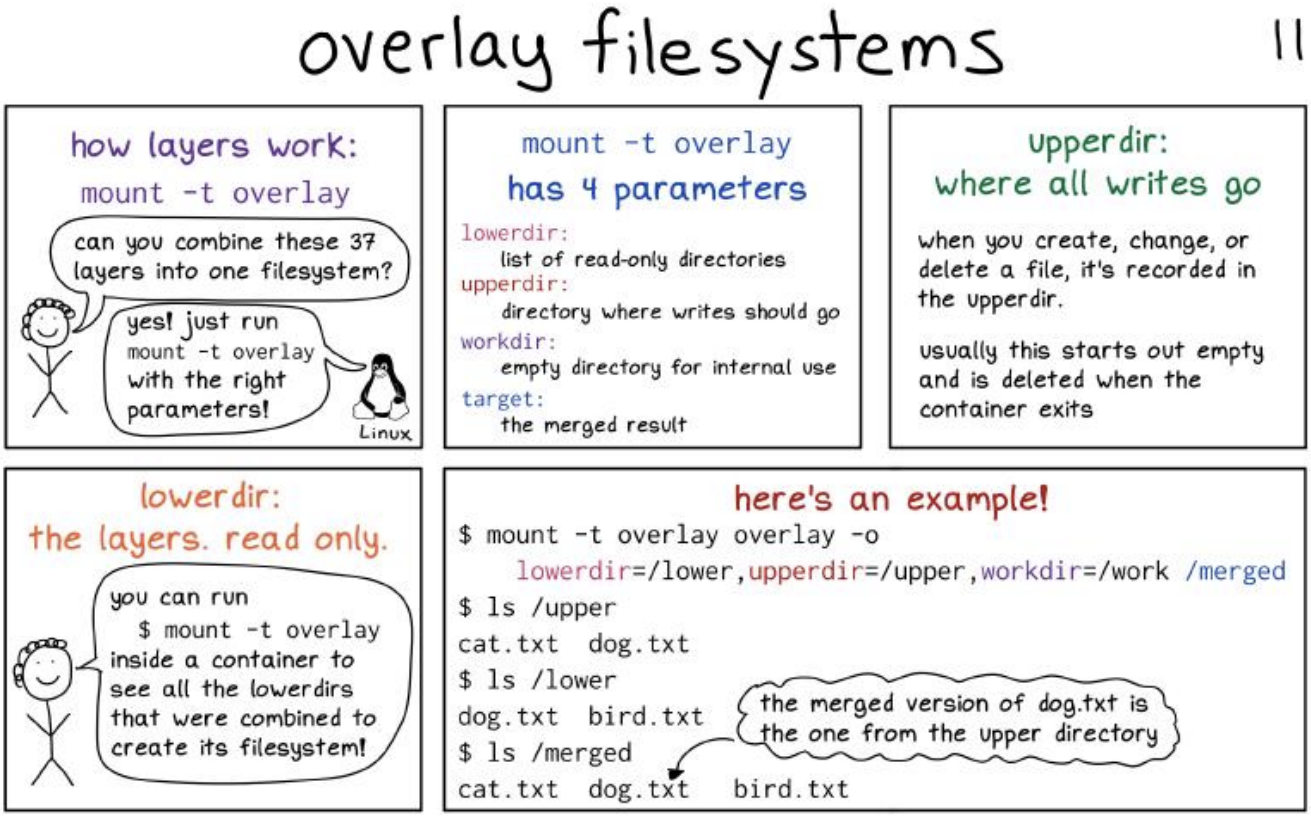

Figure 4. Overlay Filesystem - image from Julia Evans (@b0rk)

1.1.2. Comparing Containers to Virtual Machines

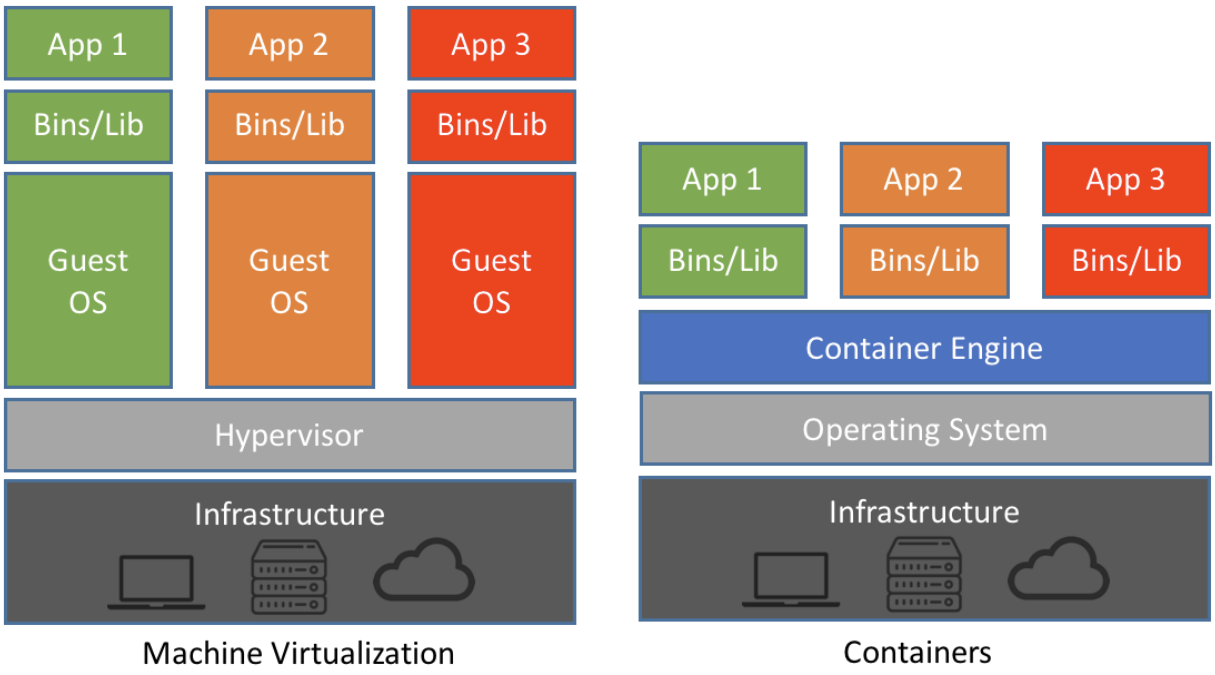

Figure 5. VM Container Comparison

The largest difference between containers and VMs ...

- Image size for VM image is measured typically in Gigabytes because OS and all components are installed, while container images are typically in Megabytes as they only contain the application and libraries needed to run the application.
- VMs require a hypervisor and generally require the same hypervisor, whereas container images can create new containers on any OCI-compliant container engine (providing an OCI compliant container image is used)

Deployment at Scale

Both VMs and Containers can be deployed at scale because they are both based on image templates. The difference is that containers require fewer resources than a virtual machine as VMs require complete operating systems and a hypervisor to be launched whereas containers only require a container engine to be launched making them more portable. Typically, containerized applications are more portable and have larger performance benefits when deployed at enterprise scale.

1.1.3. Development for Containers

Typically, containers are used as a solution for developers as they are easy to use for testing and deployment. Furthermore, based on container images being portable, it is easy to develop a container image and share the image knowing it will work using a supported container engine.

Podman

Red Hat Enterprise Linux uses **podman** to run containers and manage containers with the container engine. Podman will be discussed and used throughout this course and is the replacement for Docker

in Red Hat Enterprise Linux. Podman is meant to manage **pods** which can contain one or more containers. As part of this course, you will be using **podman** to manage individual containers directly, but it is important to note that at enterprise levels with Kubernetes and OpenShift, the smallest managed component is a **pod** which consists of one or more containers.

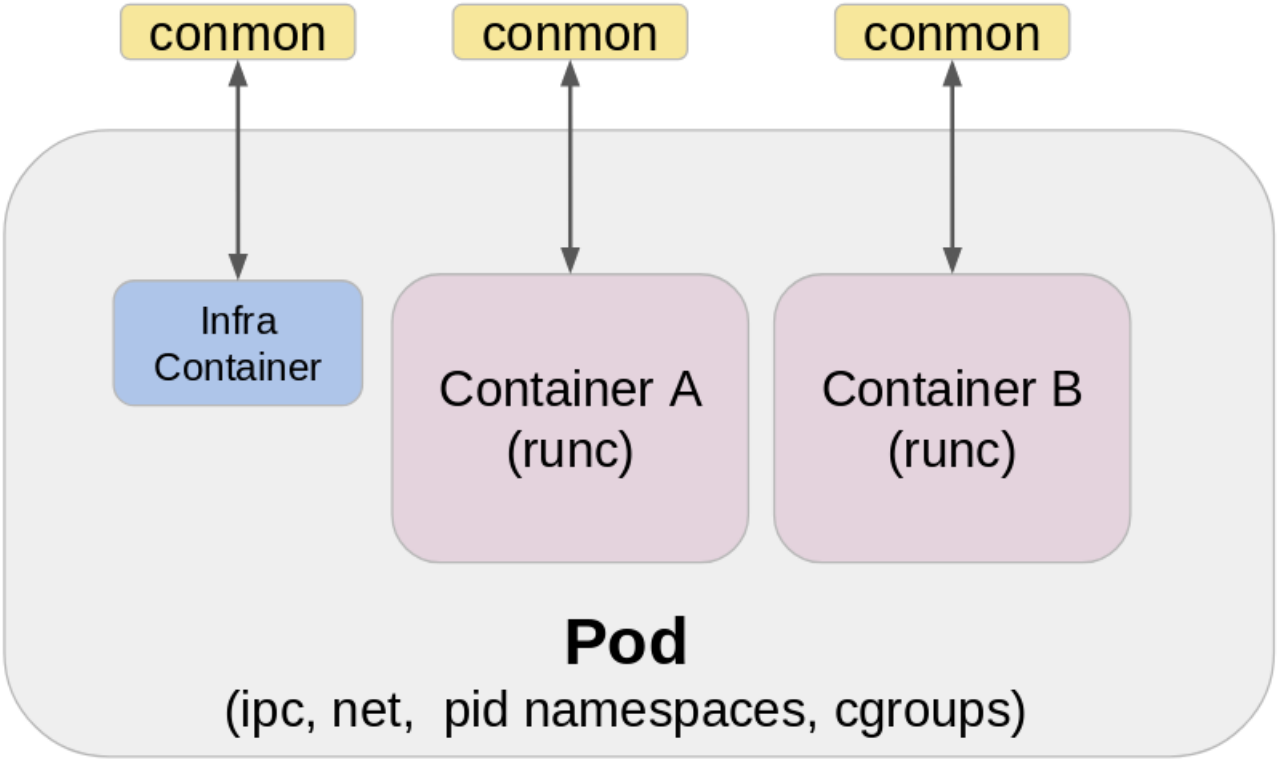

Figure 6. **podman** *Pod Architecture*

1.2. Introduction to Kubernetes and OpenShift

1.2.1. Kubernetes Overview

Kubernetes is an orchestration service meant to simplify container management. The smallest unit managed by Kubernetes is a **pod**.

Kubernetes Features

- **Service Discovery and Load Balancing**
- **Horizontal Scaling**
- **Self-Healing**
- **Automated Rollout**
- **Secrets and Configuration Management**
- **Operators**

1.2.2. Red Hat OpenShift Container Platform Overview

OpenShift is an extension of Kubernetes and built on top of Kubernetes. Red Hat OCP adds additional capabilities to Kubernetes which extend and enhance management for the enterprise.

Kubernetes to OCP as Kernel to Red Hat Enterprise Linux Comparisons

In a way, it can be conceptially thought of as if Kubernetes is the Linux Kernel, then Red Hat OpenShift would be equivalent to Red Hat Enterprise Linux. Whereas, Kubernetes is the "Kernel", Red Hat OpenShift Container Platform (OCP) is the "Distribution".

Red Hat OCP Features

- **Developer Workflow**
- **Routes**
- **Metrics and Logging**
- **Unified UI**

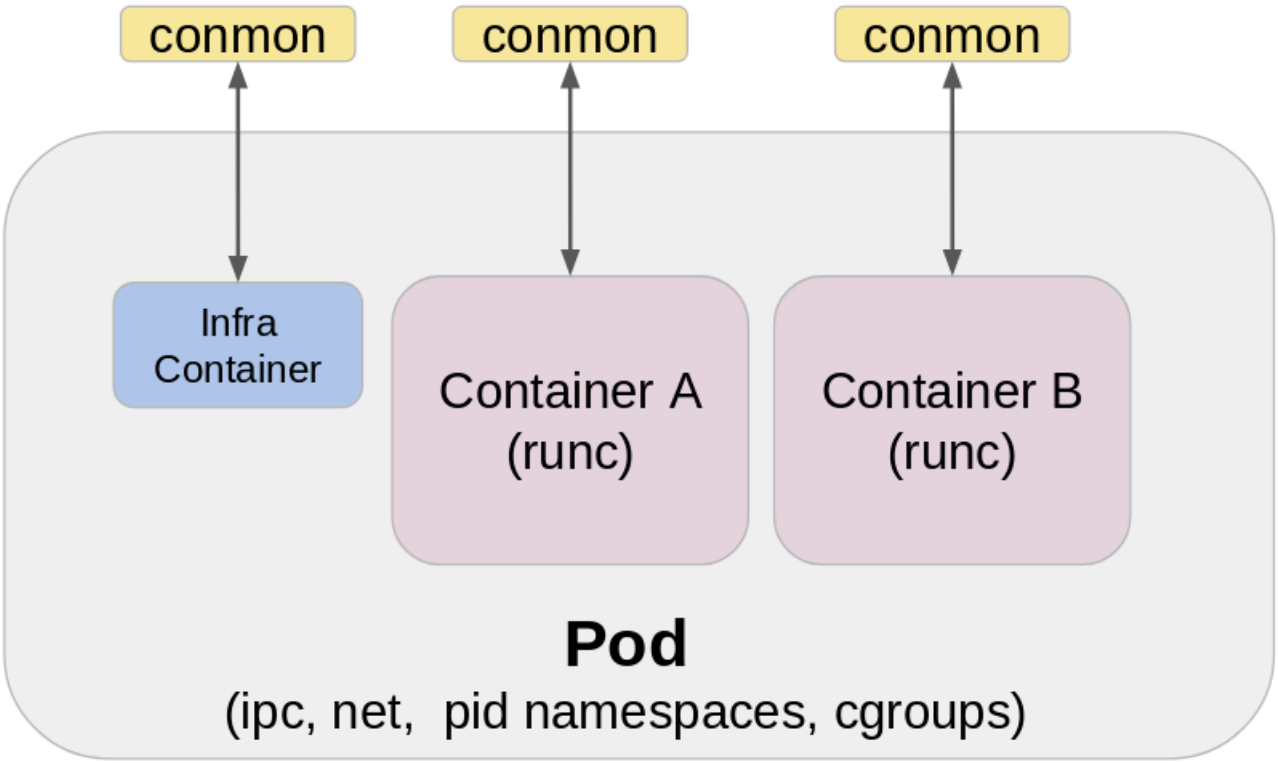

Figure 7. Podman **pod** *Architecture*

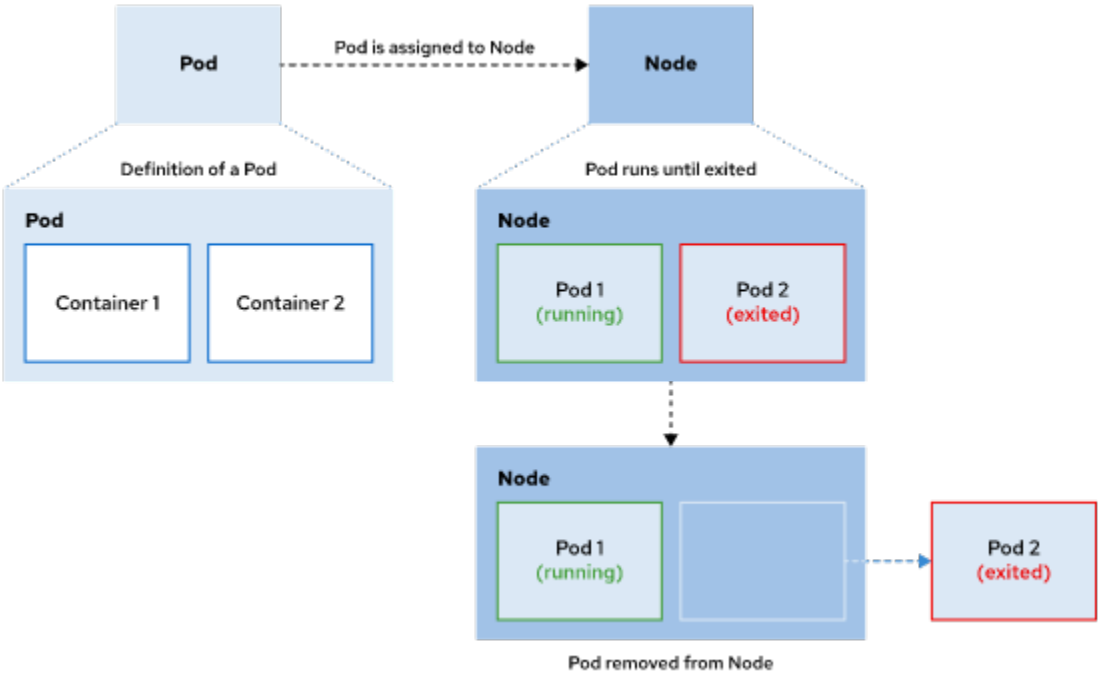

Figure 8. OCP Application and Container Lifecycle

2. Podman Basics

2.1. Creating Containers with Podman

2.1.1. Introducing Podman

Podman is the replacement for Docker and is used to manage containers locally. It allows running, building and deploying containers based on OCI container images. Unlike Docker, **podman** is daemonless. That means that **podman** containers can run without using a privileged daemon and this helps eliminate single-points-of-failure. Podman has a CLI as well as a REST-API.

Almost all **podman** commands are the same as Docker commands, so it is possible to create an alias on the system **alias docker=podman**. It should be understood that **podman** is the CLI manangement utility and it still relies on an underlying container runtime (**runc**, **crun**, **runv**, and others) which are OCI compliant container runtime environments. Podman containers can be either **root** containers or **non-root/rootless** containers.

podman *Command Reference*

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

podman *on Red Hat*

Podman uses **runc** as the OCI-compliant container runtime on RHEL-based systems.

Container Runtime References:

- **The tool that really runs your containers: deep dive into runc and OCI specifications**: https://mkdev.me/posts/the-tool-that-really-runs-your-containers-deep-dive-into-runc-and-oci-specifications
- **Using runc to explore the OCI Runtime Specification**: https://frasertweedale.github.io/blog-redhat/posts/2021-05-27-oci-runtime-spec-runc.html
- **Journey From Containerization To Orchestration And Beyond**: https://iximiuz.com/en/posts/journey-from-containerization-to-orchestration-and-beyond/

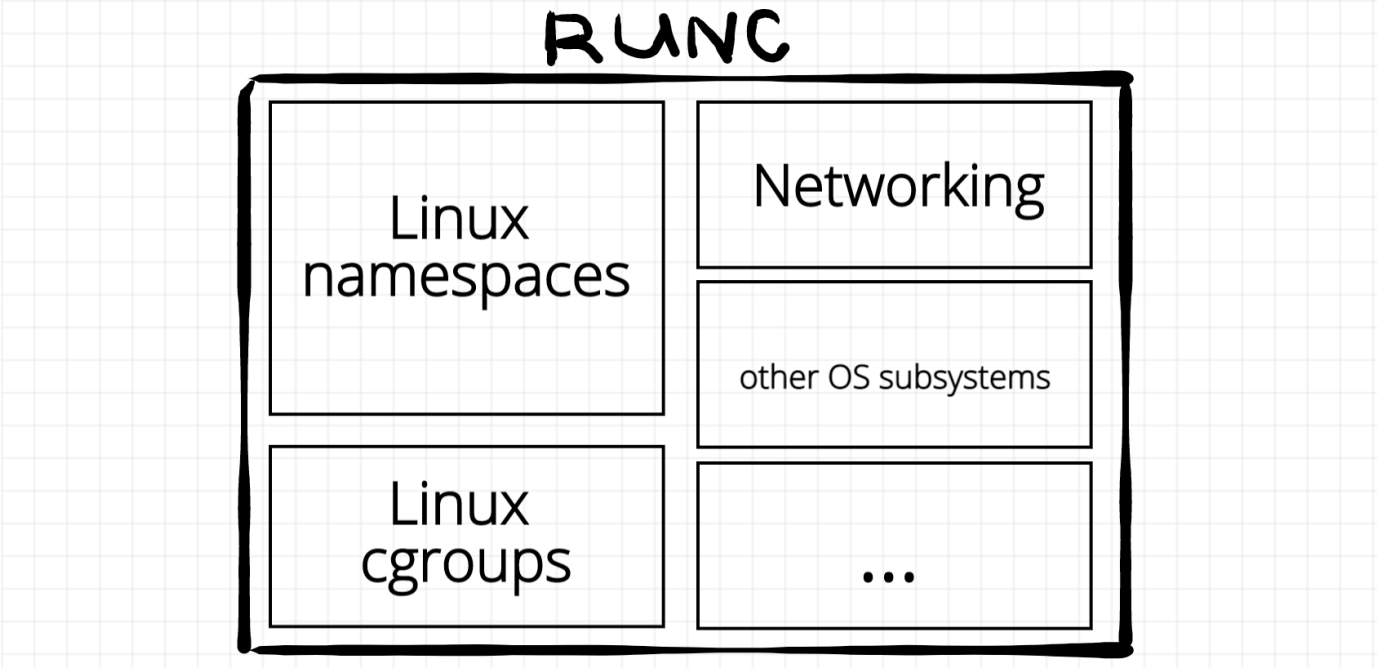

Figure 9. **runc** *Container Runtime*

2.1.2. Working with Podman

The **podman** package is installed with **Container Tools**, so it is possible to check the version in multiple ways.

Listing 3. RPM Version

```
[student@workstation ~]$ rpm -qa | grep -i podman
podman-catatonit-4.0.2-6.module+el8.6.0+14673+621cb8be.x86_64
podman-4.0.2-6.module+el8.6.0+14673+621cb8be.x86_64
podman-plugins-4.0.2-6.module+el8.6.0+14673+621cb8be.x86_64
podman-gvproxy-4.0.2-6.module+el8.6.0+14673+621cb8be.x86_64
```

Listing 4. Using the **podman** *Command*

```
[student@workstation ~]$ podman -v
podman version 4.0.2
```

Table 1. **podman** *Command Options Covered*

Command Option	Purpose
run	Run and launch a container based on a given image

Command Option	Purpose
run --name <*Given_Name*>	Run and launch a container based on a given image, but name the container based on the <*Given_Name*>
run --rm	Run and launch a container based on a given image, but removes the container when exited
ps	List running/active containers
ps -a or **ps --all**	List all containers regardless of state

2.1.2.1. Pulling and Displaying Images

We learned earlier, containers must have a container image to run. The **podman pull** command will download a container image.

```
[student@workstation ~]$ podman pull quay.io/tmichett/ibm-demo-httpd:v1.0
```

Pulling an image will store it locally on the system and can be viewed with **podman images** command.

```
[student@workstation ~]$ podman images
REPOSITORY                       TAG         IMAGE ID      CREATED       SIZE
localhost/git_demo_demo          latest      2c7ed9b2d91a  8 days ago    249 MB
quay.io/tmichett/ibm-demo-httpd  v1.0        c962db9155dd  2 months ago  258 MB
```

REMINDER: *Differences between containers and container images*

A container image contains the application and all dependencies for that application to run. A container image is required for a container to run and a container cannot exist without a container image. A container, on the other hand, is an isolated process (runtime environment) based on a container image. The isolation for a container ensures it doesn't disrupt other containers or system processes.

2.1.2.2. Running and Displaying Containers

The **podman run** command is used to launch a container from an image. At a minimum, it will require the container image being launched.

```
[student@workstation ~]$ podman run registry.ocp4.example.com:8443/ubi8/ubi-minimal:8.5
echo "I'm the Demo from the container"

I'm the Demo from the container ①
```

① The output from the container is: **I'm the Demo from the container**

The **podman ps** command gives status of all running containers while the **podman ps -a** command gives status of all containers.

```
[student@workstation ~]$ podman ps
CONTAINER ID  IMAGE       COMMAND     CREATED     STATUS      PORTS       NAMES

[student@workstation ~]$ podman ps -a
CONTAINER ID  IMAGE                                           COMMAND
CREATED        STATUS                     PORTS       NAMES
d901895f9102  registry.ocp4.example.com:8443/ubi8/ubi-minimal:8.5  echo I'm the Demo...
2 minutes ago  Exited (0) 2 minutes ago               crazy_kalam
```

As container management can get tricky at times for cleanup, it can be a good tip to use the **--rm** option which will remove a container when it exits, as the **--rm** is meant to remove. It is also possible to combine the **--name** option with the run command to give the container a meaningful name.

```
[student@workstation ~]$ podman run --rm --name Demo_Container
registry.ocp4.example.com:8443/ubi8/ubi-minimal:8.5 echo "I'm the Demo from the
container"
I'm the Demo from the container

[student@workstation ~]$ podman ps -a
CONTAINER ID  IMAGE       COMMAND     CREATED     STATUS      PORTS       NAMES
```

podman *Output Formats*

The output displayed in the screen can be formatted as the **Go** template which is standard, however, it is also possible to use the **--format=format** with most **podman** commands to display the output in JSON format to make things easier to query with something like **jq**.

2.1.2.3. Exposing Containers

The **podman** run command also has additional options which will require the container to expose or allow network traffic as well as running the container in the background like a service daemon.

Table 2. **podman run** *Additional Options*

Podman Run Option	Purpose
-p <Local_Port>:<Container_Port>	Exposes and forwards the local port network traffic to the specified container port
-d	Runs the container in the background similar to a daemon.

The commands in the above table will be covered in more detail, but they can be combined to launch things like a webserver or other network workloads.

```
[student@workstation ~]$ podman run -d -p 8080:80 --name travis-demo
quay.io/tmichett/httpd-custom-demo-new:v1.0
```

```
[student@workstation ~]$ curl localhost:8080
I am custom material for the DO180 course
```

2.1.2.4. Using Environment Variables

Lastly, we discuss the use of passing environment variables to containers. This allows a way to provide information into a container without placing it as part of the code. This can be useful when using container images like databases or something else where you have a generic image and you customize the container during launch.

Using **podman run -e** <Var_Name>=<Value>, it is possible to pass values for <**Var_Name**>.

2.2. DEMO: Creating Containers with Podman

Registry Requires Authentication

Listing 5. **podman login**

```
[student@workstation ~]$ podman login registry.ocp4.example.com:8443 -u
developer -p developer
Login Succeeded!
```

Example 1. DEMO - Using Podman

1. Explore **podman** to determine versions and downloaded images

```
[student@workstation do188]$ podman -v
podman version 4.0.2
```

Listing 6. Available Images

```
[student@workstation do188]$ podman images --format "table {{.Repository}}
{{.Tag}}"
REPOSITORY                                                  TAG
registry.ocp4.example.com:8443/ubi8/httpd-24                latest
registry.ocp4.example.com:8443/ubi8/ubi                     latest
registry.ocp4.example.com:8443/ubi9/ubi                     9.0.0-1468
registry.ocp4.example.com:8443/ubi8/ubi-minimal             8.5
registry.ocp4.example.com:8443/redhattraining/hello-world-nginx  v1.0
```

2. Use Podman to run a container based on a given image

```
[student@workstation do188]$ podman run registry.ocp4.example.com:8443/ubi8/ubi-
minimal:8.5 echo "I'm the Demo from the container"
I'm the Demo from the container
```

Listing 7. Passing Environment Variables to a Container

```
[student@workstation do188]$ podman run -e ENV_VAR1='Demo variable 1' -e ENV_VAR2
='Demo variable 2' registry.ocp4.example.com:8443/ubi8/ubi-minimal:8.5 printenv
ENV_VAR1 ENV_VAR2
Demo variable 1
Demo variable 2
```

3. Listing Containers

Listing 8. Running Containers

```
[student@workstation do188]$ podman ps
CONTAINER ID  IMAGE       COMMAND     CREATED     STATUS      PORTS       NAMES
```

Listing 9. All Containers

```
[student@workstation do188]$ podman ps -a
CONTAINER ID  IMAGE                                                    COMMAND
CREATED             STATUS                         PORTS       NAMES
8916d98e12e5  registry.ocp4.example.com:8443/ubi8/ubi-minimal:8.5  echo I'm the
Demo...  10 minutes ago      Exited (0) 10 minutes ago
vibrant_borg
60e09d073a85  registry.ocp4.example.com:8443/ubi8/ubi-minimal:8.5  printenv
ENV_VAR1...  9 minutes ago       Exited (0) 9 minutes ago
relaxed_nash
e8aeaa498e47  registry.ocp4.example.com:8443/ubi8/ubi-minimal:8.5  echo I'm the
Demo...  2 minutes ago       Exited (0) 2 minutes ago
inspiring_payne
3e5d7231a43d  registry.ocp4.example.com:8443/ubi8/ubi-minimal:8.5  printenv
ENV_VAR1...  About a minute ago  Exited (0) About a minute ago
friendly_kirch
```

Container Management

It is important to note that unless there is a cleanup or the **podman run** command contains the **--rm** option, when the container finishes a run, it will remain on the system in the "Exited" state. More on this in the **Managing the Container Lifecycle** section.

4. Run a container in the background exposing network traffic (in this case HTTP) traffic to host a website.

```
[student@workstation do188]$ podman run -d -p 8080:80 --name travis-demo
quay.io/tmichett/httpd-custom-demo-new:v1.0
```

Listing 10. Verify Website Data

```
[student@workstation do188]$ curl localhost:8080
I am custom material for the DO180 course
```

5. Accessing **podman** results as JSON

```
[student@workstation do188]$ podman images --format=json | jq '. | map(.Names)'
[
  [
    "registry.ocp4.example.com:8443/ubi8/httpd-24:latest"
  ],
  [
    "registry.ocp4.example.com:8443/ubi8/ubi:latest"
  ],
  [
    "registry.ocp4.example.com:8443/ubi9/ubi:9.0.0-1468"
  ],
  [
    "registry.ocp4.example.com:8443/ubi8/ubi-minimal:8.5"
  ],
  [
    "quay.io/tmichett/httpd-custom-demo-new:v1.0"
  ],
  [
    "registry.ocp4.example.com:8443/redhattraining/hello-world-nginx:v1.0"
  ]
]
```

jq

The **jq** command can be used to filter and process JSON output. In this instance, we wanted the **Names** field, so the **map(.Names)** was used to only return those values.

6. Cleanup Running and stopped containers

```
[student@workstation References]$ podman rm -af
```

2.3. Container Networking Basics

2.3.1. Container Networking Basics

Container networking is setup by default with the **podman** network. By default, all containers are attached to this network and can use it for communicating to each other and the local machine can communicate to the containers through this network. It is possible to use the **podman network** command to create isolated networks for communication between containers.

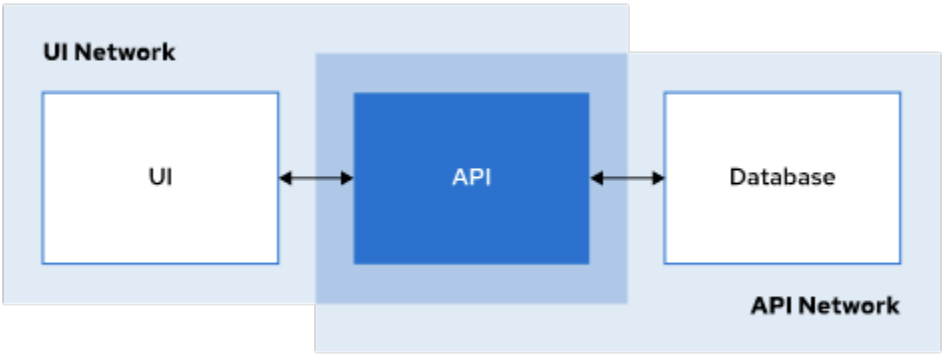

Figure 10. Example of Containers on Different Networks

Default Network Configuration

Earlier versions of **podman** kept network configuration for default networks in **/etc/cni/net.d/87-podman-bridge.conflist**. This is no longer true for the default network as **netavark** us memory-only. Podman default networks do not support DNS resolution either based on backwards compatability with Docker.

Listing 11. Obtaining Default Network Settings with Podman v4

```
[root@workstation ~]# podman network inspect podman

[student@workstation ~]$ podman network inspect podman
```

https://github.com/containers/podman/blob/main/docs/tutorials/basic_networking.md

2.3.2. Managing Podman Networks

Table 3. **podman network** *Commands*

Command	Purpose
podman network create	Creates new podman network

Command	Purpose
podman network ls	Lists podman networks and provides a summary
podman network inspect	Provides detailed JSON output on a network
podman network rm	Removes a network
podman network prune	Removes networks not un use by running containers
podman network connect	Creates a connection between an already running container and an existing container network

2.3.3. Enabling Domain Name Resolution

The default **podman** network has the DNS system disabled. In order to use DNS, a new Podman network must be created and connected to containers. When using DNS the name assigned to the container is the DNS name.

DNS and Default Network

DNS is disabled by default on the **podman** network. It is possible to override the default network and enable DNS, but that is not covered as part of this course.

2.3.4. Connecting Containers

Containers can be connected to one or more Podman networks. Once connected to a network, containers can communicate to other containers on the network via IP address or DNS resolution.

2.4. DEMO: Container Networking Basics

Example 2. DEMO - Using Podman Networks

1. Explore **podman** to determin versions and downloaded images

```
[student@workstation ~]$ podman network create demo-net
demo-net
```

2. Launch an HTTP Webserver

```
[student@workstation ~]$ podman run -d -p 9080:80 --name travis-demo2 --net demo-
net  quay.io/tmichett/httpd-custom-demo-new:v1.0
40bfc07475672d28d63b159be2123f771f62c0590327a92f38a91fe72880c646
```

3. Attempt to verify website by DNS name **travis-demo2**

```
[student@workstation ~]$ curl localhost:9080
I am custom material for the DO180 course
[student@workstation ~]$ curl travis-demo2
curl: (6) Could not resolve host: travis-demo2
```

4. Launch RHEL8 Container

```
[student@workstation ~]$ podman run --net demo-net
registry.ocp4.example.com:8443/ubi8/ubi-minimal:8.5 curl -s travis-demo2
I am custom material for the DO180 course
```

2.5. Accessing Containerized Network Services

Section Info Here

2.5.1. Port Forwarding

2.5.1.1. List Port Mappings

2.5.2. Networking in Containers

2.6. DEMO: Accessing Containerized Network Services

Example 3. DEMO - Using Podman Port Forwarding

1. Launch a webserver and port forward port **10080** to port **80**

```
[student@workstation ~]$ podman run -d -p 10080:80 --name travis-demo3 --net demo-
net  quay.io/tmichett/httpd-custom-demo-new:v1.0
5c422becbd7d4f8e349c2300d98811421d73f4b7b05f3023cdfccd5631c835d4
```

2. View ports in use using the **podman port** command

```
[student@workstation ~]$ podman port -a
5c422becbd7d    80/tcp -> 0.0.0.0:10080
93a63219de88    80/tcp -> 0.0.0.0:9080
```

3. View the IP Address and other Network Settings

```
[student@workstation ~]$ podman inspect travis-demo3 --format=json | jq '.
|map(.NetworkSettings)|map(.Networks)'
```

4. Retrieve information based on IP address

```
[student@workstation ~]$ podman run --net demo-net
registry.ocp4.example.com:8443/ubi8/ubi-minimal:8.5 curl -s 10.89.0.7
I am custom material for the DO180 course
```

5. Demonstrate on default container network

```
[student@workstation ~]$ podman run -d -p 11080:80 --name travis-demo4
quay.io/tmichett/httpd-custom-demo-new:v1.0
```

Listing 12. Obtaining IP Address

```
[student@workstation ~]$ podman inspect travis-demo4 --format=json | jq '.
|map(.NetworkSettings)|map(.Networks)'
```

No IP Address

A "rootless" container uses SLIRP Network and doesn't assign an IP Address to the container on the default network namespace. Furthermore, Podman containerized DNS is also not available

6. Demonstrate on default container network as "root" conatainer

```
[student@workstation ~]$ sudo podman run -d -p 12080:80 --name travis-demo5
quay.io/tmichett/httpd-custom-demo-new:v1.0
```

Listing 13. Obtaining IP Address

```
[student@workstation ~]$ sudo podman inspect travis-demo5 --format=json | jq
'.|map(.NetworkSettings)|map(.Networks)'
```

7. Verify access from workstation machine

```
[student@workstation ~]$ curl 10.88.0.3
I am custom material for the DO180 course
```

Since we are using the default network from Podman, we have direct access to that network space from the **workstation** machine. Therefore we can access the data within the container by using the container IP address and not relying on port-forwarding.

8. Cleanup all Containers

Listing 14. Cleanup Root Containers

```
[student@workstation ~]$ sudo podman rm -af
94348119c10655d62bf24565b4b1f0bcde18b7f8f96a9a926e4a5415addf901b
```

Listing 15. Cleanup Rootless Containers

```
[student@workstation ~]$ podman rm -af
2e27deeb333cd2cd4cc47b3dc1b2bbffece56559224be222f976432dfb3068f1
3e50448096d5d7caa4a4af9ff5aeb7c6a32b4ded12a2ceb519e35b11225ee2f3
```

2.7. Accessing Containers

2.7.1. Container Transparency

2.7.2. Start Processes in Containers

2.7.3. Open an Interactive Session in Containers

2.7.4. Copy Files in and Out of Containers

2.8. DEMO: Accessing Containers

Example 4. DEMO - Using **exec** *and* **cp** *to Access Containers*

1. Run the HTTPD container

```
podman run -d -p 2080:80 --name demo-web quay.io/tmichett/httpd-custom-demo-
new:v1.0
```

2. View website content

```
[student@workstation ~]$ curl localhost:2080
I am custom material for the DO180 course
```

3. Modify content for website

Listing 16. View content

```
[student@workstation ~]$ podman exec demo-web cat /var/www/html/index.html
I am custom material for the DO180 course
```

Listing 17. Interactively Edit the Content

```
[student@workstation ~]$ podman exec -it demo-web /bin/bash
bash-4.4#

bash-4.4# echo "This is a demo for DO188" > /var/www/html/index.html

bash-4.4# curl localhost
This is a demo for DO188
```

4. View modified website content from workstation.

```
[student@workstation ~]$ curl localhost:2080
This is a demo for DO188
```

5. Copy new website into container

```
[student@workstation ~]$ podman cp ~/Github/OCP_Demos/DO188_Web/index.html demo-
web:/var/www/html/
[student@workstation ~]$ podman cp ~/Github/OCP_Demos/DO188_Web/penguin3.jpeg
demo-web:/var/www/html/
```

6. View website

```
[student@workstation ~]$ curl localhost:2080
```

Instructor Demo - Open in Firefox (Remotely)

Listing 18. Temporarily Add Port

```
[student@workstation ~]$ sudo firewall-cmd --add-port=2080/tcp
success
```

Then open in a GUI Web Browser!!

2.9. Managing the Container Lifecycle

Section Info Here

2.9.1. Container Lifecycle

2.9.2. Inspect a Container

2.9.3. Stop a Container

2.9.3.1. Stop a Container Gracefully

2.9.3.2. Stop a Container Forcefully

2.9.3.3. Pause a Container

2.9.4. Restarting a Container

2.9.5. Remove a Container

2.10. DEMO: Accessing Containers

*Example 5. DEMO - **podman** and Container Lifecycles*

1. Run a simple container using **podman**

```
[student@workstation ~]$
[student@workstation ~]$ podman run --name simple-demo -e ENV_VAR1='Demo variable
1' -e ENV_VAR2='Demo variable 2' registry.ocp4.example.com:8443/ubi8/ubi-
minimal:8.5 printenv ENV_VAR1 ENV_VAR2
Demo variable 1
Demo variable 2
```

2. Run a **podman ps** to show containers

```
[student@workstation ~]$ podman ps
CONTAINER ID  IMAGE                                          COMMAND
CREATED      STATUS          PORTS                  NAMES
42b0e858b01f  quay.io/tmichett/httpd-custom-demo-new:v1.0  /bin/sh -c /usr/s...  2
hours ago  Up 2 hours ago  0.0.0.0:2080->80/tcp  demo-web
```

Only the HTTPD **demo-web** container is running and displayed

3. Run a **podman ps -a** to show all containers

```
[student@workstation ~]$ podman ps -a --format "{{.Image}} {{.Names}} {{.Status}}"
quay.io/tmichett/httpd-custom-demo-new:v1.0 demo-web Up 2 hours ago
registry.ocp4.example.com:8443/ubi8/ubi-minimal:8.5 simple-demo Exited (0) 6
minutes ago
```

Using **--format** *with Podman*

To return specific results, in addition to JSON, it is possible to format data and pull certain fields from the table.

https://docs.podman.io/en/latest/markdown/podman-ps.1.html

4. Run a simple container using **podman run --rm** demonstrating the **--rm** option

```
[student@workstation ~]$
[student@workstation ~]$ podman run --rm --name simple-demo2 -e ENV_VAR1
='Container erased' -e ENV_VAR2='after this!!'
registry.ocp4.example.com:8443/ubi8/ubi-minimal:8.5 printenv ENV_VAR1 ENV_VAR2
Container erased
after this!!
```

Automatically Cleaning Up Exited Containers

We have learned that the **--rm** command will remove a stopped container. We also know that it can remove a running container when the **-f** or **--force** is used, so what does the **--rm** do when combined with the **podman run** command?

Simply put, the **--rm** command will attempt to remove the container. While the container is running, the command does nothing, as soon as the container "exits", the **--rm** causes the exited container to be removed.

5. Verify that the **simple-demo2** doesn't exist.

```
[student@workstation ~]$ podman ps -a --format "{{.Image}} {{.Names}} {{.Status}}"
quay.io/tmichett/httpd-custom-demo-new:v1.0 demo-web Up 3 hours ago
registry.ocp4.example.com:8443/ubi8/ubi-minimal:8.5 simple-demo Exited (0) About
an hour ago
```

3. Container Images

3.1. Container Image Registries

3.1.1. Container Registries

3.1.2. The Containerfile

3.1.3. Red Hat Registry

3.1.3.1. Useful Container Images

3.1.4. Quay.io

3.1.5. Manage Registries with Podman

3.1.6. Manage Registry Credentials with Podman

3.2. DEMO: Container Image Registries

Example 6. DEMO - Creating a New Repository and Container Image from Github

1. Login to Quay.io

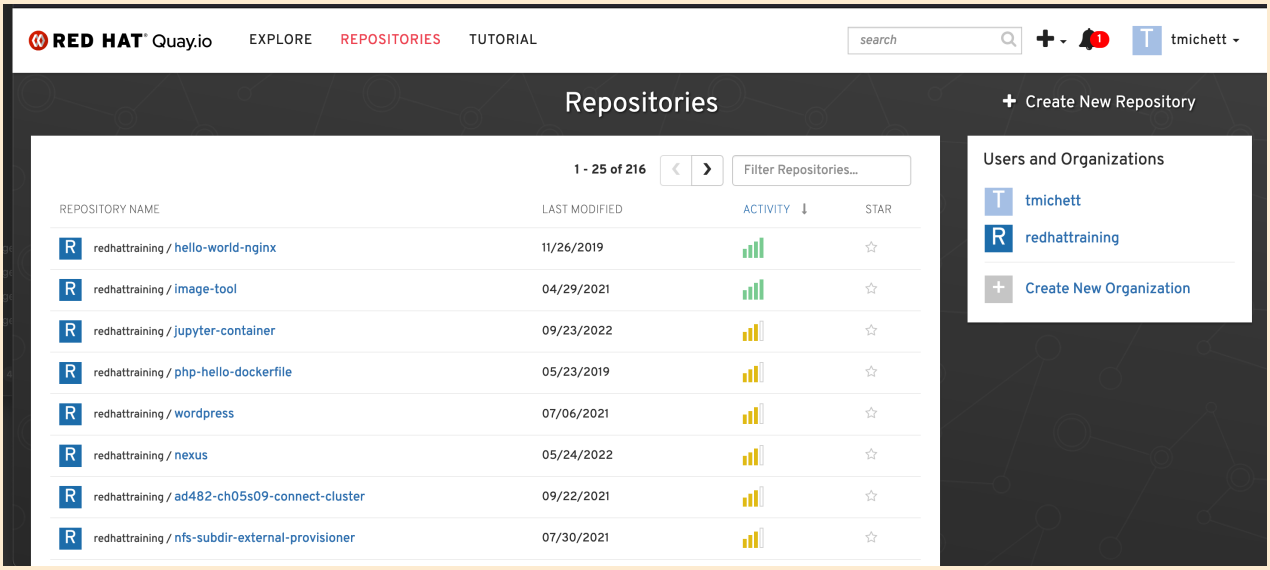

Figure 11. Quay Repositories

2. Click "+ **Create New Repository**" and select "New Repository"
   - Repository Name: **do188_git_demo**
   - Visibility: **public**

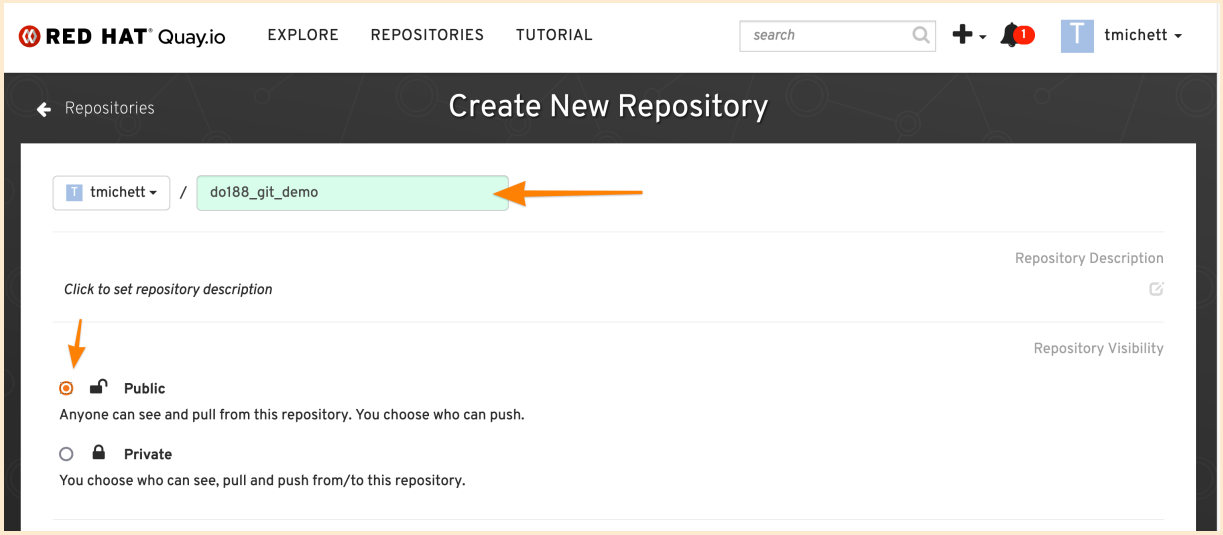

Figure 12. Repository Name and Visibility

   - Select **Link to a Github Repository Push**
   - Click **Create Public Repository**

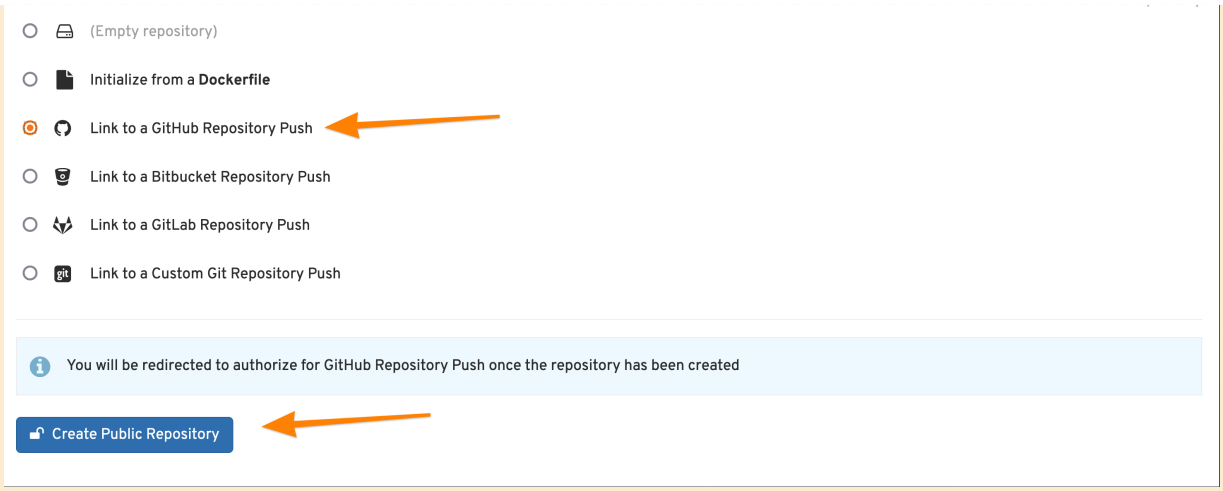

Figure 13. Linking to Version Control Dockerfile/Containerfile

3. Select the Organization containing the Github repository you wish to use and click "Continue"

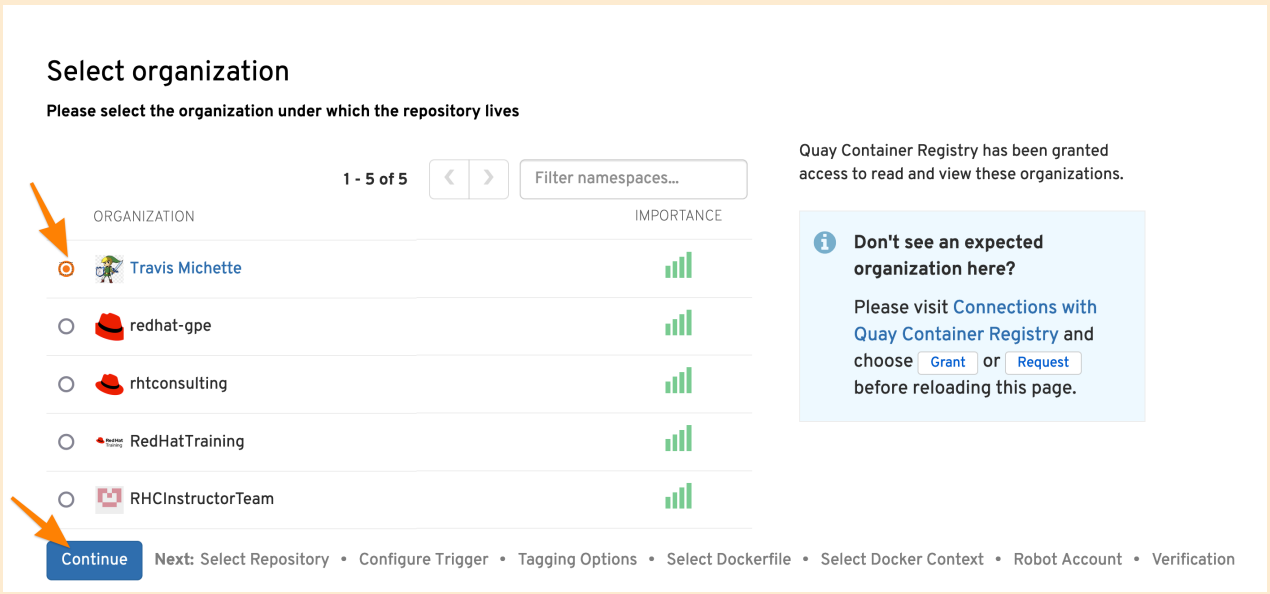

Figure 14. Github Organization Selection

4. Select the Repository you wish to use and click **Continue**

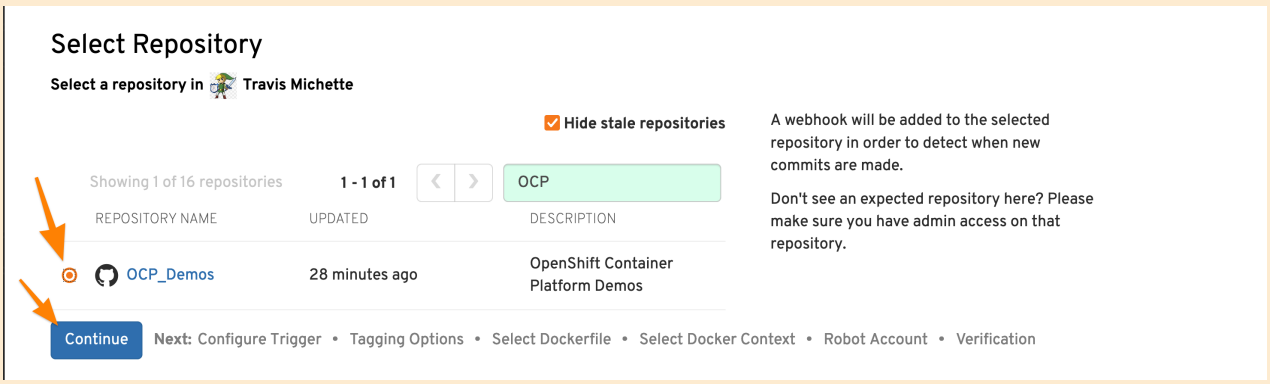

Figure 15. Github Repository Selection

5. Select the automated build trigger options and click **Continue**

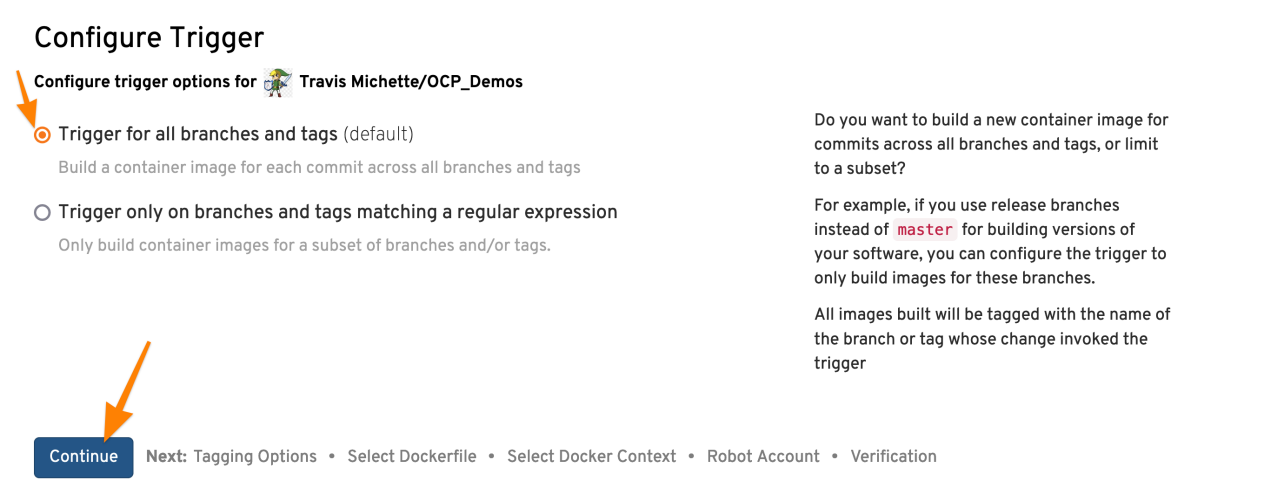

Figure 16. Github Branch and Tag Trigger Option Selection

6. Select the tagging options and click **Continue**

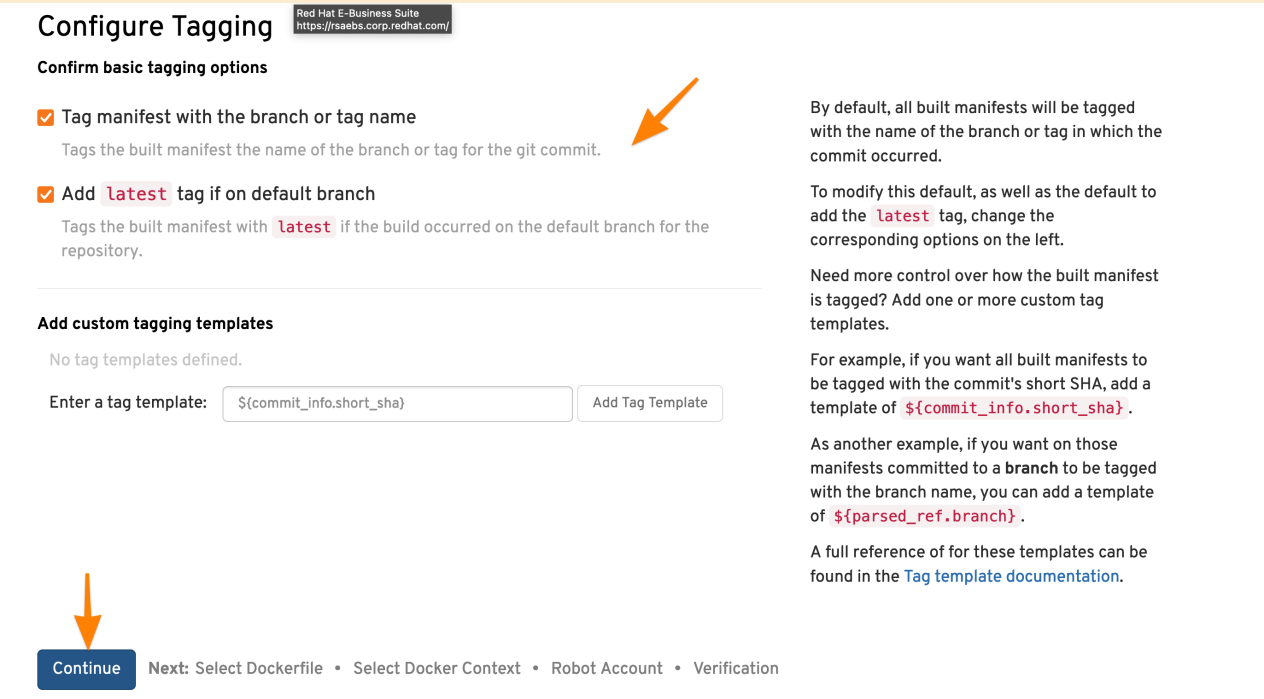

Figure 17. Github Container Tagging Option Selection

7. Select the name and path of the Dockerfile/Containerfile and click **Continue**

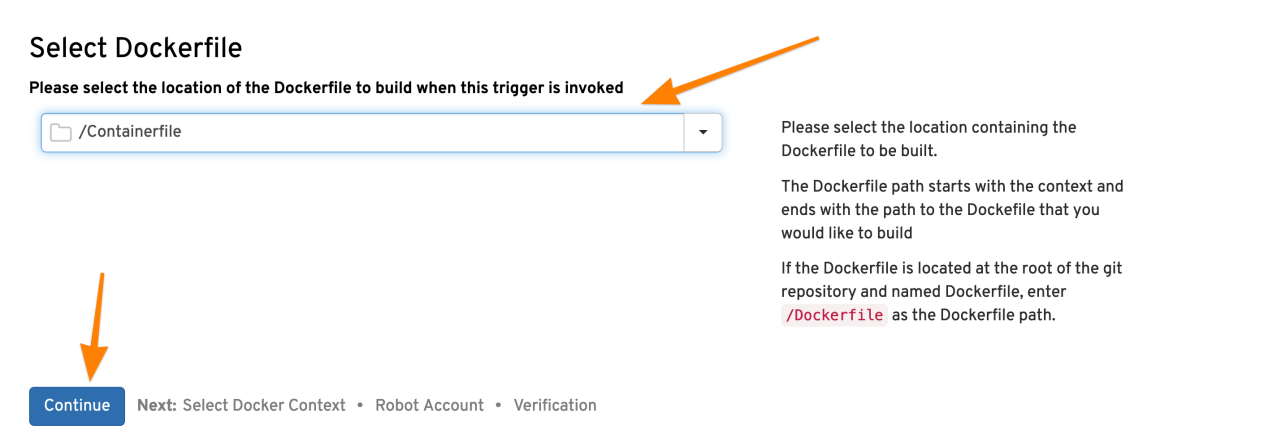

Figure 18. Github Dockerfile/Containerfile Selection

8. Select the Context directory and click **Continue**

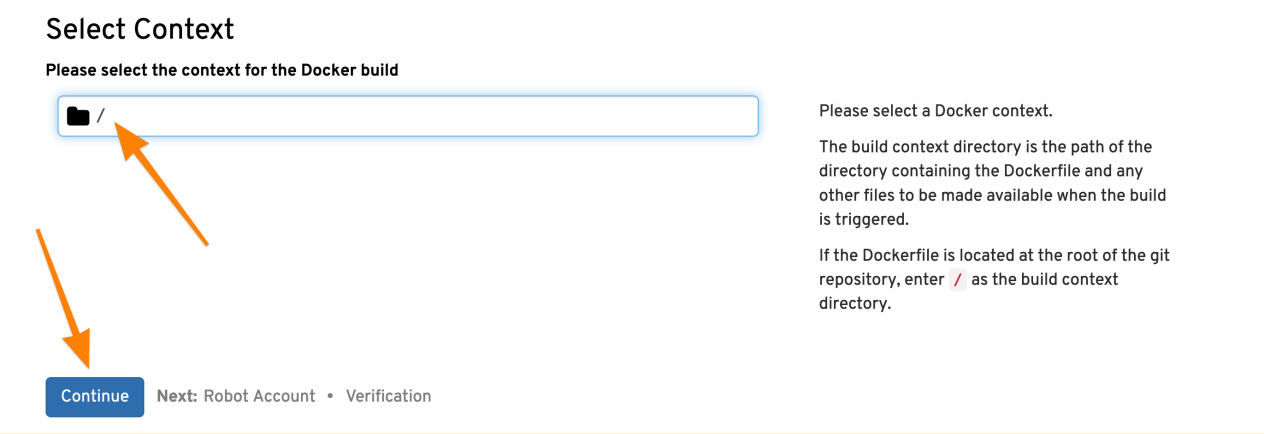

Figure 19. Container Context Build Selection

9. Skip the "Robot Account" and click **Continue**
10. Verify the build trigger and click **Continue**

Figure 20. Verification of All Options

Authorizing Service

You may be required to authorize the service (Quay) to contact Github. Additionally, when creating the trigger, it will add and updated SSH keys to the source control repository.

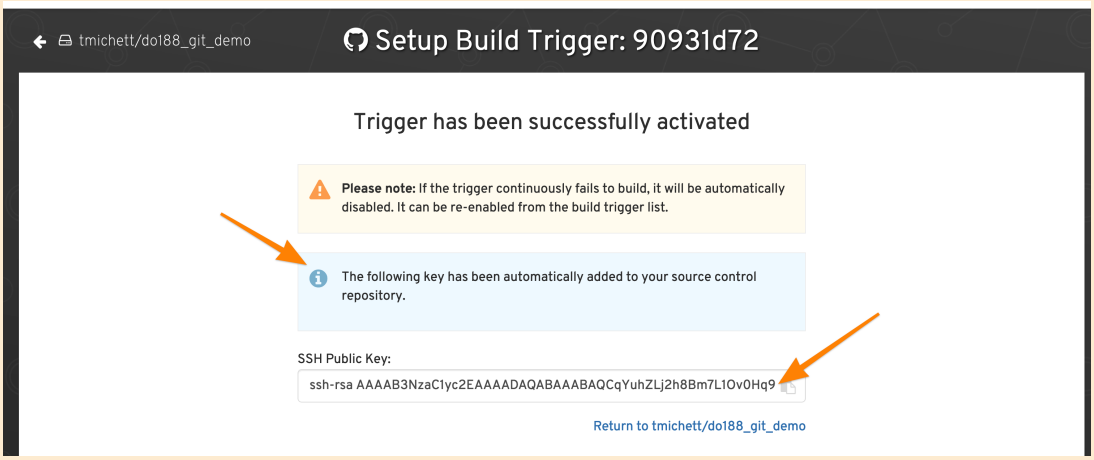

Figure 21. Trigger Activation Confirmation

Testing the Build Trigger

1. Return to the repository and look at the build triggers.

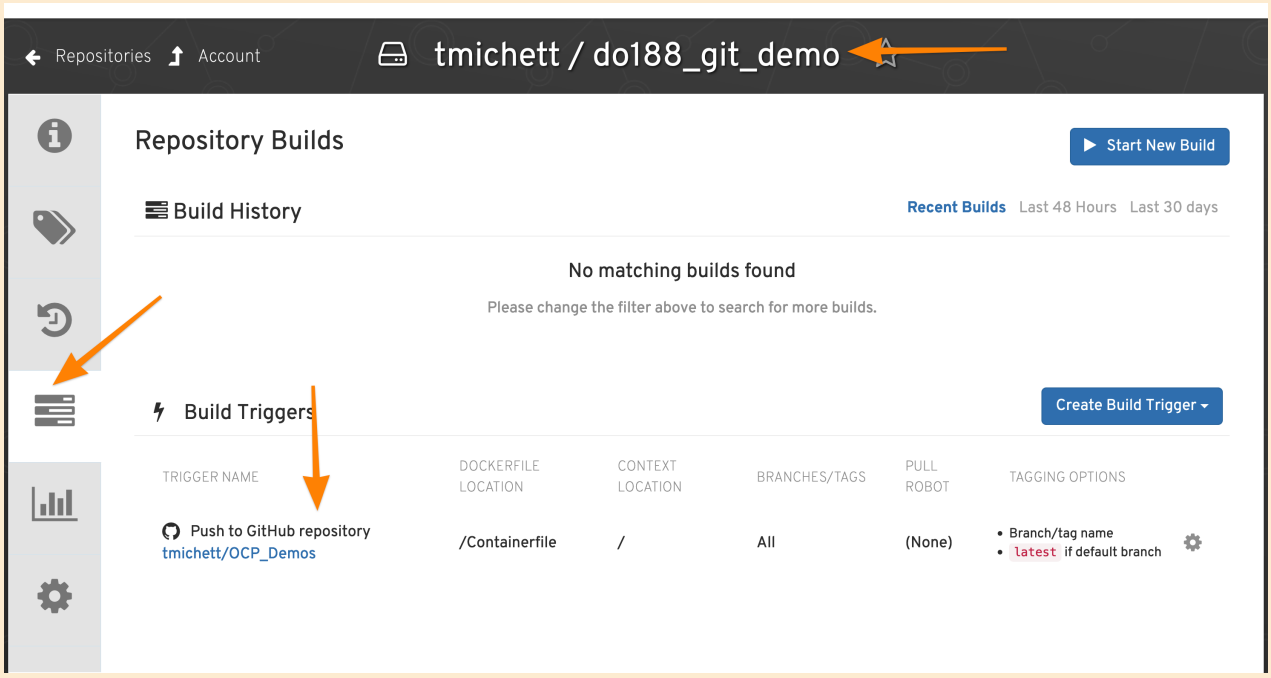

Figure 22. Repository Build Triggers

2. Click the **Gear** icon and select **Run Trigger Now**

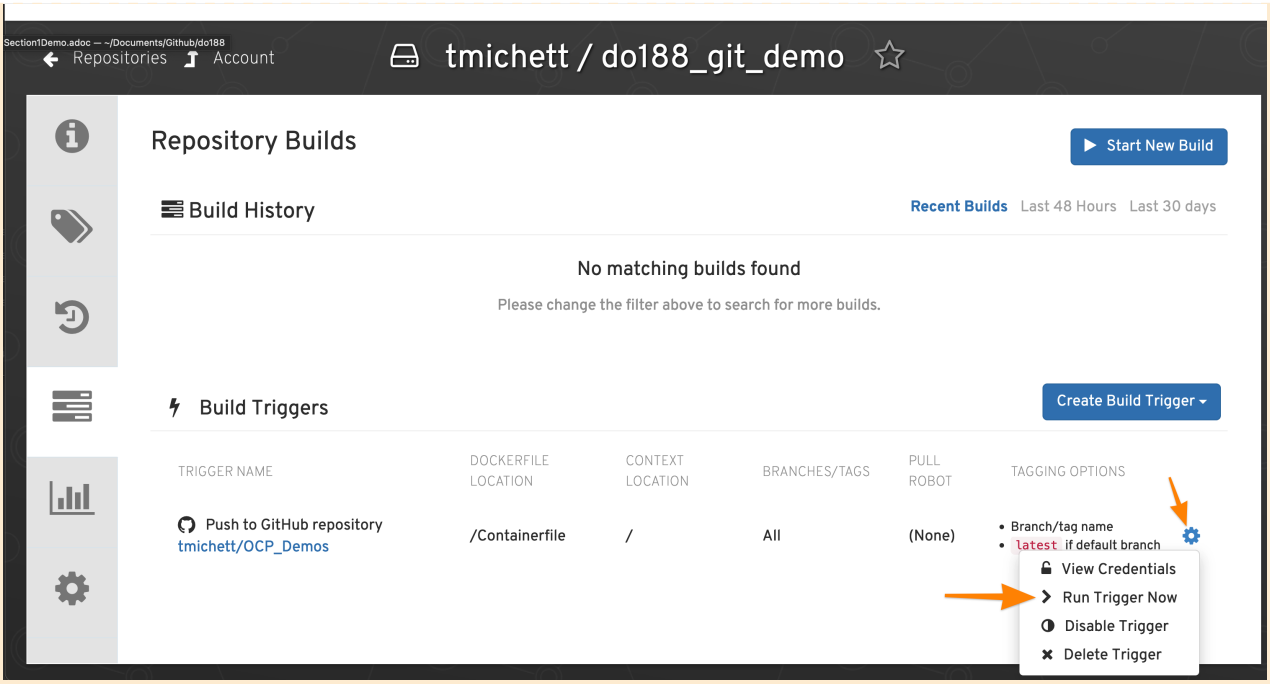

Figure 23. Testing Build Trigger

3. On the **Manually Start Build Trigger** leave everything default and click **Start Build**

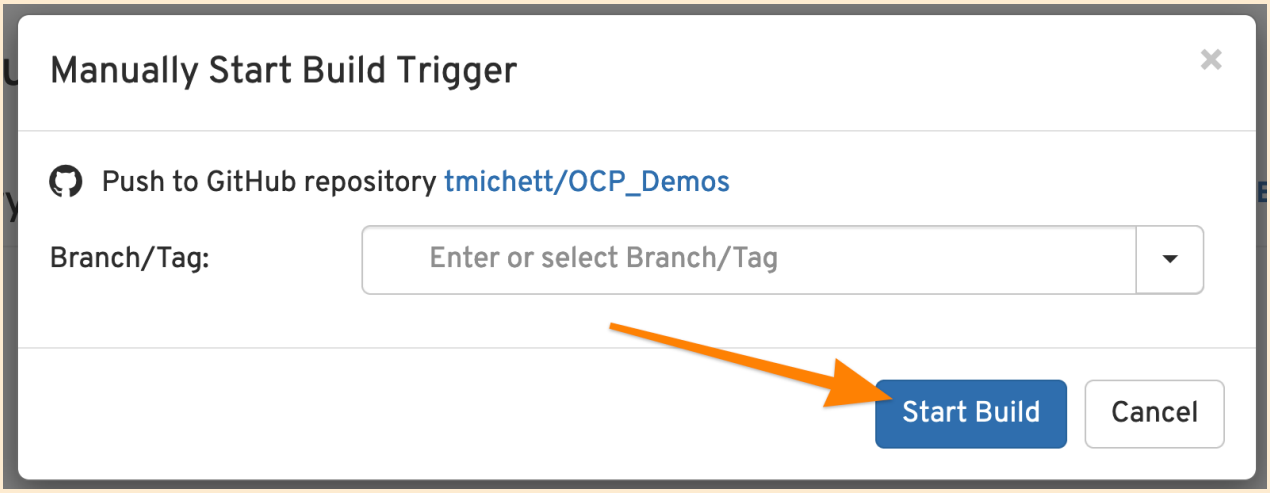

Figure 24. Manually Starting Build Trigger

Figure 25. Manual Build

4. Click the **Build ID** and watch the build. Then return to main repository

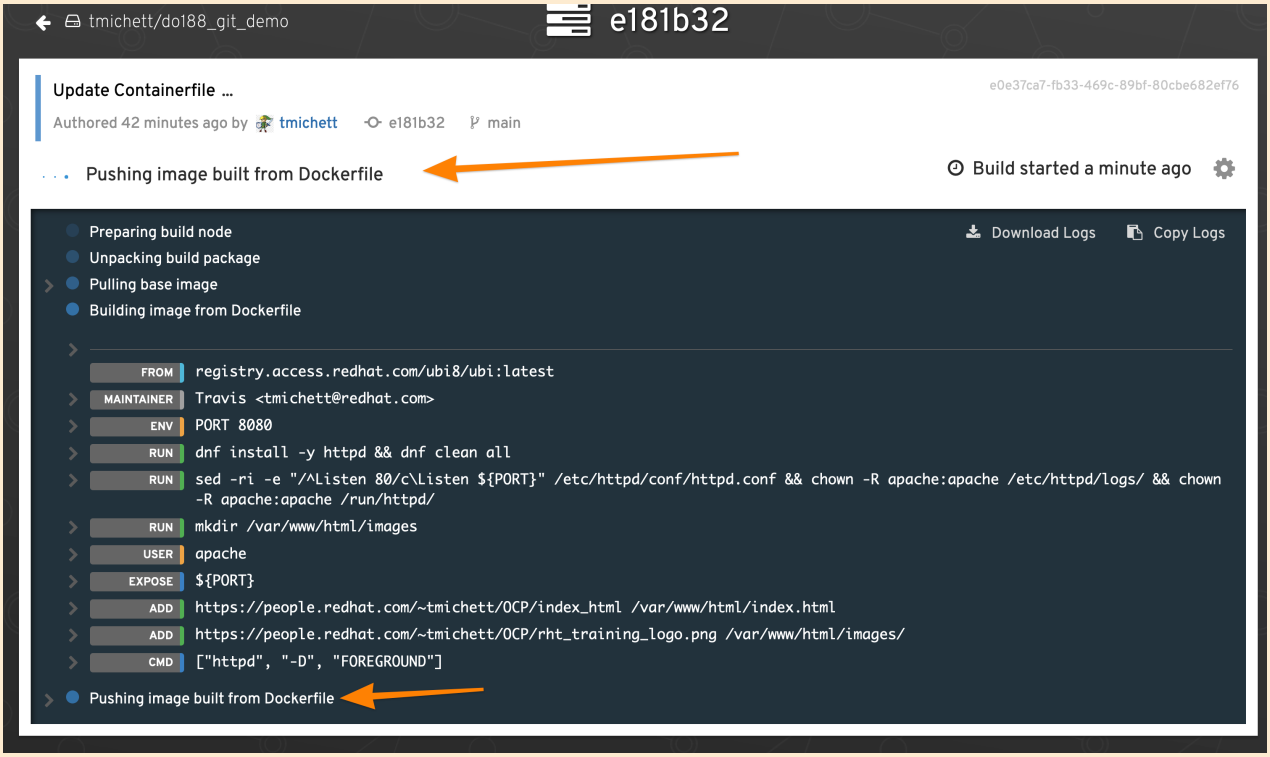

Figure 26. Build Log and Verification

Figure 27. Successful Build

5. Attempt to run a container from the new image.

```
[student@workstation ~]$ podman run --rm -d --name do188_custom_ws_demo -p
8080:8080 quay.io/tmichett/do188_git_demo
```

6. Test the website

```
[student@workstation ~]$ curl localhost:8080
This is a test and demo for the Containerfile build.


<img src="images/rht_training_logo.png" width="50%" height="50%">
```

3.3. Managing Images

Section Info Here

3.3.1. Image Management

3.3.1.1. Image Versioning and Tags

3.3.1.2. Pulling Images

3.3.1.3. Building Images

3.3.1.4. Pushing Images

3.3.1.5. Inspecting Images

3.3.1.6. Image Removal

3.4. DEMO: Managing Images

Example 7. DEMO - Tagging and Managing Container Images

1. Login and Pull a Container image.

```
[student@workstation do188]$ podman login registry.ocp4.example.com:8443 -u
developer -p developer
Login Succeeded!
```

```
[student@workstation do188]$ podman pull
registry.ocp4.example.com:8443/redhattraining/podman-python-server
Trying to pull registry.ocp4.example.com:8443/redhattraining/podman-python-
server:latest...
Getting image source signatures
Copying blob 25ad9a06c050 done

... OUTPUT OMITTED ...

28c663e6c7fd217b7f4d5c9d7d7f21ad8462ad1adc9051ee2bd5cfb80ee34042
```

2. View Images with **podman images**

```
[student@workstation do188]$ podman images
REPOSITORY                                                      TAG
IMAGE ID      CREATED       SIZE
quay.io/tmichett/do188_git_demo                                 latest
2c7ed9b2d91a  4 days ago    249 MB
registry.ocp4.example.com:8443/redhattraining/podman-python-server  latest
28c663e6c7fd  3 months ago  229 MB
```

3. Tag Images

```
[student@workstation do188]$ podman tag 2c7ed9b2d91a git_demo_demo

[student@workstation do188]$ podman tag
registry.ocp4.example.com:8443/redhattraining/podman-python-server
registry.ocp4.example.com:8443/redhattraining/podman-python-server:demo
```

4. View Image Tags

```
[student@workstation do188]$ podman images
REPOSITORY                                                      TAG
IMAGE ID      CREATED       SIZE
quay.io/tmichett/do188_git_demo                                 latest
2c7ed9b2d91a  4 days ago    249 MB
localhost/git_demo_demo                                         latest
2c7ed9b2d91a  4 days ago    249 MB
registry.ocp4.example.com:8443/redhattraining/podman-python-server  latest
28c663e6c7fd  3 months ago  229 MB
registry.ocp4.example.com:8443/redhattraining/podman-python-server  demo
28c663e6c7fd  3 months ago  229 MB
```

5. Delete image tag(s) and view images

```
[student@workstation do188]$ podman rmi quay.io/tmichett/do188_git_demo
Untagged: quay.io/tmichett/do188_git_demo:latest
```

```
[student@workstation do188]$ podman images
REPOSITORY                                                      TAG
IMAGE ID      CREATED       SIZE
localhost/git_demo_demo                                         latest
2c7ed9b2d91a  4 days ago    249 MB
registry.ocp4.example.com:8443/redhattraining/podman-python-server  latest
28c663e6c7fd  3 months ago  229 MB
registry.ocp4.example.com:8443/redhattraining/podman-python-server  demo
28c663e6c7fd  3 months ago  229 MB
```

6. Delete image completely

```
[student@workstation do188]$ podman rmi 28c663e6c7fd
Error: unable to delete image
"28c663e6c7fd217b7f4d5c9d7d7f21ad8462ad1adc9051ee2bd5cfb80ee34042" by ID with more
than one tag ([registry.ocp4.example.com:8443/redhattraining/podman-python-
server:latest registry.ocp4.example.com:8443/redhattraining/podman-python-
server:demo]): please force removal
```

Deleting Images with Multiple Tags

It is possible to remove an image with multiple tags, but you must use the **--force** or **-f** option.

Listing 19. Removing Images with Multiple Tags

```
[student@workstation do188]$ podman rmi 28c663e6c7fd -f
Untagged: registry.ocp4.example.com:8443/redhattraining/podman-
python-server:latest
Untagged: registry.ocp4.example.com:8443/redhattraining/podman-
python-server:demo
Deleted:
28c663e6c7fd217b7f4d5c9d7d7f21ad8462ad1adc9051ee2bd5cfb80ee34042
```

Listing 20. Images and tags fully removed

```
[student@workstation do188]$ podman images
REPOSITORY                TAG         IMAGE ID      CREATED     SIZE
localhost/git_demo_demo   latest      2c7ed9b2d91a  4 days ago  249 MB
```

Tags like Hard-Links to **inodes**

For some people, it is easy to think of images with multiple tags like having hard-links to **inodes**. A tag points to a unique image ID, so tags can be removed without removing the image. When the last tag is removed, the image is removed.

If you remove the image by the ID, it will remove the image and all image tags which is why it is required to use the **-f** or **--force** options. This is similar to deleting an **inode** on a Linux filesystem.

7. Inspecting Images

```
[student@workstation do188]$ podman inspect localhost/git_demo_demo
[
     {
          "Id":
"2c7ed9b2d91a13ee511e4d06e04ec602754f151f6b74218437b617a0c7cab1a6",
          "Digest":
"sha256:2a4ef2b67a120ee0a10cb2518f8f075b57aa27ff0d13445dbb6a489e2b240baa",

... OUTPUT OMITTED ...
```

The **podman inspect** *Command*

There are a few things to know about **podman inspect**.

- Images must be downloaded and exist locally on the system in order to be inspected. (*skopeo is used for remote images*).
- Output is extremely verbose, so Go templating can be used with the **--format** option and the curly braces.

8. View environment variables and ports exposed

```
[student@workstation ~]$ podman inspect localhost/git_demo_demo --format
="{{.Config.Env}}"
[PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin container=oci
PORT=8080]
```

Viewing and Accessing as JSON

It is also possible to use **jq** and process outputs of the commands using JSON filtering and access techniques.

```
 [student@workstation ~]$ podman inspect localhost/git_demo_demo |
jq '.[].Config.Env'
[

"PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
",
  "container=oci",
  "PORT=8080"
]
```

You can do all kinds of things with **jq** and **grep**.

Listing 21. The **jq** *command and* **grep** *Creatively*

```
[student@workstation ~]$ podman inspect localhost/git_demo_demo | jq '.[].Config.Env'
| jq -c '.[] | tostring' | grep PORT
"PORT=8080"
```

4. Custom Container Images

4.1. Create Images with Containerfiles

Section Info Here

4.1.1. Creating Images with Containerfiles

4.1.2. Choosing a Base Image

4.1.3. Containerfile Instructions

DNF vs MicroDNF

It is common, especially for container images, to use MicroDNF as DNF may be generally unavailable based on size and it requiring Python. Traditionally, the **minimal** images will have **microdnf** and DNF will not be available.

- https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_atomic_host/7/html/getting_started_with_containers/using_red_hat_universal_base_images_standard_minimal_and_runtimes
- https://fedoramagazine.org/building-smaller-container-images/

4.1.4. Container Image Tags

4.2. Build Images With Advance Containerfile Instructions

Section Info Here

4.2.1. Advanced Containerfile Instructions

4.2.2. The ENV Instruction

4.2.3. The VOLUME Instruction

4.2.4. The ENTRYPOINT and CMD Instructions

4.2.5. Multistage Builds

4.2.6. Examine Container Data Layers

4.2.6.1. Cache Image Layers

4.2.6.2. Reduce Image Layers

4.3. Rootless Podman

Section Info Here

4.3.1. Container Workload Isolation

4.3.2. Analyzing Rootless Containers

4.3.2.1. Changing the Container User

4.3.2.2. Explaining User Mapping

4.3.2.3. Limitations of Rootless Containers

5. Persisting Data

5.1. Volume Mounting

5.1.1. Copy-on-write File System

Image Layers

When building container images, each instruction modifying the container filesystem will create a new READ-ONLY data layer for the image. Image layers are stacked as a set of changes from previous layers.

Listing 22. Containerfile

```
FROM registry.access.redhat.com/ubi8/ubi-minimal ①

RUN microdnf install httpd ②
RUN microdnf clean all ③
RUN rm -rf /var/cache/yum ④

CMD httpd -DFOREGROUND ⑤
```

① Base layer of image

② Image layer where application is installed

③ Image layer where DNF cache is cleaned

④ Image layer where the **/var/cache/yum** us removed

⑤ Command launched with container based on this image

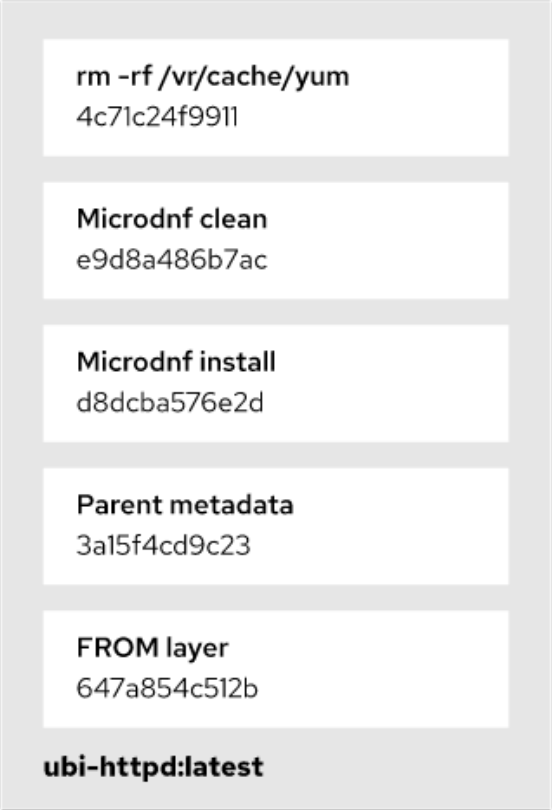

Figure 28. Container Image from Containerfile

Each Command Creates an Image Layer

Every command in a Dockerfile/Containerfile creates new layers to the container image (even if contents are removed and not available at runtime). In order to create optimized container images, it is often better practice to have a single instruction creating an image layer.

Listing 23. Optimized **Containerfile**

```
FROM registry.access.redhat.com/ubi8/ubi-minimal

RUN microdnf install httpd && \
    microdnf clean all && \
    rm -rf /var/cache/yum

CMD httpd -DFOREGROUND
```

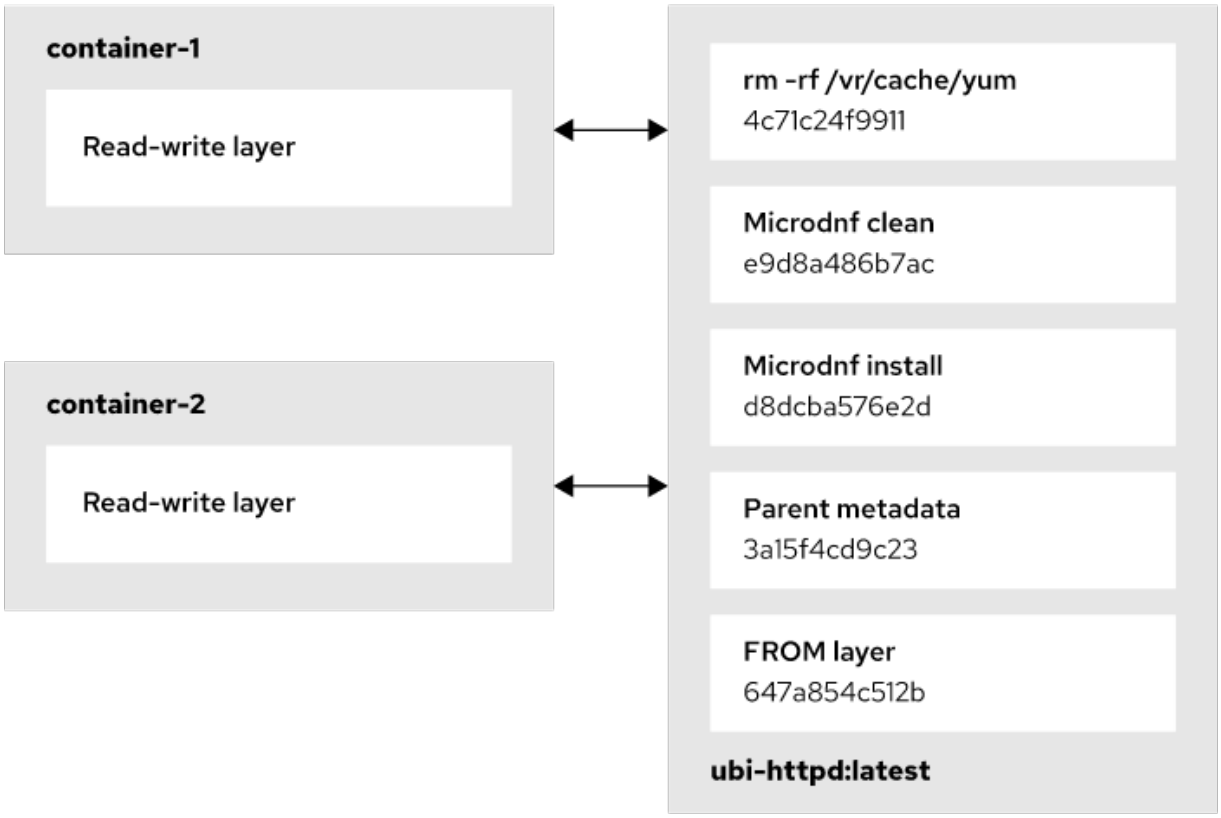

Figure 29. Running Containers - showing Container Image and the Read/Write Ephemeral Layer

Multiple Containers based on Single Container Image

You should recall that there can be multiple containers based on a single container image. Each running container will get its own READ/WRITE filesystem layer that is destroyed when the container is destroyed.

By default, the **/etc/containers/storage.conf** file controls where storage is used. For **root-based** containers, we use an Overlay filesystem. For **rootless** containers, FUSE-Overlay is used. This file also defines the temporary storage location for the READ/WRITE layers on the filesystem.

5.1.1.1. Implications of a COW File System

COW (Copy-on-Write) is the filesystem used for many VMs and containers. This allow the READ/WRITE layer using a union filesystem as shown in the images above.

Image Layers and Sharing

COW and container images are shared between different containers because the container image layers are immutable (read-only) layers

Performance Issues

Based on how union filesystems behave, it is possible to introduce performance issues for write-intensive processes. Because the COW layer for READ/WRITE will be a larger difference between that and the container image.

5.1.2. Store Data on Host Machine

In most instances it is desirable to store data for use in containers. This is called persistent storage of data and is accomplished using external mounts by using either ***volumes*** or ***bind mounts***.

Container Volumes

In previous courses, we focused on **bind mounts** like in the DO180 course. Using these mounts accomplished persistent data storage, but required using things like **:Z** when mounting or setting SELinux and other filesystem permissions to function properly. Volumes on the other hand are fully managed by Podman and are generally though as easier to use (especially when moving to an orchestration tool like Kubernetes and OCP which use volumes).

- **Persistence**
- **Use of Host File System**
- **Ease of Sharing**

Volumes vs. Bind Mounts

It should be noted that **Volumes** are managed by Podman, while **Bind Mounts** are mounts managed by the user.

The Podman command can mount **volumes** or **bind mounts** using the **-v** or **--volume** parameter.

Listing 24. Typical Mounting options for Volumes

```
--volume /path/on/host:/path/in/container:OPTIONS
```

Listing 25. Typical Mounting options for Mount

```
--mount type=TYPE,source=/path/on/host,destination=/path/in/container
```

The **mount** command is preferred way to mount directories in a container, but the **-v** parameter is the most widely used.

Using **podman mount**

The **--mount** option specifies and identifies the volume type

Volume Types

- **bind** - Used for Bind mounts
- **volume** - Used for Volume mounts
- **tmpfs** - Used for creating read-only (ephemeral mounts)

5.1.3. Storing Data with Bind Mounts

Bind mounts can mount existing directories on the host filesystem into the container. However, it is important to note that things like SELinux and permsissions must also be taken into account. The **:Z** can often be used and assist with this.

5.1.4. Storing Data with Volumes

Volumes allow Podman to manage data and mounts. The **podman volume** command is used to create volumes and leverage volumes.

Importing and Exporting Data in Volumes

The **podman volume** command supports both **import** and **export** arguments which is capable of importing or exporting data into a Podman volume.

By default, the volumes are stored on the host system

- **rootless**: $HOME/.local/share/containers/storage/volumes/

5.1.5. Storing Data with a tmpfs Mount

5.2. Working with Databases

Section Info Here

5.2.1. Stateful Database Containers

5.2.2. Good Practices for Database Containers

5.2.3. Importing Database Data

5.2.3.1. Database Containers with Data-loading Features

5.2.3.2. Data Loading with a Database Client

5.2.4. Red Hat Database Containers

6. Troubleshooting Containers

6.1. Container Logging and Troubleshooting

Section Info Here

6.2. Remote Debugging Containers

Section Info Here

7. Multi-Container Applications with Compose

7.1. Compose Overview and Use Cases

Section Info Here

7.1.1. Orchestrate Containers with Podman Compose

7.1.2. Podman Pods

7.1.3. The Compose File

7.1.3.1. Start and Stop Containers with Podman Compose

7.1.4. Networking

7.1.5. Volumes

7.2. Build Developer Environments with Compose

7.2.1. Compose Overview

7.2.2. Podman Compose and Podman

7.2.3. Multi-container Developer Environments with Compose

8. Container Orchestration with OpenShift and Kubernetes

8.1. Deploy Applications in OpenShift

Section Info Here

8.2. Multi-pod Applications

Section Info Here

Appendix A: EX188 Exam Objectives

The **EX188** exam objectives are listed here: https://www.redhat.com/en/services/training/ex188-red-hat-certified-specialist-containers-exam?section=Objectives. For convenience, a snapshot of the objectives have been placed in this guide.

- Implement images using Podman
  - Understand and use FROM (the concept of a base image) instruction.
  - Understand and use RUN instruction.
  - Understand and use ADD instruction.
  - Understand and use COPY instruction.
  - Understand the difference between ADD and COPY instructions.
  - Understand and use WORKDIR and USER instructions.
  - Understand security-related topics.
  - Understand the differences and applicability of CMD vs. ENTRYPOINT instructions.
  - Understand ENTRYPOINT instruction with param.
  - Understand when and how to expose ports from a Containerfile.
  - Understand and use environment variables inside images.
  - Understand ENV instruction.
  - Understand container volume.
  - Mount a host directory as a data volume.
  - Understand security and permissions requirements related to this approach.
  - Understand the lifecycle and cleanup requirements of this approach.
- Manage Images
  - Understand private registry security.
  - Interact with many different registries.
  - Understand and use image tags
  - Push and pull images from and to registries.
  - Back up an image with its layers and meta data vs. backup a container state.
- Run containers locally using Podman
  - Run containers locally using Podman
  - Get container logs.
  - Listen to container events on the container host.
  - Use Podman inspect.

- Specifying environment parameters.
- Expose public applications.
- Get application logs.
- Inspect running applications.

- Run multi-container applications with Podman
  - Create application stacks
  - Understand container dependencies
  - Working with environment variables
  - Working with secrets
  - Working with volumes
  - Working with configuration
- Troubleshoot containerized applications
  - Understand the description of application resources
  - Get application logs
  - Inspect running applications
  - Connecting to running containers