



Red Hat Training and Certification

DO188 - Introduction to Containers with
Podman

Travis Michette

Version 1.0

Table of Contents

Introduction	1
Demo Setup/Preparing to Teach	1
1. Introduction and Overview of Containers	3
1.1. Introduction to Containers	3
1.1.1. Describing Containers	3
1.1.2. Comparing Containers to Virtual Machines	6
1.1.3. Development for Containers	7
1.2. Introduction to Kubernetes and OpenShift	8
1.2.1. Kubernetes Overview	8
1.2.2. Red Hat OpenShift Container Platform Overview	9
2. Podman Basics	11
2.1. Creating Containers with Podman	11
2.1.1. Introducing Podman	11
2.1.2. Working with Podman	11
2.1.2.1. Pulling and Displaying Images	11
2.1.2.2. Running and Displaying Containers	11
2.1.2.3. Exposing Containers	11
2.1.2.4. Using Environment Variables	11
2.2. DEMO: Creating Containers with Podman	11
2.3. Container Networking Basics	14
2.3.1. Container Networking Basics	14
2.3.2. Managing Podman Networks	14
2.3.3. Enabling Domain Name Resolution	14
2.3.4. Connecting Containers	15
2.4. DEMO: Container Networking Basics	15
2.5. Accessing Containerized Network Services	15
2.5.1. Port Forwarding	15
2.5.1.1. List Port Mappings	15
2.5.2. Networking in Containers	15
2.6. DEMO: Accessing Containerized Network Services	16
2.7. Accessing Containers	18
2.7.1. Container Transparency	18
2.7.2. Start Processes in Containers	18
2.7.3. Open an Interactive Session in Containers	19
2.7.4. Copy Files in and Out of Containers	19
2.8. DEMO: Accessing Containers	19

2.9. Managing the Container Lifecycle	21
2.9.1. Container Lifecycle	21
2.9.2. Inspect a Container	21
2.9.3. Stop a Container	21
2.9.3.1. Stop a Container Gracefully	21
2.9.3.2. Stop a Container Forcefully	21
2.9.3.3. Pause a Container	21
2.9.4. Restarting a Container	21
2.9.5. Remove a Container	21
2.10. DEMO: Accessing Containers	21
3. Container Images	24
3.1. Container Image Registries	24
3.1.1. Container Registries	24
3.1.2. The Containerfile	24
3.1.3. Red Hat Registry	24
3.1.3.1. Useful Container Images	24
3.1.4. Quay.io	24
3.1.5. Manage Registries with Podman	24
3.1.6. Manage Registry Credentials with Podman	24
3.2. DEMO: Container Image Registries	24
3.3. Managing Images	32
3.3.1. Image Management	32
3.3.1.1. Image Versioning and Tags	32
3.3.1.2. Pulling Images	32
3.3.1.3. Building Images	32
3.3.1.4. Pushing Images	32
3.3.1.5. Inspecting Images	32
3.3.1.6. Image Removal	32
3.4. DEMO: Managing Images	32
4. Custom Container Images	37
4.1. Create Images with Containerfiles	37
4.1.1. Creating Images with Containerfiles	37
4.1.2. Choosing a Base Image	37
4.1.3. Containerfile Instructions	37
4.1.4. Container Image Tags	37
4.2. Build Images With Advance Containerfile Instructions	37
4.2.1. Advanced Containerfile Instructions	37
4.2.2. The ENV Instruction	37

4.2.3. The VOLUME Instruction	37
4.2.4. The ENTRYPOINT and CMD Instructions	37
4.2.5. Multistage Builds	37
4.2.6. Examine Container Data Layers	37
4.2.6.1. Cache Image Layers	37
4.2.6.2. Reduce Image Layers	37
4.3. Rootless Podman	37
4.3.1. Container Workload Isolation	37
4.3.2. Analyzing Rootless Containers	38
4.3.2.1. Changing the Container User	38
4.3.2.2. Explaining User Mapping	38
4.3.2.3. Limitations of Rootless Containers	38
5. Persisting Data	39
5.1. Volume Mounting	39
5.1.1. Copy-on-write File System	39
5.1.1.1. Implications of a COW File System	39
5.1.2. Store Data on Host Machine	39
5.1.3. Storing Data with Bind Mounts	39
5.1.3.1. Troubleshoot Bind Mounts	39
5.1.4. Storing Data with Volumes	39
5.1.5. Storing Data with a tmpfs Mount	39
5.2. Working with Databases	39
5.2.1. Stateful Database Containers	39
5.2.2. Good Practices for Database Containers	39
5.2.3. Importing Database Data	39
5.2.3.1. Database Containers with Data-loading Features	39
5.2.3.2. Data Loading with a Database Client	39
5.2.4. Red Hat Database Containers	39
6. Troubleshooting Containers	40
6.1. Container Logging and Troubleshooting	40
6.2. Remote Debugging Containers	40
7. Multi-Container Applications with Compose	41
7.1. Compose Overview and Use Cases	41
7.1.1. Orchestrate Containers with Podman Compose	41
7.1.2. Podman Pods	41
7.1.3. The Compose File	41
7.1.3.1. Start and Stop Containers with Podman Compose	41
7.1.4. Networking	41

7.1.5. Volumes	41
7.2. Build Developer Environments with Compose	41
7.2.1. Compose Overview	41
7.2.2. Podman Compose and Podman	41
7.2.3. Multi-container Developer Environments with Compose	41
8. Container Orchestration with OpenShift and Kubernetes	42
8.1. Deploy Applications in OpenShift	42
8.2. Multi-pod Applications	42
Appendix A: EX188 Exam Objectives	43

Introduction

§ == Repositories for this Course

Main Repository

The DO467 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 DO4188 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 D0188_Demo/Demos
```

QUAY Registry Password

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



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

OCP Credentials and Passwords

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.

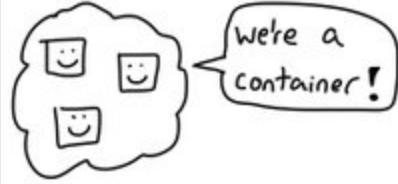
Containers use the following Linux components:

- **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.

JULIA EVANS
@b0rk

containers = processes

A container is a group of Linux processes



a container process can have 2 PIDs
(or more!)



I started 'top' in a Docker container.
Here's what that looks like in ps:

outside the container

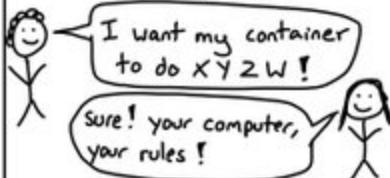
```
$ ps aux | grep top
USER      PID START   COMMAND
root    23540 20:55   top
bork   23546 20:57   top
```

inside the container

```
$ ps aux | grep top
USER      PID START   COMMAND
root      25 20:55   top
```

these two are the same process!

container processes can do anything a normal process can



but you can set rules about what they can do

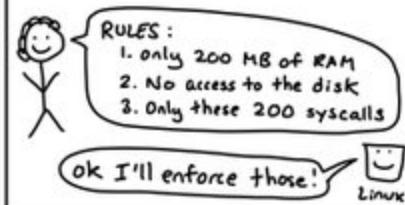


Figure 1. Containers are Processes - image from Julia Evans (@b0rk)

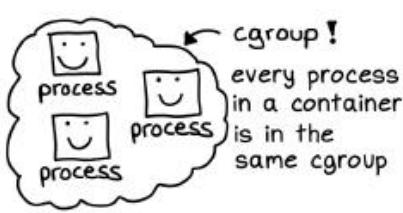
cgroups

13

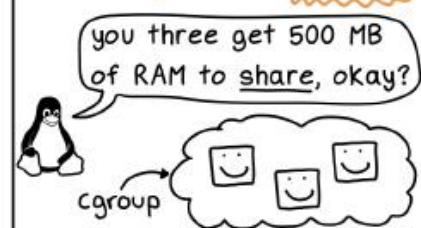
processes can use a lot of memory



a cgroup is a group of processes

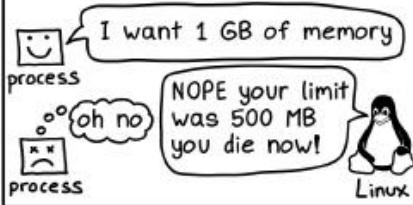


cgroups have memory/CPU {limits}

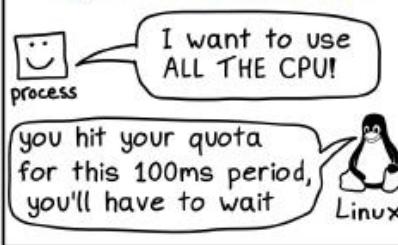


use too much memory: get OOM killed

"out of memory"



use too much CPU: get slowed down



cgroups track memory & CPU usage

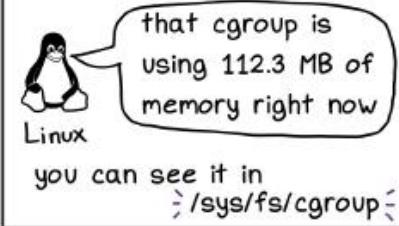


Figure 2. CGroup Overview - image from Julia Evans (@b0rk)



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

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

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.

overlay filesystems

11

how layers work:
`mount -t overlay`

can you combine these 37 layers into one filesystem?

yes! just run
`mount -t overlay`
with the right parameters!

mount -t overlay has 4 parameters

lowerdir: list of read-only directories
upperdir: directory where writes should go
workdir: empty directory for internal use
target: the merged result

upperdir: where all writes go

when you create, change, or delete a file, it's recorded in the upperdir.

usually this starts out empty and is deleted when the container exits

lowerdir: the layers. read only.

you can run
`$ mount -t overlay`
inside a container to see all the lowerdirs that were combined to create its filesystem!

here's an example!

```
$ mount -t overlay overlay -o
    lowerdir=/lower,upperdir=/upper,workdir=/work /merged
$ ls /upper
cat.txt dog.txt
$ ls /lower
dog.txt bird.txt
$ ls /merged
cat.txt dog.txt
                the merged version of dog.txt is
                the one from the upper directory
                                bird.txt
```

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

1.1.2. Comparing Containers to Virtual Machines

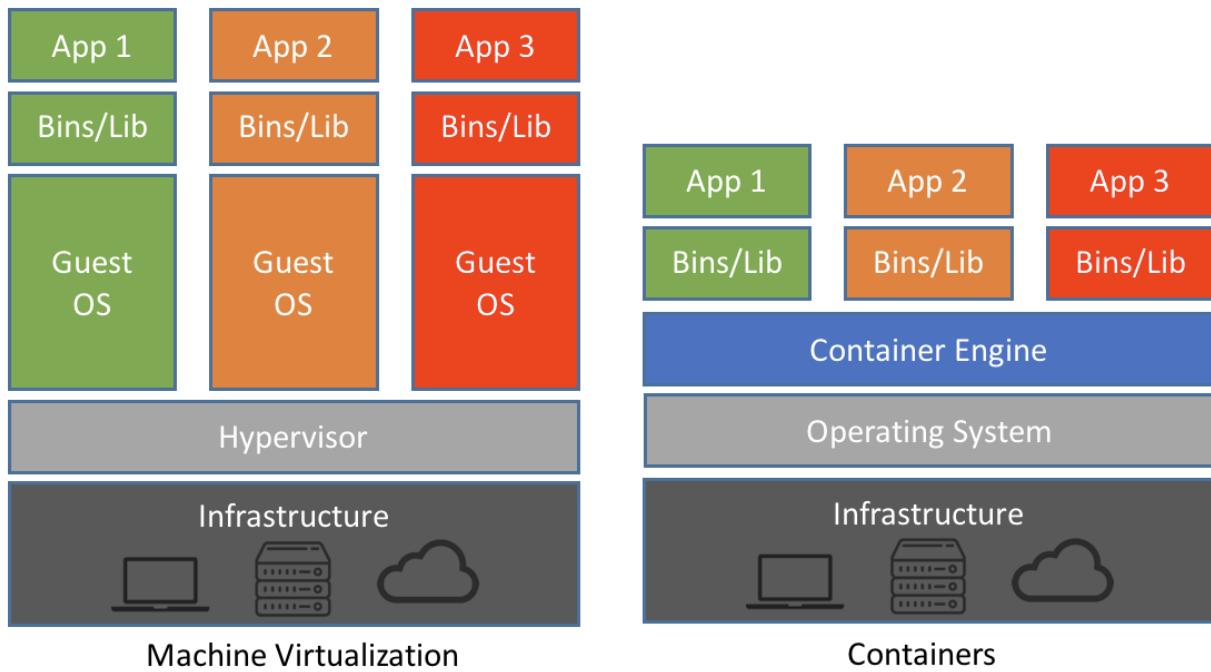


Figure 5. VM Container Comparison

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.

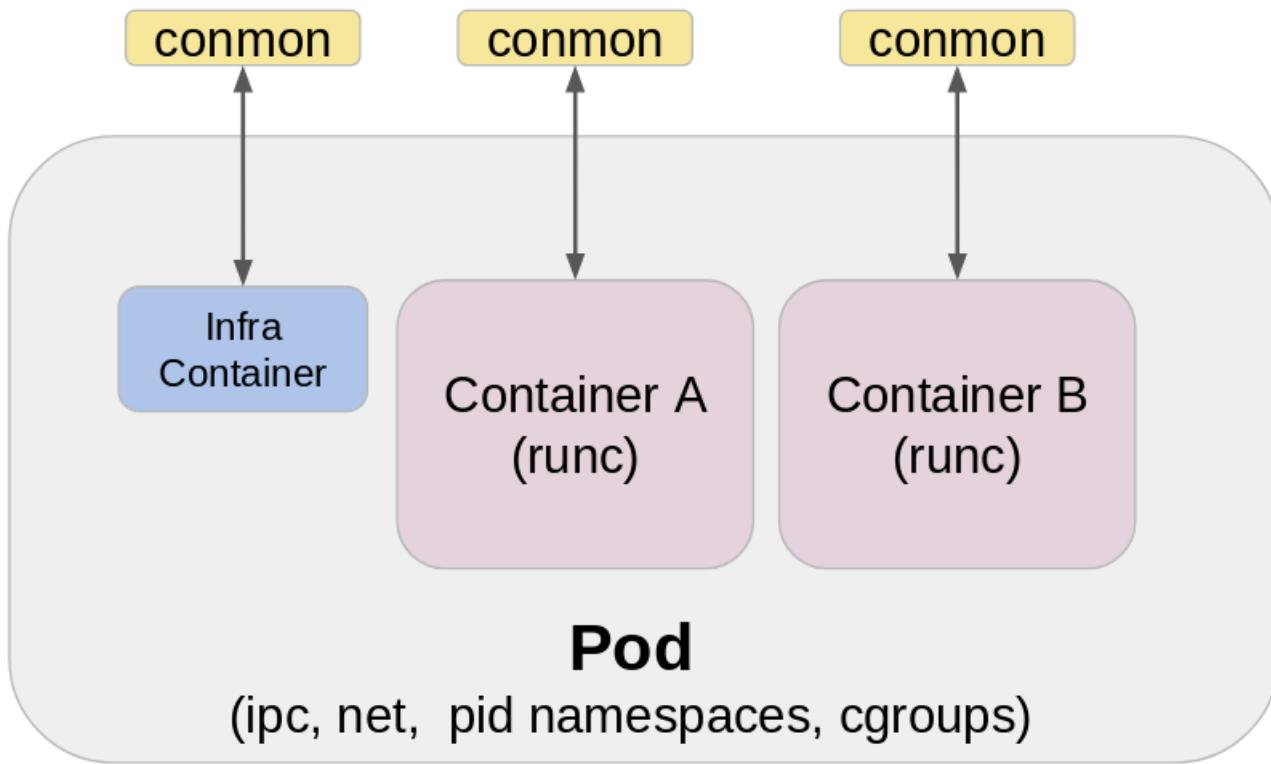


Figure 6. **podman** Pod Architecture

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 conceptually 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**

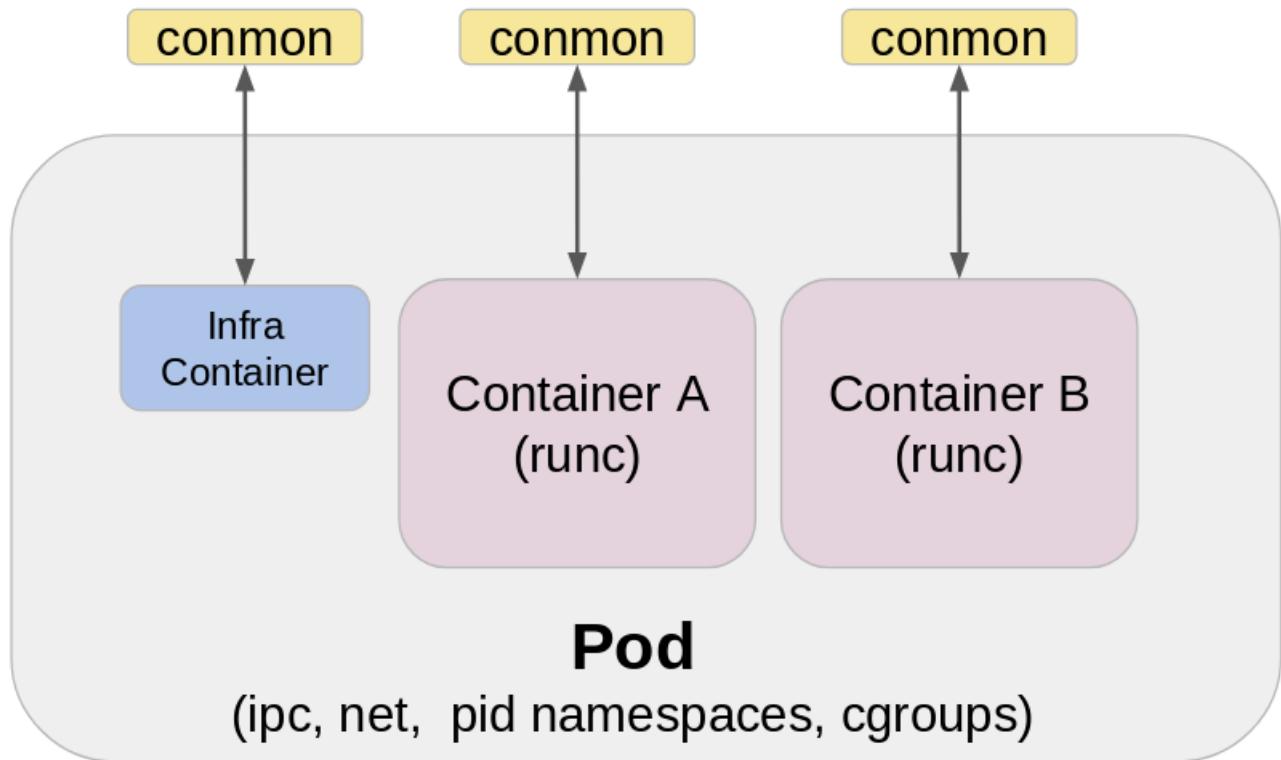


Figure 7. Podman pod Architecture

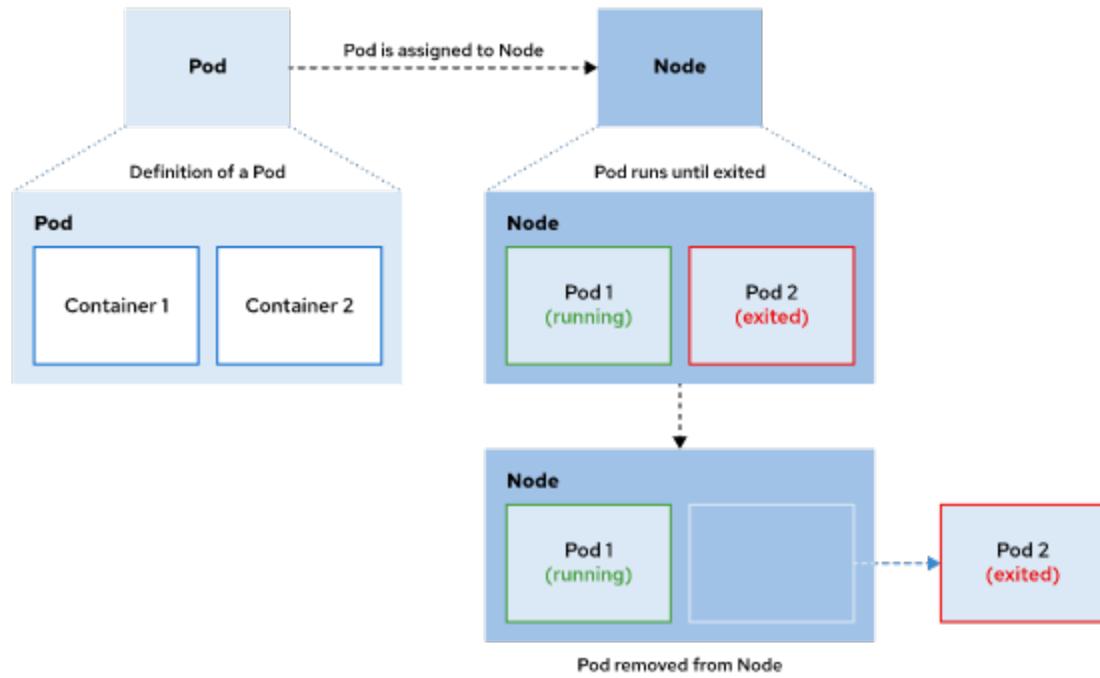


Figure 8. OCP Application and Container Lifecycle

2. Podman Basics

2.1. Creating Containers with Podman

Section Info Here

2.1.1. Introducing Podman

2.1.2. Working with Podman

2.1.2.1. Pulling and Displaying Images

2.1.2.2. Running and Displaying Containers

2.1.2.3. Exposing Containers

2.1.2.4. Using Environment Variables

2.2. DEMO: Creating Containers with Podman

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 1. 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 2. 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 3. Running Containers

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

Listing 4. 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 5. 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

Section Info Here

2.3.1. Container Networking Basics

2.3.2. Managing Podman Networks

2.3.3. Enabling Domain Name Resolution

2.3.4. Connecting Containers

2.4. DEMO: Container Networking Basics

Example 2. DEMO - Using Podman Networks

1. Explore **podman** to determine 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 D0180 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 D0180 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  
5c422becbd7d4f8e349c2300d98811421d73f4b7b05f3023cdfcccd5631c835d4
```

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 D0180 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 6. 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" container

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

Listing 7. Obtaining IP Address

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

7. Verify access from workstation machine

```
[student@workstation ~]$ curl 10.88.0.3  
I am custom material for the D0180 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 8. Cleanup Root Containers

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

Listing 9. 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 D0180 course
```

3. Modify content for website

Listing 10. View content

```
[student@workstation ~]$ podman exec demo-web cat /var/www/html/index.html  
I am custom material for the D0180 course
```

Listing 11. 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 D0188" > /var/www/html/index.html  
  
bash-4.4# curl localhost  
This is a demo for D0188
```

4. View modified website content from workstation.

```
[student@workstation ~]$ curl localhost:2080  
This is a demo for D0188
```

5. Copy new website into container

```
[student@workstation ~]$ podman cp ~/Github/OCP_Demos/D0188_Web/index.html demo-web:/var/www/html/  
[student@workstation ~]$ podman cp ~/Github/OCP_Demos/D0188_Web/penguin3.jpeg demo-web:/var/www/html/
```

6. View website

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

Instructor Demo - Open in Firefox (Remotely)

Listing 12. 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 ~]$ 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

The screenshot shows the Quay.io interface. At the top, there's a navigation bar with the Red Hat logo, 'REDACTED Quay.io', 'EXPLORE', 'REPOSITORIES' (which is the active tab), and 'TUTORIAL'. To the right of the navigation are a search bar, a '+' button, a notification bell with a red badge, and a user profile for 'tmichett'. The main area is titled 'Repositories' and displays a list of 216 repositories. Each repository entry includes the name, last modified date, activity level (indicated by a bar chart icon), and a star icon. On the right side of the list, there's a sidebar titled 'Users and Organizations' which lists 'tmichett' and 'redhattraining', along with a '+ Create New Organization' button.

Figure 9. Quay Repositories

2. Click "+ Create New Repository" and select "New Repository"

- Repository Name: **do188_git_demo**
- Visibility: **public**

The screenshot shows the 'Create New Repository' form. At the top, it has a back arrow labeled 'Repositories' and the title 'Create New Repository'. Below the title, there's a breadcrumb navigation showing 'tmichett / do188_git_demo' with an orange arrow pointing to the repository name. The form has fields for 'Repository Description' (with a placeholder 'Click to set repository description') and 'Repository Visibility' (with a 'Public' option selected). A note below the visibility field says 'Anyone can see and pull from this repository. You choose who can push.' There's also a 'Private' option with the note 'You choose who can see, pull and push from/to this repository.'.

Figure 10. Repository Name and Visibility

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

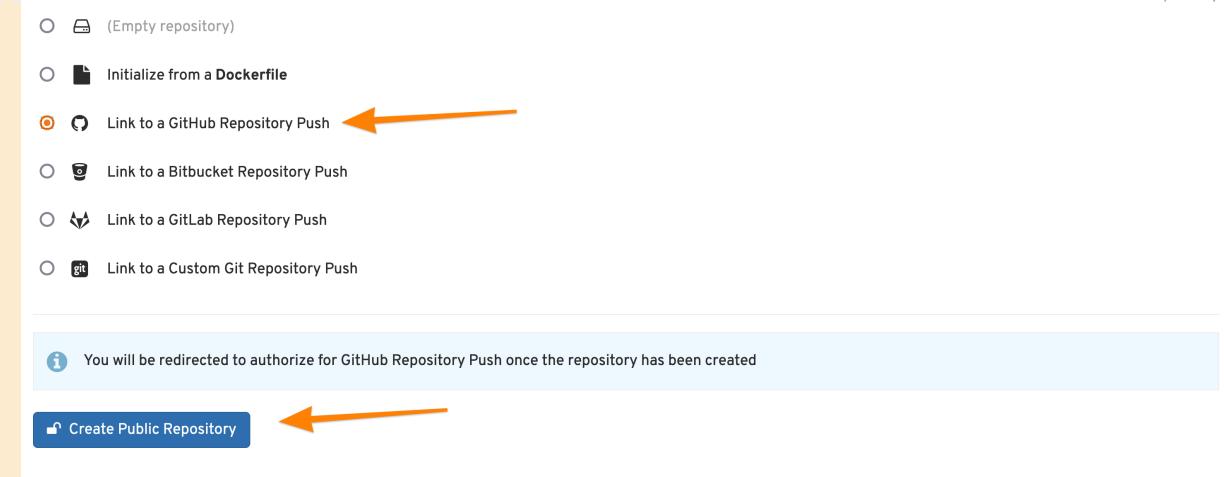


Figure 11. Linking to Version Control Dockerfile/Containerfile

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

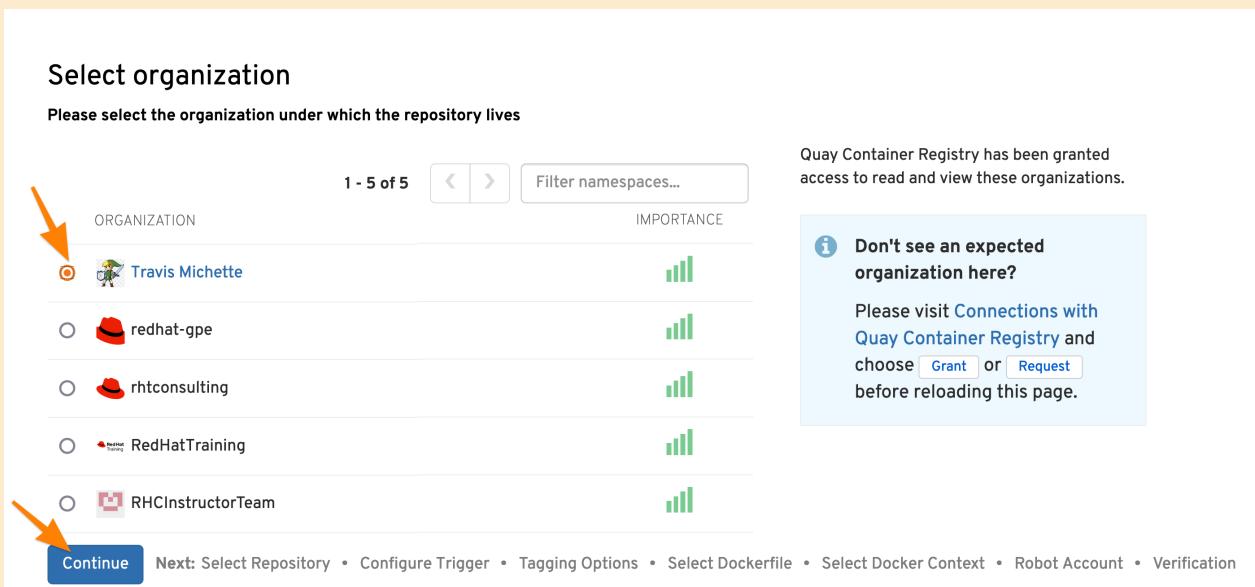


Figure 12. Github Organization Selection

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

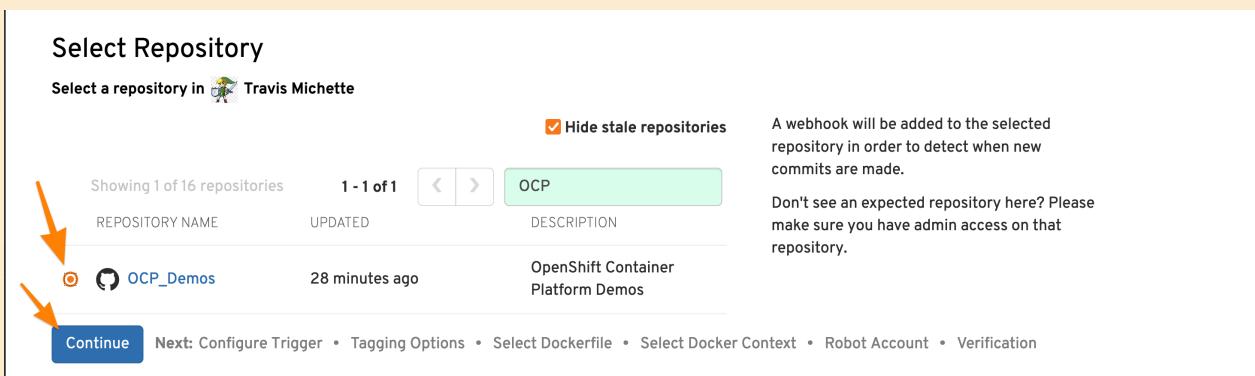


Figure 13. Github Repository Selection

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

Configure Trigger

Configure trigger options for  Travis Michette/OCP_Demos

Trigger for all branches and tags (default)

Build a container image for each commit across all branches and tags

Do you want to build a new container image for commits across all branches and tags, or limit to a subset?

Trigger only on branches and tags matching a regular expression

Only build container images for a subset of branches and/or tags.

For example, if you use release branches instead of `master` for building versions of your software, you can configure the trigger to only build images for these branches.

All images built will be tagged with the name of the branch or tag whose change invoked the trigger

Continue

Next: Tagging Options • Select Dockerfile • Select Docker Context • Robot Account • Verification

Figure 14. Github Branch and Tag Trigger Option Selection

6. Select the tagging options and click Continue

Configure Tagging

Red Hat E-Business Suite
<https://rsaebs.corp.redhat.com/>

Confirm basic tagging options

Tag manifest with the branch or tag name

Tags the built manifest the name of the branch or tag for the git commit.



By default, all built manifests will be tagged with the name of the branch or tag in which the commit occurred.

Add `latest` tag if on default branch

Tags the built manifest with `latest` if the build occurred on the default branch for the repository.

To modify this default, as well as the default to add the `latest` tag, change the corresponding options on the left.

Add custom tagging templates

No tag templates defined.

Enter a tag template:

`$(commit_info.short_sha)`

Add Tag Template

Need more control over how the built manifest is tagged? Add one or more custom tag templates.

For example, if you want all built manifests to be tagged with the commit's short SHA, add a template of `$(commit_info.short_sha)`.

As another example, if you want on those manifests committed to a `branch` to be tagged with the branch name, you can add a template of `$(parsed_ref.branch)`.

A full reference of for these templates can be found in the [Tag template documentation](#).

Continue

Next: Select Dockerfile • Select Docker Context • Robot Account • Verification

Figure 15. Github Container Tagging Option Selection

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

Select Dockerfile

Please select the location of the Dockerfile to build when this trigger is invoked

Continue

Next: Select Docker Context • Robot Account • Verification

Please select the location containing the Dockerfile to be built.

The Dockerfile path starts with the context and ends with the path to the Dockerfile that you would like to build

If the Dockerfile is located at the root of the git repository and named Dockerfile, enter `/Dockerfile` as the Dockerfile path.

Figure 16. Github Dockerfile/Containerfile Selection

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

Select Context

Please select the context for the Docker build

Continue

Next: Robot Account • Verification

Please select a Docker context.

The build context directory is the path of the directory containing the Dockerfile and any other files to be made available when the build is triggered.

If the Dockerfile is located at the root of the git repository, enter `/` as the build context directory.

Figure 17. Container Context Build Selection

9. Skip the "Robot Account" and click **Continue**

10. Verify the build trigger and click **Continue**

Ready to go!

Click "Continue" to complete setup of this build trigger.

Continue

Figure 18. 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.

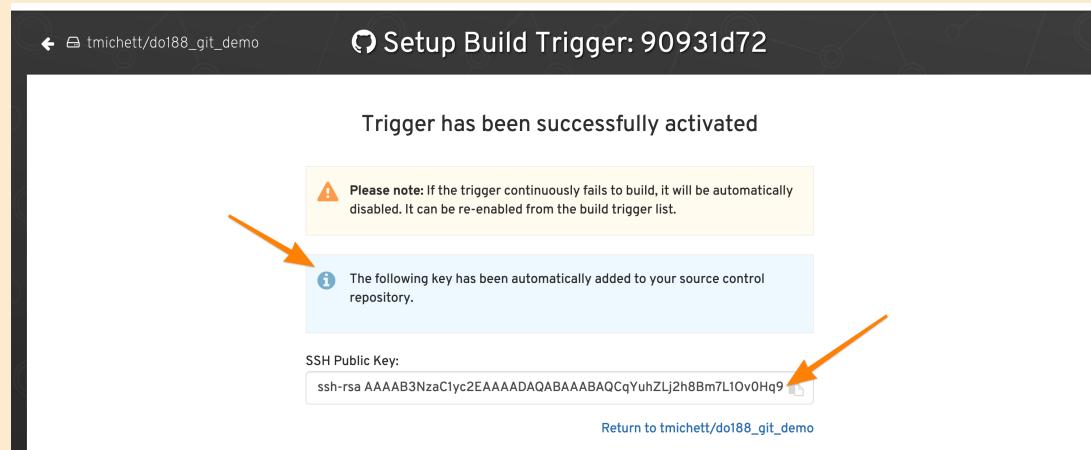


Figure 19. Trigger Activation Confirmation

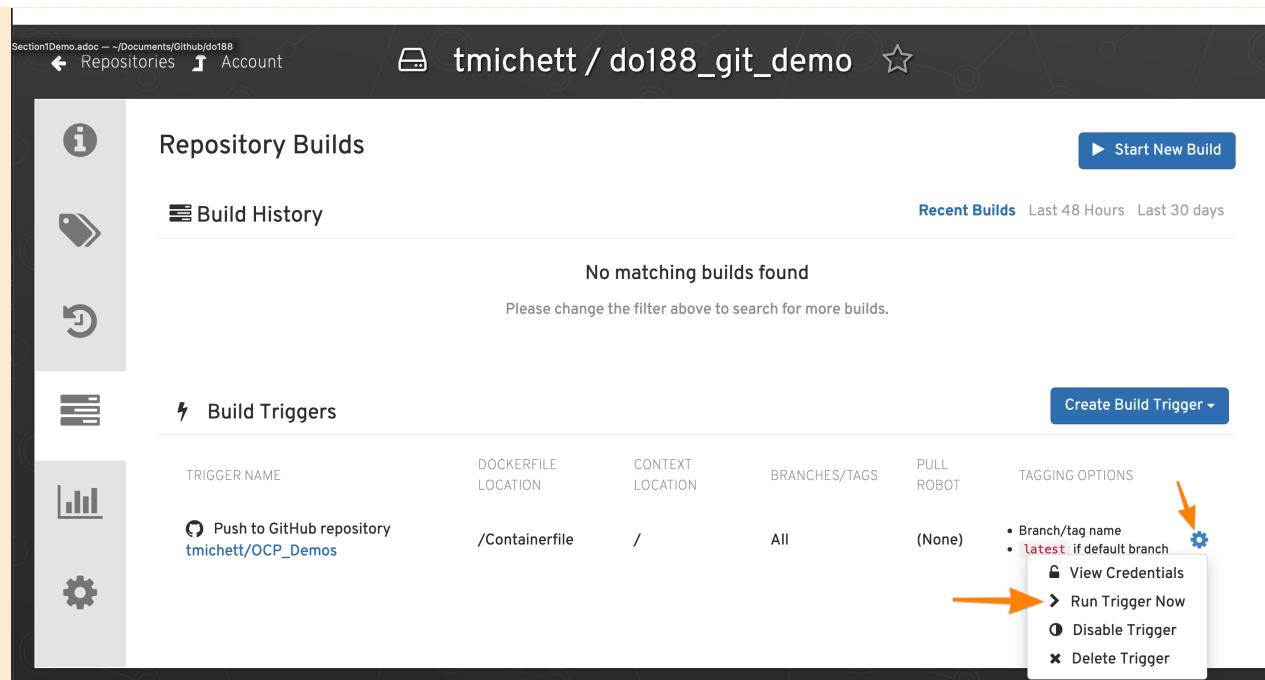
Testing the Build Trigger

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

The screenshot shows the "tmichett / do188_git_demo" repository page. On the left sidebar, there are icons for Repository Builds, Build History, Build Triggers (which is highlighted with an orange arrow), and other repository management options. The main content area is titled "Repository Builds" and shows a "Build Triggers" section. It lists one trigger: "Push to GitHub repository tmichett/OCP_Demos". The trigger details are: TRIGGER NAME "Push to GitHub repository tmichett/OCP_Demos", DOCKERFILE LOCATION "/Containerfile", CONTEXT LOCATION "/", BRANCHES/TAGS "All", PULL ROBOT "(None)", and TAGGING OPTIONS with options "Branch/tag name" and "latest if default branch". A "Create Build Trigger" button is visible at the top right of the triggers table. A blue arrow points to the "Start New Build" button at the top right of the page.

Figure 20. Repository Build Triggers

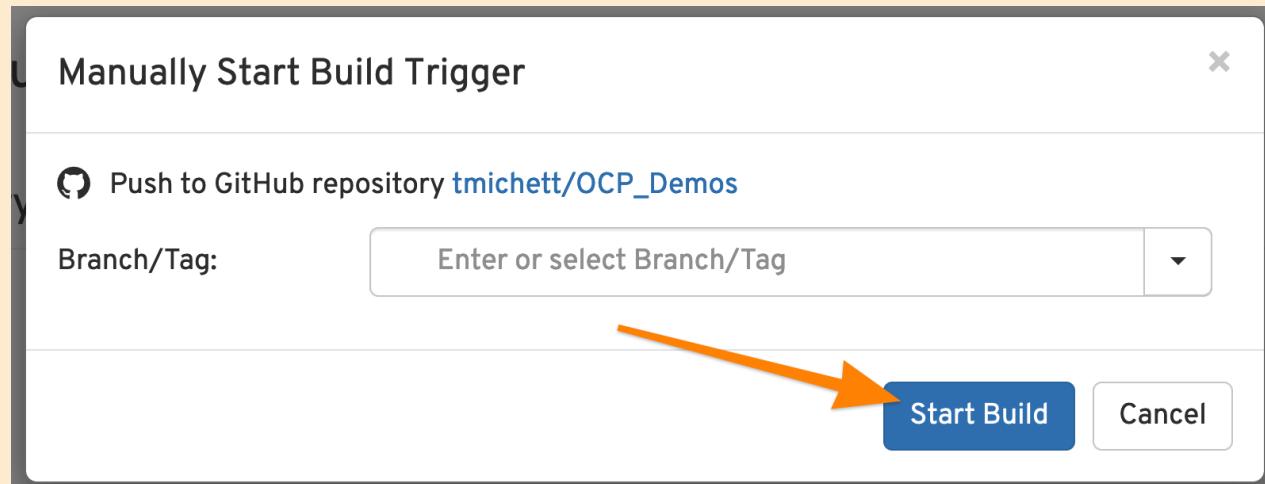
2. Click the Gear icon and select Run Trigger Now



The screenshot shows the GitHub Actions interface for the repository `tmichett / do188_git_demo`. On the left sidebar, there are icons for Repository Builds, Build History, and Build Triggers. The Build Triggers section lists a trigger named "Push to GitHub repository `tmichett/OCP_Demos`". To the right of the trigger, there are filters for Dockerfile Location (set to "/Containerfile"), Context Location (set to "/"), Branches/Tags (set to All), and Pull Robot (set to (None)). Under Tagging Options, there are two items: "Branch/tag name" and "latest if default branch". A gear icon is highlighted with an orange arrow, opening a context menu with the following options: "View Credentials", "Run Trigger Now" (which is also highlighted with an orange arrow), "Disable Trigger", and "Delete Trigger".

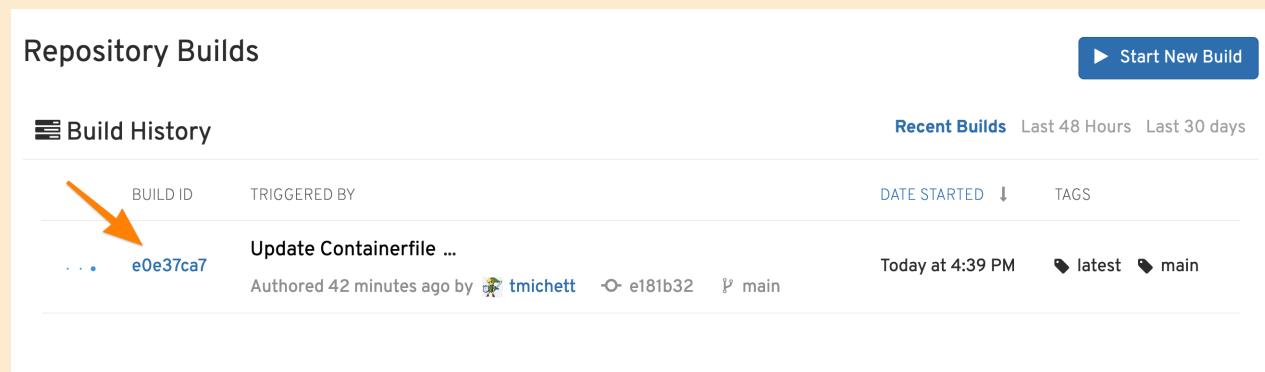
Figure 21. Testing Build Trigger

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



The screenshot shows a modal dialog titled "Manually Start Build Trigger". It contains a section for "Push to GitHub repository `tmichett/OCP_Demos`". Below it is a "Branch/Tag:" input field with a placeholder "Enter or select Branch/Tag". At the bottom right of the dialog are two buttons: "Start Build" (highlighted with an orange arrow) and "Cancel".

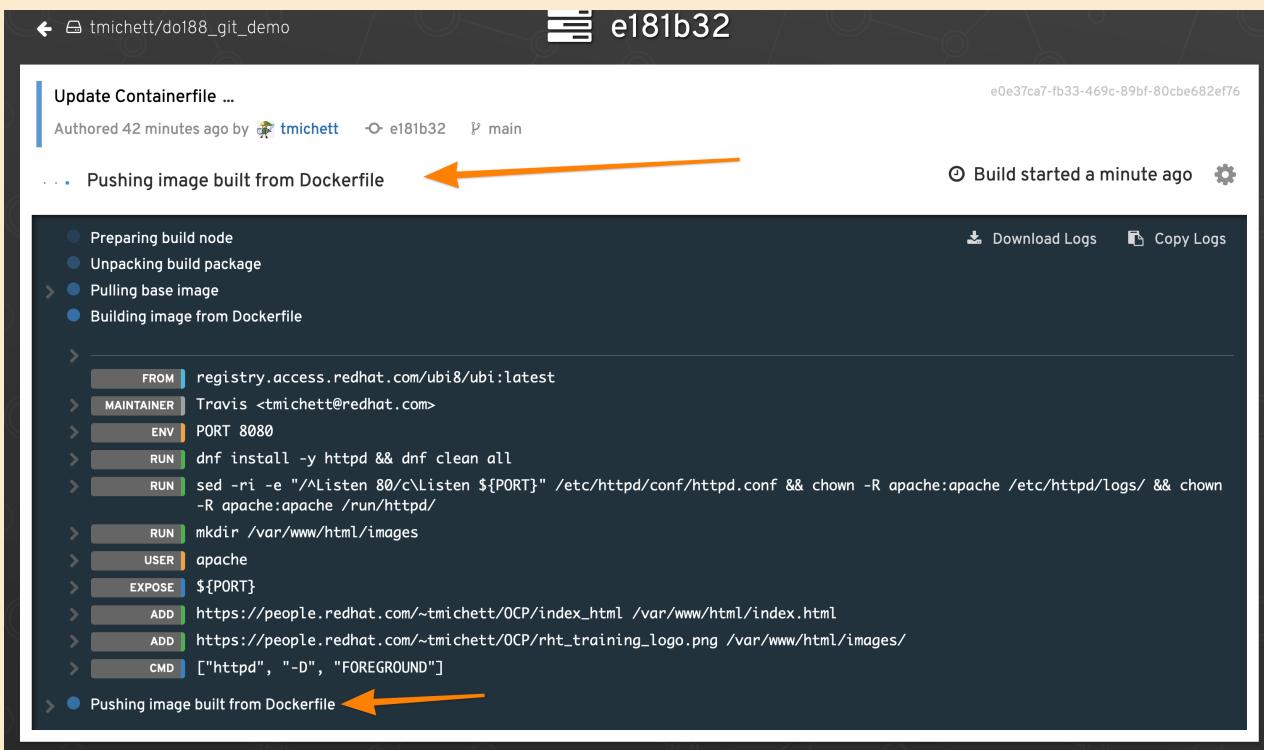
Figure 22. Manually Starting Build Trigger



The screenshot shows the "Repository Builds" page for the repository `tmichett / do188_git_demo`. The "Build History" section displays a single entry: "Update Containerfile ...". This entry was triggered by "Authored 42 minutes ago by  tmichett" and has a build ID of "e0e37ca7". The build was started "Today at 4:39 PM" and has tags "latest" and "main".

Figure 23. Manual Build

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



```

tmichett/do188_git_demo e181b32
Update Containerfile ...
Authored 42 minutes ago by tmichett · e181b32 · main
Pushing image built from Dockerfile →
Build started a minute ago · ⚙️
Preparing build node
Unpacking build package
Pulling base image
Building image from Dockerfile
FROM registry.access.redhat.com/ubi8/ubi:latest
MAINTAINER Travis <tmichett@redhat.com>
ENV PORT 8080
RUN dnf install -y httpd && dnf clean all
RUN sed -ri -e "/^Listen 80/cListen ${PORT}" /etc/httpd/conf/httpd.conf && chown -R apache:apache /etc/httpd/logs/ && chown -R apache:apache /run/httpd/
RUN mkdir /var/www/html/images
USER apache
EXPOSE ${PORT}
ADD https://people.redhat.com/~tmichett/OCP/index.html /var/www/html/index.html
ADD https://people.redhat.com/~tmichett/rht_training_logo.png /var/www/html/images/
CMD ["httpd", "-D", "FOREGROUND"]
Pushing image built from Dockerfile ←

```

Figure 24. Build Log and Verification



Figure 25. 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.
```

```

```

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 13. 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 14. 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 15. 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

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

Section Info Here

5.1.1. Copy-on-write File System

5.1.1.1. Implications of a COW File System

5.1.2. Store Data on Host Machine

5.1.3. Storing Data with Bind Mounts

5.1.3.1. Troubleshoot Bind Mounts

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

6.2. Remote Debugging Containers

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

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

8.2. Multi-pod Applications

Section Info Here

Appendix A: EX188 Exam Objectives

The **EX188** exam objectives are listed here: . For convenience, a snapshot of the objectives have been placed in this guide.