



Red Hat Training and Certification

Student Workbook (ROLE)

Red Hat OpenShift Container Platform 4.14 DO188

**Red Hat OpenShift Development I:
Introduction to Containers with Podman**
Edition 2



Join a community dedicated to learning open source

The Red Hat® Learning Community is a collaborative platform for users to accelerate open source skill adoption while working with Red Hat products and experts.



Network with tens of thousands of community members



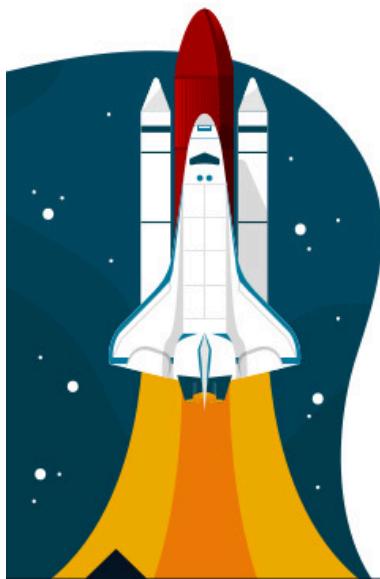
Engage in thousands of active conversations and posts



Join and interact with hundreds of certified training instructors



Unlock badges as you participate and accomplish new goals



This knowledge-sharing platform creates a space where learners can connect, ask questions, and collaborate with other open source practitioners.

Access free Red Hat training videos

Discover the latest Red Hat Training and Certification news

Connect with your instructor - and your classmates - before, after, and during your training course.

Join peers as you explore Red Hat products

Join the conversation learn.redhat.com



Copyright © 2020 Red Hat, Inc. Red Hat, Red Hat Enterprise Linux, the Red Hat logo, and Ansible are trademarks or registered trademarks of Red Hat, Inc. or its subsidiaries in the United States and other countries.

Red Hat OpenShift Development I: Introduction to Containers with Podman

Red Hat OpenShift Container Platform 4.14 DO188
Red Hat OpenShift Development I: Introduction to Containers
with Podman
Edition 2 20240723
Publication date 20240723

Authors: Marek Czernek, Jaime Ramírez Castillo
Course Architect: Ravi Srinivasan
DevOps Engineers: Richard Allred, Zachary Guterman
Editor: Sam Ffrench

© 2024 Red Hat, Inc.

The contents of this course and all its modules and related materials, including handouts to audience members, are © 2024 Red Hat, Inc.

No part of this publication may be stored in a retrieval system, transmitted or reproduced in any way, including, but not limited to, photocopy, photograph, magnetic, electronic or other record, without the prior written permission of Red Hat, Inc.

This instructional program, including all material provided herein, is supplied without any guarantees from Red Hat, Inc. Red Hat, Inc. assumes no liability for damages or legal action arising from the use or misuse of contents or details contained herein.

If you believe Red Hat training materials are being used, copied, or otherwise improperly distributed, please send email to training@redhat.com [mailto:training@redhat.com] or phone toll-free (USA) +1 (866) 626-2994 or +1 (919) 754-3700.

Red Hat, Red Hat Enterprise Linux, the Red Hat logo, JBoss, OpenShift, Fedora, Hibernate, Ansible, RHCA, RHCE, RHCSA, Ceph, and Gluster are trademarks or registered trademarks of Red Hat, Inc. or its subsidiaries in the United States and other countries.

Linux® is the registered trademark of Linus Torvalds in the United States and other countries.

Java® is a registered trademark of Oracle American, Inc. and/or its affiliates.

XFS® is a registered trademark of Hewlett Packard Enterprise Development LP or its subsidiaries in the United States and/or other countries.

MySQL® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js® is a trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack word mark and the Square O Design, together or apart, are trademarks or registered trademarks of OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. Red Hat, Inc. is not affiliated with, endorsed by, or sponsored by the OpenStack Foundation or the OpenStack community.

All other trademarks are the property of their respective owners.

Contributors: Alejandro Serna-Borja Lencina, Eduardo Ramírez Ronco, Jaime Yagüe, Jose Henrique Cardoso, Pablo Solar Vilariño

Document Conventions	ix
Admonitions	ix
Inclusive Language	x
Introduction	xi
Red Hat OpenShift Development I: Introduction to Containers with Podman	xi
Orientation to the Classroom Environment	xii
Performing Lab Exercises	xvii
1. Introduction and Overview of Containers	1
Introduction to Containers	2
Quiz: Introduction to Containers	6
Introduction to Kubernetes and OpenShift	10
Quiz: Introduction to Kubernetes and OpenShift	13
Summary	17
2. Podman Basics	19
Creating Containers with Podman	20
Guided Exercise: Creating Containers with Podman	26
Container Networking Basics	31
Quiz: Container Networking Basics	34
Accessing Containerized Network Services	38
Guided Exercise: Accessing Containerized Network Services	40
Accessing Containers	45
Guided Exercise: Accessing Containers	50
Managing the Container Lifecycle	53
Guided Exercise: Managing the Container Lifecycle	60
Lab: Podman Basics	64
Summary	70
3. Container Images	71
Container Image Registries	72
Guided Exercise: Container Image Registries	78
Managing Images	81
Guided Exercise: Managing Images	87
Lab: Container Images	90
Summary	95
4. Custom Container Images	97
Create Images with Containerfiles	98
Guided Exercise: Create Images with Containerfiles	102
Build Images with Advanced Containerfile Instructions	109
Guided Exercise: Build Images with Advanced Containerfile Instructions	119
Rootless Podman	123
Guided Exercise: Rootless Podman	129
Lab: Custom Container Images	133
Summary	142
5. Persisting Data	143
Volume Mounting	144
Guided Exercise: Volume Mounting	151
Working with Databases	155
Guided Exercise: Working with Databases	160
Lab: Persisting Data	169
Summary	174
6. Troubleshooting Containers	175
Container Logging and Troubleshooting	176

Guided Exercise: Container Logging and Troubleshooting	180
Remote Debugging Containers	186
Guided Exercise: Remote Debugging Containers	188
Lab: Troubleshooting Containers	195
Summary	204
7. Multi-container Applications with Compose	205
Compose Overview and Use Cases	206
Quiz: Compose Overview and Use Cases	213
Build Developer Environments with Compose	221
Guided Exercise: Build Developer Environments with Compose	223
Lab: Multi-container Applications with Compose	232
Summary	242
8. Container Orchestration with OpenShift and Kubernetes	243
Deploy Applications in OpenShift	244
Guided Exercise: Deploy Applications in OpenShift	254
Multi-pod Applications	258
Guided Exercise: Multi-pod Applications	264
Lab: Container Orchestration with Kubernetes and OpenShift	269
Summary	277
9. Comprehensive Review	279
Comprehensive Review	280
Lab: Comprehensive Review	282

Document Conventions

This section describes various conventions and practices that are used throughout all Red Hat Training courses.

Admonitions

Red Hat Training courses use the following admonitions:



References

These describe where to find external documentation that is relevant to a subject.



Note

Notes are tips, shortcuts, or alternative approaches to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on something that makes your life easier.



Important

Important sections provide details of information that is easily missed: configuration changes that apply only to the current session, or services that need restarting before an update applies. Ignoring these admonitions will not cause data loss, but might cause irritation and frustration.



Warning

Do not ignore warnings. Ignoring these admonitions will most likely cause data loss.

Inclusive Language

Red Hat Training is currently reviewing its use of language in various areas to help remove any potentially offensive terms. This is an ongoing process and requires alignment with the products and services that are covered in Red Hat Training courses. Red Hat appreciates your patience during this process.

Introduction

Red Hat OpenShift Development I: Introduction to Containers with Podman

Red Hat OpenShift Development I: Introduction to Containers with Podman (DO188) introduces students to building, running, and managing containers with Podman and Red Hat OpenShift. This course helps students build the core skills for developing containerized applications through hands-on experience. These skills can be applied using all versions of OpenShift, including Red Hat OpenShift on AWS (ROSA), Azure Red Hat OpenShift (ARO), and OpenShift Container Platform. This course is based on Red Hat® Enterprise Linux® 9, Podman 4.4 and Red Hat OpenShift® 4.14.

Course Objectives

- Building and managing containers with Podman for deployment on a Kubernetes and OpenShift 4 cluster.
- This course prepares the student to take the Red Hat Certified Specialist in containerized applications exam (EX188).

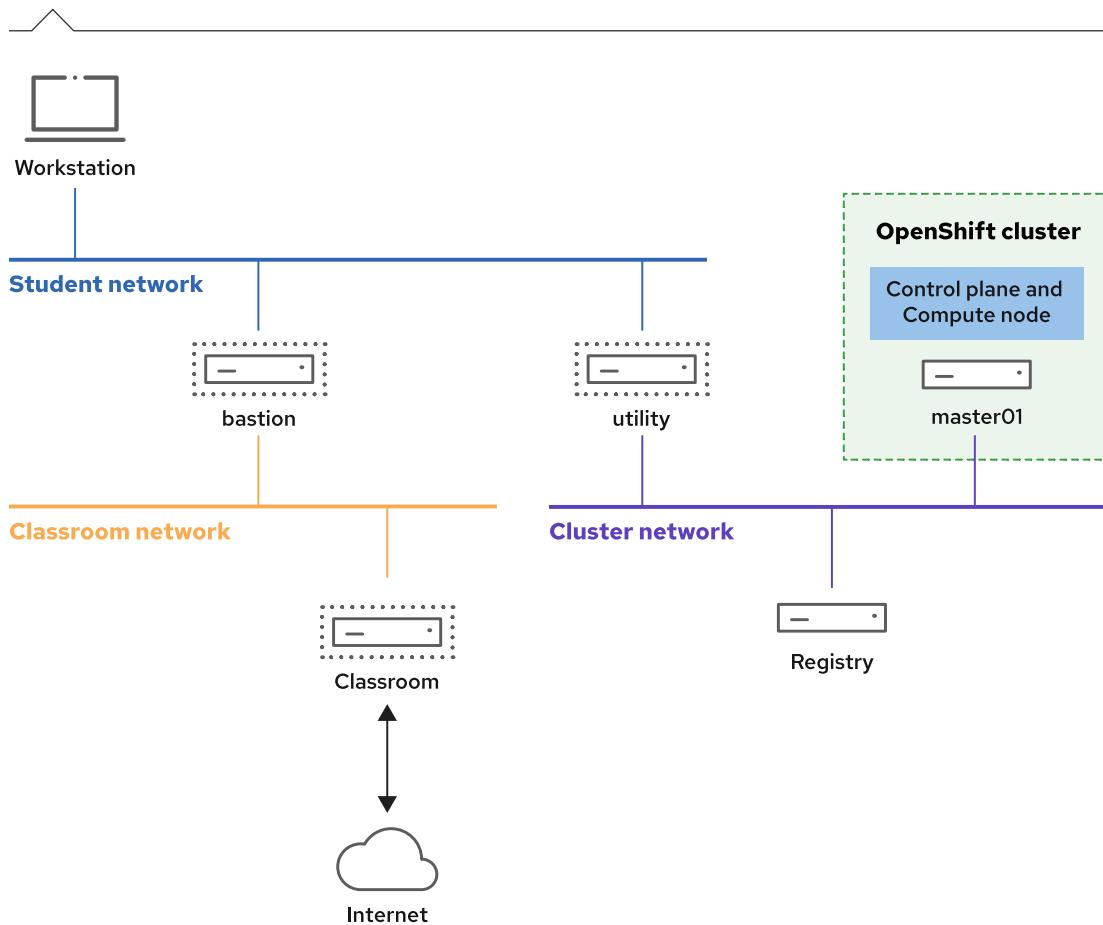
Audience

- Developers and site reliability engineers that are new to container technology.

Prerequisites

- Take our free assessment to gauge whether this offering is the best fit for your skills. Some experience with web application architectures and their corresponding technologies. Experience in the use of a Linux terminal session, issuing operating system commands, and familiarity with shell scripting is recommended.

Orientation to the Classroom Environment



In this course, the main computer system for hands-on learning activities is **workstation**.

The system called **bastion** must always be running.

These two systems are in the `lab.example.com` DNS domain.

All student computer systems have a standard user account, `student`, which has `student` as the password. The `root` password on all student systems is `redhat`.

Classroom Machines

Machine name	IP addresses	Role
<code>workstation.lab.example.com</code>	172.25.250.9	Graphical workstation that the student uses
<code>bastion.lab.example.com</code>	172.25.250.254	Router to link VMs to central course servers

Machine name	IP addresses	Role
classroom.lab.example.com	172.25.252.254	Server to host the required classroom materials for the course
utility.lab.example.com	172.25.250.253	Server to provide supporting services that the RHOCP cluster requires, including DHCP and NFS and routing to the RHOCP servers
master01.ocp4.example.com	192.168.50.10	An OpenShift control plane and compute node
registry.ocp4.example.com	192.168.50.50	Server to provide an image registry service

The **bastion** system acts as a router between the network that connects the student machines and the classroom network. If **bastion** is down, then other student machines might not function properly, or might even hang during boot.

The **utility** system acts as a router between the network that connects the OpenShift cluster machines and the student network. If **utility** is down, then the OpenShift cluster does not function properly, or might even hang during boot.

Students use the **workstation** machine to access a dedicated OpenShift cluster, for which they have cluster administrator privileges.

OpenShift Access Methods

Access method	Endpoint
Web console	https://console-openshift-console.apps.ocp4.example.com
API	https://api.ocp4.example.com:6443

The OpenShift cluster has a standard user account, **developer**, which has **developer** as the password. The administrative account, **admin**, has **redhatocp** as the password.

Troubleshooting

Cannot log in to RHOCP

Red Hat OpenShift Container Platform (RHOCP) can take a long time to start. Consequently, you might encounter authentication errors when executing lab scripts, for example:

```
[student@workstation ~]$ lab start openshift-multipod
Starting lab.

SUCCESS Copying container files
SUCCESS Copy exercise files
FAIL    Verifying your OpenShift API URL
        - API could not be reached: https://api.ocp4.example.com:6443
        - Cannot continue starting lab
```

You might also encounter errors when executing the `oc login` command, for example:

```
[student@workstation ~]$ oc login -u developer -p developer \
https://api.ocp4.example.com:6443
error: EOF
```

In that case, wait for RHOCUP to come online, and try again. This can take up to 20 minutes, depending on the classroom load.

Controlling Your Systems

You are assigned remote computers in a Red Hat Online Learning (ROLE) classroom. Self-paced courses are accessed through a web application that is hosted at `rol.redhat.com` [`http://rol.redhat.com`]. Log in to this site with your Red Hat Customer Portal user credentials.

Controlling the Virtual Machines

The virtual machines in your classroom environment are controlled through web page interface controls. The state of each classroom virtual machine is displayed on the **Lab Environment** tab.

Machine States

Virtual Machine State	Description
building	The virtual machine is being created.
active	The virtual machine is running and available. If it just started, it still might be starting services.
stopped	The virtual machine is completely shut down. On starting, the virtual machine boots into the same state that it was in before shutdown. The disk state is preserved.

Classroom Actions

Button or Action	Description
CREATE	Create the ROLE classroom. Creates and starts all the virtual machines that are needed for this classroom. Creation can take several minutes to complete.

Button or Action	Description
CREATING	The ROLE classroom virtual machines are being created. Creates and starts all the virtual machines that are needed for this classroom. Creation can take several minutes to complete.
DELETE	Delete the ROLE classroom. Destroys all virtual machines in the classroom. All saved work on those systems' disks is lost.
START	Start all virtual machines in the classroom.
STARTING	All virtual machines in the classroom are starting.
STOP	Stop all virtual machines in the classroom.

Machine Actions

Button or Action	Description
OPEN CONSOLE	Connect to the system console of the virtual machine in a new browser tab. You can log in directly to the virtual machine and run commands, when required. Normally, log in to the workstation virtual machine only, and from there, use ssh to connect to the other virtual machines.
ACTION > Start	Start (power on) the virtual machine.
ACTION > Shutdown	Gracefully shut down the virtual machine, preserving disk contents.
ACTION > Power Off	Forcefully shut down the virtual machine, while still preserving disk contents. This action is equivalent to removing the power from a physical machine.
ACTION > Reset	Forcefully shut down the virtual machine and reset associated storage to its initial state. All saved work on that system's disks is lost.

At the start of an exercise, if instructed to reset a single virtual machine node, click **ACTION > Reset** for only that specific virtual machine.

If you are instructed to reset all virtual machines, click **ACTION > Reset** on each virtual machine in the list.

If you want to return the classroom environment to its original state at the start of the course, then click **DELETE** to remove the entire classroom environment. After the lab is deleted, then click **CREATE** to provision a new set of classroom systems.



Warning

The **DELETE** operation cannot be undone. All completed work in the classroom environment is lost.

The Auto-stop and Auto-destroy Timers

The Red Hat Online Learning enrollment entitles you to a set allotment of computer time. To help to conserve your allotted time, the ROLE classroom uses timers, which shut down or delete the classroom environment when the appropriate timer expires.

To adjust the timers, locate the two + buttons at the bottom of the course management page. Click the auto-stop + button to add another hour to the auto-stop timer. Click the auto-destroy + button to add another day to the auto-destroy timer. Auto-stop has a maximum of 11 hours, and auto-destroy has a maximum of 14 days. Be careful to keep the timers set while you are working, so that your environment is not unexpectedly shut down. Be careful not to set the timers unnecessarily high, which could waste your subscription time allotment.

Performing Lab Exercises

You might see the following lab activity types in this course:

- A *guided exercise* is a hands-on practice exercise that follows a presentation section. It walks you through a procedure to perform, step by step.
- A *quiz* is typically used when checking knowledge-based learning, or when a hands-on activity is impractical for some other reason.
- An *end-of-chapter lab* is a gradable hands-on activity to help you to test your learning. You work through a set of high-level steps, based on the guided exercises in that chapter, but the steps do not walk you through every command. A solution is provided with a step-by-step walk-through.
- A *comprehensive review lab* is used at the end of the course. It is also a gradable hands-on activity, and might cover content from the entire course. You work through a specification of what to accomplish in the activity, without receiving the specific steps to do so. Again, a solution is provided with a step-by-step walk-through that meets the specification.

To prepare your lab environment at the start of each hands-on activity, run the `lab start` command with a specified activity name from the activity's instructions. Likewise, at the end of each hands-on activity, run the `lab finish` command with that same activity name to clean up after the activity. Each hands-on activity has a unique name within a course.

The syntax for running an exercise script is as follows:

```
[student@workstation ~]$ lab action exercise
```

The `action` is a choice of `start` or `finish`. All exercises support `start` and `finish`.

start

The `start` action verifies the required resources to begin an exercise. It might include configuring settings, creating resources, checking prerequisite services, and verifying necessary outcomes from previous exercises. You can perform an exercise at any time, even without performing preceding exercises.

finish

The `finish` action cleans up resources that were configured during the exercise. You can perform an exercise as many times as you want.

The `lab` command supports tab completion. For example, to list all exercises that you can start, enter `lab start` and then press the Tab key twice.

Chapter 1

Introduction and Overview of Containers

Goal

Describe how containers facilitate application development.

Objectives

- Describe the basics of containers and how containers differ from Virtual Machines.
- Describe container orchestration and the features of Red Hat OpenShift.

Sections

- Introduction to Containers (and Quiz)
- Introduction to Kubernetes and OpenShift (and Quiz)

Introduction to Containers

Objectives

- Describe the basics of containers and how containers differ from Virtual Machines.

Describing Containers

In computing, a *container* is an encapsulated process that includes the required runtime dependencies for the program to run. In a container, application-specific libraries are independent of the host operating system libraries. Libraries and functions that are not specific to the containerized application are provided by the operating system and kernel. The provided libraries and functions help to ensure that the container remains compact, and that it can quickly execute and stop as needed.

A container engine creates a union file system by merging container image layers. Because container image layers are immutable, a container engine adds a writable layer for runtime file modifications. Containers are *ephemeral* by default, which means that the container engine removes the writable layer when you remove the container.

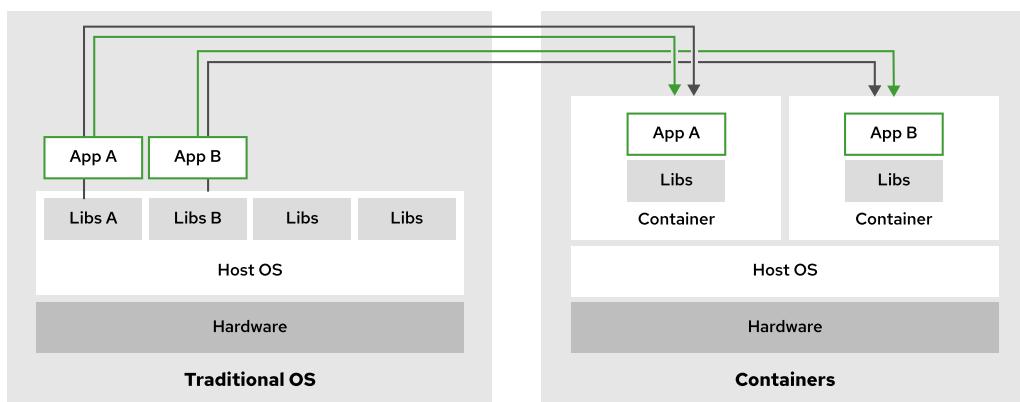


Figure 1.1: Applications in containers versus on host operating system

Containers use Linux kernel features, such as namespaces and Control Groups (cgroups). For example, containers use cgroups for resource management, such as CPU time allocation and system memory. Namespaces in particular provide the functionality to isolate processes within containers from each other and from the host system. As such, the environment within a container is Linux-based, regardless of the host operating system. When using containers on non-Linux operating systems, these Linux-specific features are often virtualized by the container engine implementation.

Containerization originated from technologies such as `chroot`, a method to partially or fully isolate an environment, and evolved to the *Open Container Initiative (OCI)*, which is a governance organization that defines standards for creating and running containers. Most container engines conform to the OCI specifications, so developers can confidently build their deployable target artifacts to run as OCI containers.

Images versus Instances

Containers can be split into two similar but distinct ideas: *container images* and *container instances*. A *container image* contains effectively immutable data that defines an application and its libraries. You can use container images to create *container instances*, which are executable versions of the image that include references to networking, disks, and other runtime necessities.

You can use a single container image multiple times to create many distinct container instances. You can also run these instances across multiple hosts. The application within a container is independent of the host environment.



Note

OCI container images are defined by the `image-spec` specification, whereas OCI container instances are defined by the `runtime-spec` specification.

Another way to think about *container images* versus *container instances* is that an instance relates to an image as an object relates to a class in object-oriented programming.

Comparing Containers to Virtual Machines

Containers generally serve a similar role to *virtual machines* (VMs), where an application resides in a self-contained environment with virtualized networking for communication. Although this use case initially seems to be the same, containers have a smaller footprint, and start and stop faster than a virtual machine. For both memory and disk usage, VMs are often measured in gigabytes, whereas containers are measured in megabytes.

A VM is useful when an additional full computing environment is required, such as when an application requires specific, dedicated hardware. Additionally, a VM is preferable when an application requires a non-Linux operating system or a different kernel from the host.

Virtual Machines versus Containers

Virtual machines and containers use different software for management and functionality. Hypervisors, such as KVM, Xen, VMware, and Hyper-V, are applications that provide the virtualization functionality for VMs. The container equivalent of a hypervisor is a container engine, such as Podman.

	Virtual machines	Containers
Machine-level functionality	Hypervisor	Container engine
Management	VM management interface	Container engine or orchestration software
Virtualization level	Fully virtualized environment	Only relevant parts
Size	Measured in gigabytes	Measured in megabytes
Portability	Generally only same hypervisor	Any OCI-compliant engine

You can manage hypervisors with additional management software, which can be included with the hypervisor, or be external, such as Virtual Machine Manager with KVM. In contrast, you can

manage containers directly through the container engine itself. Additionally, you can use container orchestration tools, such as Red Hat OpenShift Container Platform (RHOC) and Kubernetes, to run and manage containers at scale. RHOC manages both containers and virtual machines from a common interface.

With VMs, interoperability is uncommon. A VM that runs on one hypervisor is usually not guaranteed to run on a different hypervisor. In contrast, containers that follow the OCI specification do not require a particular container engine to function. Many container engines can function as drop-in replacements for each other.

Deployment at Scale

Both containers and VMs can work well at various scales. Because a container requires considerably fewer resources than a VM, containers have performance and resource benefits at a larger scale. A common method in large-scale environments is to use containers that run inside VMs. This configuration takes advantage of the strong points in each technology.

Development for Containers

Containerization provides many advantages for the development process, such as easier testing and deployments, by providing tools for stability, security, and flexibility.

Testing and Workflows

One of the greatest advantages of containers for developers is the ability to scale. A developer can write software and test locally, and then deploy the finished application to a cloud server or a dedicated cluster with few or no changes. This workflow is especially useful when creating microservices, which are small and ephemeral containers that are designed to spin up and down as needed. Additionally, developers that use containers can take advantage of *Continuous Integration/Continuous Development (CI/CD)* pipelines to deploy containers to various environments. In particular, RHOC offers various integration features with CI/CD pipelines and workflows in mind.

Stability

As mentioned earlier, container images are a stable target for developers. Software applications require specific versions of libraries to be available for deployment, which can result in dependency issues or specific OS requirements. Because libraries are included in the container image, a developer can be confident of no dependency issues in a deployment. Having the libraries integrated within the container removes variability between testing and production environments. For example, a container with a specific version of Python ensures that the same version of Python is used in every testing or deployment environment.

Multi-container Applications

A multi-container application is distributed across several containers. You can run the containers from the same image for *high availability* (HA) replicas, or from several different images. For example, a developer can create an application that includes a database container that runs separately from the application's web API container. An application can rely on the container management software to provide HA replicas with multi-container options, such as Podman Pods or the `compose - spec` specification.



References

Red Hat topic for containers

<https://www.redhat.com/en/topics/containers>

Red Hat topic for virtualization

<https://www.redhat.com/en/topics/virtualization>

Podman official documentation

<https://docs.podman.io/en/latest/>

About OCI specifications

<https://opencontainers.org/about/overview/>

Kubernetes official documentation

<https://kubernetes.io/docs/home/>

For more information about Red Hat OpenShift Container Platform, refer to the documentation at

[https://access.redhat.com/documentation/en-us/
openshift_container_platform/4.14](https://access.redhat.com/documentation/en-us/openshift_container_platform/4.14)

► Quiz

Introduction to Containers

Choose the correct answers to the following questions:

- ▶ **1. What is an effectively immutable version of an application that includes its libraries?**
 - a. Container instance
 - b. Virtual machine
 - c. Container image
 - d. Container engine
- ▶ **2. What are two advantages of using containers? (Choose two.)**
 - a. Faster program execution
 - b. Stable development target
 - c. Ease of testing and deployment
 - d. Fully virtualized environment
- ▶ **3. Which organization directly governs the standards for containerization?**
 - a. Docker
 - b. Red Hat
 - c. Open Container Initiative
 - d. Linux Foundation
- ▶ **4. True or false? An application in an OCI container runs on any OCI-compliant container engine.**
 - a. True
 - b. False
- ▶ **5. What is an executable version of an application that includes references to networking, disks, and other runtime necessities?**
 - a. Container instance
 - b. Container image
 - c. Container engine
 - d. Virtual machine
- ▶ **6. True or false? A virtual machine most often has a smaller footprint compared to a container.**
 - a. True
 - b. False

► **7. What provides machine-level functionality for running containerized applications?**

- a. Container instance
- b. Container image
- c. Container engine
- d. Hypervisor

► Solution

Introduction to Containers

Choose the correct answers to the following questions:

- ▶ **1. What is an effectively immutable version of an application that includes its libraries?**
 - a. Container instance
 - b. Virtual machine
 - c. Container image
 - d. Container engine

- ▶ **2. What are two advantages of using containers? (Choose two.)**
 - a. Faster program execution
 - b. Stable development target
 - c. Ease of testing and deployment
 - d. Fully virtualized environment

- ▶ **3. Which organization directly governs the standards for containerization?**
 - a. Docker
 - b. Red Hat
 - c. Open Container Initiative
 - d. Linux Foundation

- ▶ **4. True or false? An application in an OCI container runs on any OCI-compliant container engine.**
 - a. True
 - b. False

- ▶ **5. What is an executable version of an application that includes references to networking, disks, and other runtime necessities?**
 - a. Container instance
 - b. Container image
 - c. Container engine
 - d. Virtual machine

- ▶ **6. True or false? A virtual machine most often has a smaller footprint compared to a container.**
 - a. True
 - b. False

► **7. What provides machine-level functionality for running containerized applications?**

- a. Container instance
- b. Container image
- c. Container engine
- d. Hypervisor

Introduction to Kubernetes and OpenShift

Objectives

- Describe container orchestration and the features of Red Hat OpenShift.

Kubernetes Overview

Kubernetes is an orchestration service that simplifies the deployment, management, and scaling of containerized applications. It manages complex pools of resources, such as CPU, RAM, storage, and networking. Kubernetes provides high uptime and fault tolerance for containerized application deployments, removing the concern that developers might have regarding how their applications use resources.

The smallest manageable unit in Kubernetes is a pod, which represents a single application and consists of one or more containers, including storage resources and an IP address.

Kubernetes Features

Kubernetes clusters provide a modern container platform that addresses the concerns and challenges of running applications at scale. No matter the deployment size, Kubernetes implementations deliver a robust infrastructure and ease of management:

Service discovery and load balancing

Kubernetes enables inter-service communication by assigning a single DNS entry to each set of containers. To enable the cluster to change the container's location and IP address, the requesting service must know the target's DNS name. As a result, Kubernetes can load-balance the request across the pool of containers that provide the service. For example, Kubernetes can evenly split incoming requests to an NGINX web server by taking into account the availability of the NGINX pods.

Horizontal scaling

Applications can scale up and down manually or automatically with a configuration set, with either the Kubernetes command-line interface or the web UI.

Self-healing

Kubernetes can use user-defined health checks to monitor pods to restart and reschedule them in the event of failure.

Automated rollout

Kubernetes can gradually roll out updates to your application's containers while checking their status. If something goes wrong during the rollout, then Kubernetes can roll back to the previous version of the deployment.

Secrets and configuration management

You can manage the configuration settings and secrets of your applications without rebuilding containers. Application secrets can be usernames, passwords, and service endpoints, or any configuration setting that must be kept private.

Operators

Operators are packaged Kubernetes applications that also bring the knowledge of the application's lifecycle into the Kubernetes cluster. Applications that are packaged as operators

use the Kubernetes API to update the cluster's state, and react to changes in the application state.

Red Hat OpenShift Container Platform Overview

Red Hat OpenShift Container Platform (RHOC) is a set of modular components and services that are built on top of the Kubernetes container infrastructure. RHOC adds capabilities to a production platform, such as remote management, multitenancy, increased security, monitoring and auditing, application lifecycle management, and self-service interfaces for developers.

Red Hat OpenShift Container Platform Features

RHOC adds the following features to a Kubernetes cluster:

Developer workflow

Integrates a built-in container registry, *Continuous Integration/Continuous Delivery (CI/CD)* pipelines, and *Source-to-Image (S2I)*, a tool to build artifacts from source repositories to container images.

Routes

Exposes services to the outside world easily.

Metrics and logging

Includes a built-in and self-analyzing metrics service and aggregated logging.

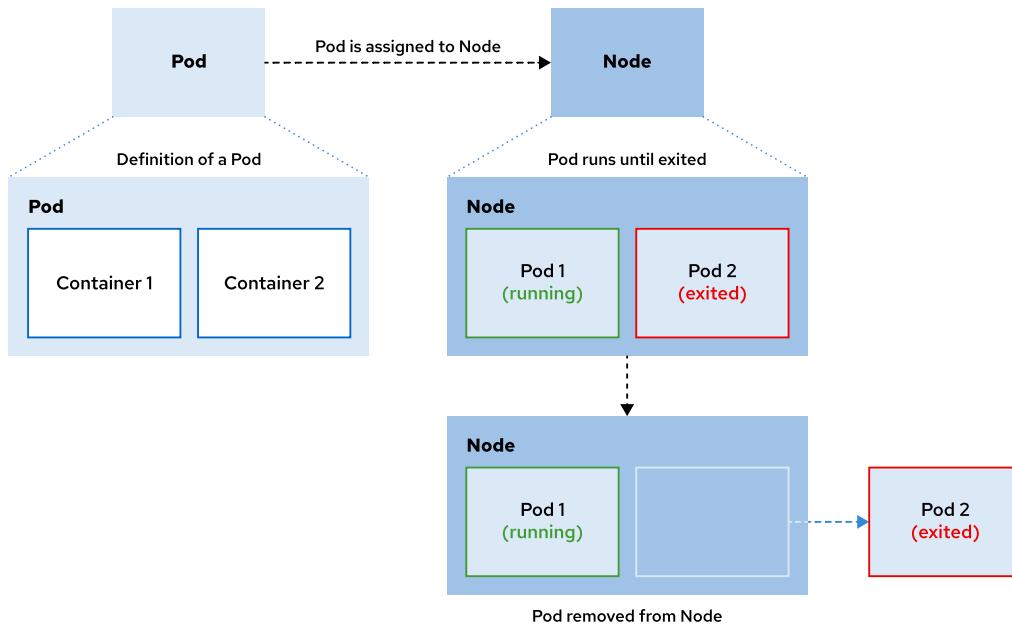
Unified UI

Provides unified tools and an integrated UI to manage the different capabilities.

Lifecycle of Applications in Red Hat OpenShift Container Platform

The following figure illustrates the basic lifecycle of an application that is deployed in a RHOC cluster:

1. Starts with the definition of a pod and the containers that it is composed of, which contain the application.
2. Pods are assigned to a healthy node.
3. Pods run until their containers exit.
4. Pods and their containers are removed from the node.



Depending on policy and exit code, RHOCP might remove pods after exiting, or might retain them to enable access to the pod container logs.



References

For more information about Kubernetes, see *Kubernetes Documentation* at <https://kubernetes.io/docs/home/>

For more information about Red Hat OpenShift Container Platform, refer to the documentation at https://access.redhat.com/documentation/en-us/openshift_container_platform/4.14

► Quiz

Introduction to Kubernetes and OpenShift

Choose the correct answers to the following questions:

► 1. **Which statement about Kubernetes is true?**

- a. Applications cannot scale up and down manually or automatically with a configuration set.
- b. You must manage the configuration settings and secrets of your applications by rebuilding containers.
- c. Kubernetes is an orchestration service that simplifies the deployment, management, and scaling of containerized applications.
- d. Kubernetes implementation and management depend on the deployment size.

► 2. **True or false? You can gradually roll updates out to your application's containers while checking their status.**

- a. True
- b. False

► 3. **True or false? Kubernetes enables inter-service communication by assigning a single DNS entry to each set of containers.**

- a. True
- b. False

► 4. **Which two features does Red Hat OpenShift Container Platform (RHOC) offer to extend Kubernetes capabilities? (Choose two.)**

- a. Unified UI that provides unified tools and an integrated UI to manage the different capabilities.
- b. Self-healing with user-defined health checks to monitor pods to restart and reschedule them in the event of failure.
- c. Developer workflow that integrates a built-in container registry, Continuous Integration/Continuous Delivery (CI/CD) pipelines, and a Source-to-Image (S2I) tool.
- d. Horizontal scaling to scale applications up and down manually or automatically with a configuration set.

► **5. Which two statements are true regarding RHOC? (Choose two.)**

- a. Integrates a built-in container registry to build artifacts from source repositories to container images.
- b. Kubernetes and RHOC are mutually exclusive.
- c. Pods remain on RHOC nodes after their containers exit.
- d. RHOC simplifies routing and load balancing.

► Solution

Introduction to Kubernetes and OpenShift

Choose the correct answers to the following questions:

► 1. **Which statement about Kubernetes is true?**

- a. Applications cannot scale up and down manually or automatically with a configuration set.
- b. You must manage the configuration settings and secrets of your applications by rebuilding containers.
- c. Kubernetes is an orchestration service that simplifies the deployment, management, and scaling of containerized applications.
- d. Kubernetes implementation and management depend on the deployment size.

► 2. **True or false? You can gradually roll updates out to your application's containers while checking their status.**

- a. True
- b. False

► 3. **True or false? Kubernetes enables inter-service communication by assigning a single DNS entry to each set of containers.**

- a. True
- b. False

► 4. **Which two features does Red Hat OpenShift Container Platform (RHOC) offer to extend Kubernetes capabilities? (Choose two.)**

- a. Unified UI that provides unified tools and an integrated UI to manage the different capabilities.
- b. Self-healing with user-defined health checks to monitor pods to restart and reschedule them in the event of failure.
- c. Developer workflow that integrates a built-in container registry, Continuous Integration/Continuous Delivery (CI/CD) pipelines, and a Source-to-Image (S2I) tool.
- d. Horizontal scaling to scale applications up and down manually or automatically with a configuration set.

► **5. Which two statements are true regarding RHOC? (Choose two.)**

- a. Integrates a built-in container registry to build artifacts from source repositories to container images.
- b. Kubernetes and RHOC are mutually exclusive.
- c. Pods remain on RHOC nodes after their containers exit.
- d. RHOC simplifies routing and load balancing.

Summary

- Containers differ from VMs by providing only the necessary runtime dependencies, such as the required application-specific libraries for a program to run, rather than a full operating system.
- Use container images to create container instances. Container instances are executable versions of the image, and include references to networking, disks, and other runtime necessities.
- Podman is a container engine to build and run containers on an individual host.
- Kubernetes and Red Hat OpenShift Container Platform (RHOCP) orchestrate containers across multiple hosts called *nodes*.
- RHOCP is a set of modular components and services that are built on top of Kubernetes to add capabilities for the following features:
 - Remote management
 - Multiple tenants
 - Increased security
 - Continuous integration
 - Continuous development

Chapter 2

Podman Basics

Goal

Manage and run containers with Podman.

Objectives

- Run a containerized service with Podman.
- Describe how containers communicate with each other.
- Expose ports to access containerized services.
- Explore running containers.
- List, stop, and delete containers with Podman.

Sections

- Creating Containers with Podman (and Guided Exercise)
- Container Networking Basics (and Quiz)
- Accessing Containerized Network Services (and Guided Exercise)
- Accessing Containers (and Guided Exercise)
- Managing the Container Lifecycle (and Guided Exercise)

Lab

- Podman Basics

Creating Containers with Podman

Objectives

- Run a containerized service with Podman.

An Introduction to Podman

Podman is an open source tool that you can use to manage your containers locally. With Podman, you can find, run, build or deploy OCI (Open Container Initiative) containers and container images.

By default, Podman is daemonless. A daemon is a process that is always running and ready for receiving incoming requests. Some other container tools use a daemon to proxy the requests, which brings a single point of failure. In addition, a daemon might require elevated privileges, which is a security concern. Podman interacts directly with containers, images, and registries without a daemon.

Podman comes in the form of a command-line interface (CLI), which is supported for several operating systems. Along with the CLI, Podman provides two additional ways to interact with your containers and automate processes, the RESTful API and a desktop application called *Podman Desktop*.

Working with Podman

After you install Podman, you can use it by running the `podman` command. The following command displays the version that you are using.

```
[user@host ~]$ podman -v  
podman version VERSION
```

Pulling and Displaying Images

Before you can run your application in a container, you must create a container image.

With Podman, you fetch container images from image registries by using the `podman pull` command. For example, the following command fetches a containerized version of Red Hat Enterprise Linux 7 from the Red Hat Registry.

```
[user@host ~]$ podman pull registry.redhat.io/rhel7/rhel:7.9  
Trying to pull registry.redhat.io/rhel7/rhel:7.9...  
Getting image source signatures  
...output omitted...  
Writing manifest to image destination  
Storing signatures  
b85986059f7663c1b89431f74cdcb783f6540823e4b85c334d271f2f2d8e06d6
```

A container image is referenced in the form `NAME:VERSION`. In the previous example, you fetch the 7.9 version of the `registry.redhat.io/rhel7/rhel` image.

Chapter 2 | Podman Basics

After the execution of the `pull` command, the image is stored locally in your system. You can list the images in your system by using the `podman images` command.

```
[user@host ~]$ podman images
REPOSITORY           TAG      IMAGE ID      CREATED     SIZE
registry.redhat.io/rhel7/rhel    7.9      52617ef413bd  4 weeks ago  216 MB
```

Container and Container Images

A container is an isolated runtime environment where applications are executed as isolated processes. The isolation of the runtime environment ensures that they do not interrupt other containers or system processes.

A container image contains a packaged version of your application, with all the dependencies necessary for the application to run. Images can exist without containers, but containers are dependent of images because containers use container images to build a runtime environment to execute applications.

Running and Displaying Containers

With the image stored in your local system, you can use the `podman run` command to create a new container that uses the image. The RHEL image from previous examples accepts Bash commands as an argument. These commands are provided as an argument and are executed within a RHEL container.

```
[user@host ~]$ podman run registry.redhat.io/rhel7/rhel:7.9 echo 'Red Hat'
Red Hat
```

In the previous example, the `echo 'Red Hat'` command is provided as an argument to the `podman run` command. Podman executes the `echo` command inside the RHEL container and displays the output of the command.



Note

If you run a container from an image that is not stored in your system then Podman tries to pull the image before running the container. Therefore, it is not necessary to execute the `pull` command first.

When the container finishes the execution of `echo`, the container is stopped because no other process keeps it running. You can list the running containers by using the `podman ps` command.

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

By default, the `podman ps` command lists the following details for your containers.

- the container's ID
- the name of the image that the container is using
- the command that the container is executing
- the time that the container was created
- the status of the container
- the exposed ports in the container
- the name of the container.

However, stopping a container is not the same as removing a container. Although the container is stopped, Podman does not remove it. You can list all containers (running and stopped) by adding the `--all` flag to the `podman ps` command.

```
[user@host ~]$ podman ps --all
CONTAINER ID  IMAGE                                COMMAND      CREATED
 STATUS        PORTS     NAMES
20236410bcef  registry.redhat.io/rhel7/rhel:7.9    echo Red Hat  1 second ago
              Exited (0) 1 second ago                  hungry_mclaren
```

You can also automatically remove a container when it exits by adding the `--rm` option to the `podman run` command.

```
[user@host ~]$ podman run --rm registry.redhat.io/rhel7/rhel:7.9 echo 'Red Hat'
Red Hat
[user@host ~]$ podman ps --all
CONTAINER ID  IMAGE                                COMMAND      CREATED
 STATUS        PORTS     NAMES
```

The container lifecycle is covered later in this course.

If a name is not provided during the creation of a container, then Podman generates a random string name for the container. It is important to define a unique name to facilitate the identification of your containers when managing their lifecycle.

You can assign a name to your containers by adding the `--name` flag to the `podman run` command.

```
[user@host ~]$ podman run --name podman_rhel7 \
registry.redhat.io/rhel7/rhel:7.9 echo 'Red Hat'
Red Hat
```

Podman can identify the containers either by the *Universal Unique Identifier (UUID)* short identifier, which is composed of twelve alphanumeric characters, or by the UUID long identifier, which is composed of 64 alphanumeric characters, as shown in the example.

```
[user@host ~]$ podman ps --all
CONTAINER ID  IMAGE                                COMMAND      CREATED
 STATUS        PORTS     NAMES
20236410bcef  registry.redhat.io/rhel7/rhel:7.9    echo Red Hat  1 second ago
              Exited (0) 1 second ago                  podman_rhel7
```

In this example, `20236410bcef` is the container's ID, or UUID short identifier. Additionally, `podman_rhel7` is listed as the container's name.

If you want to retrieve the UUID long container ID, then you can add the `--format=json` flag to the `podman ps --all` command.

```
[user@host ~]$ podman ps --all --format=json
{
  "AutoRemove": false,
  "Command": [
```

```
"echo",
"Red Hat"
],
...output omitted...
"Id": "2023...b0b2", ①
"Image": "registry.redhat.io/rhel7/rhel:7.9",
...output omitted...
```

- ① UUID long container ID.

You can retrieve information about a container in either the JSON format or as a Go template.

Exposing Containers

Many applications, such as web servers or databases, keep running indefinitely waiting for connections. Therefore, the containers for these applications must run indefinitely. At the same time, it is usually necessary for these applications to be accessed externally through a network protocol.

You can use the `-p` option to map a port in your local machine to a port inside the container. This way, the traffic in your local port is forwarded to the port inside the container, thus, allowing you to access the application from your computer.

The following example creates a new container that runs an Apache HTTP server by mapping the 8080 port in your local machine to the 8080 port inside the container.

```
[user@host ~]$ podman run -p 8080:8080 \
registry.access.redhat.com/ubi8/httpd-24:latest
...output omitted...
[Thu Apr 21 12:58:57.048491 2022] [ssl:warn] [pid 1:tid 140257793613248] AH01909:
10.0.2.100:8443:0 server certificate does NOT include an ID which matches the
server name
[Thu Apr 21 12:58:57.048899 2022] [:notice] [pid 1:tid 140257793613248]
ModSecurity for Apache/2.9.2 (http://www.modsecurity.org/) configured.
...output omitted...
[Thu Apr 21 12:58:57.136272 2022] [mpm_event:notice] [pid 1:tid 140257793613248]
AH00489: Apache/2.4.37 (Red Hat Enterprise Linux) OpenSSL/1.1.1k configured --
resuming normal operations
[Thu Apr 21 12:58:57.136332 2022] [core:notice] [pid 1:tid 140257793613248]
AH00094: Command line: 'httpd -D FOREGROUND'
```

You can access the HTTP server at `localhost:8080`.

If you want the container to run in the background, to avoid the terminal being blocked, then you can use the `-d` option.

```
[user@host ~]$ podman run -d -p 8080:8080 \
registry.access.redhat.com/ubi8/httpd-24:latest
b7eb467781106e4f416ba79cede91152239bfc74f6a570c6d70baa4c64fa636a
```

The networking capabilities of Podman are covered later in this course.

Using Environment Variables

Environment variables are variables used in your applications that are set outside of the program. The operating system or the environment where the application runs provides the value of the variable. You can access the environment variable in your application at runtime. For example, in Node.js, you can access environment variables by using `process.env.VARIABLE_NAME`.

Environment variables are a useful and safe way of injecting environment-specific configuration values into your application. For example, your application might use a database hostname that is different for each application environment, such as the `database.local`, `database.stage`, or `database.test` hostnames.

You can pass environment variables to a container by using the `-e` option. In the following example, an environment variable called `NAME` with the value `Red Hat` is passed. Then, the environment variable is printed by using the `printenv` command inside the container.

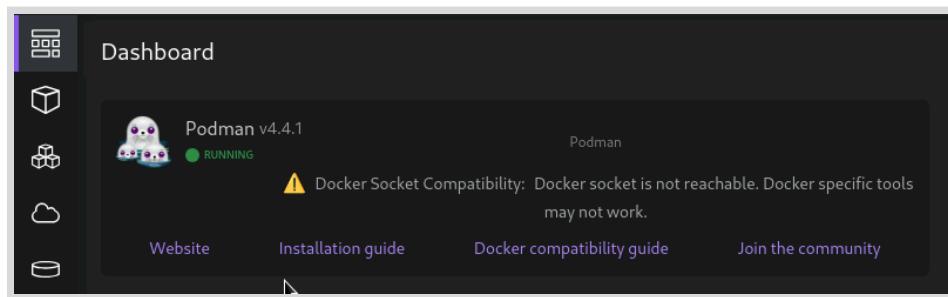
```
[user@host ~]$ podman run -e NAME='Red Hat' \
  registry.redhat.io/rhel7/rhel:7.9 printenv NAME
Red Hat
```

Environment variables are covered later in this course.

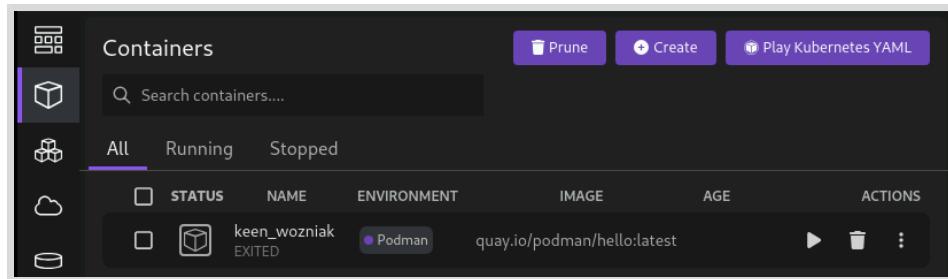
Podman Desktop

Podman Desktop is a graphical user interface, which is used to manage and interact with containers in local environments. It uses the Podman engine by default, and supports other container engines as well, such as Docker.

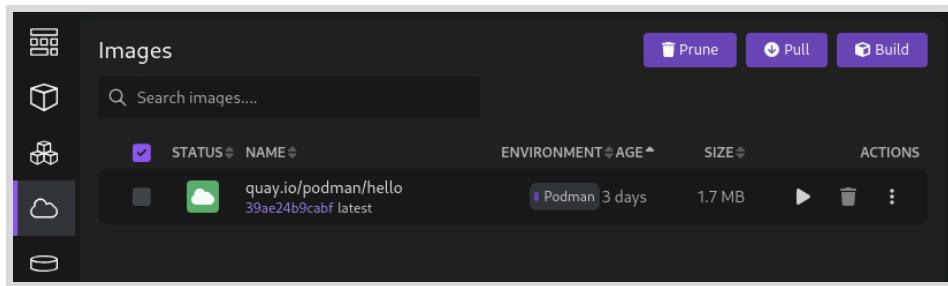
On launch, Podman Desktop displays a dashboard with information about the status of the Podman engine. The dashboard might display warnings or errors, for example, if the Podman engine is not installed, or if the Docker compatibility is not fully set up.



With Podman Desktop, you can perform many of the tasks that you can do with the `podman` CLI, such as pulling images and creating containers. For example, you can create, list, and run containers from the `Containers` section, as the following image shows:



Similarly, you can pull, list images, and create containers from those images in the **Images** section.



Podman Desktop is an addition to the podman CLI, rather than a replacement. For more advanced commands or options, which are covered later in the course, the CLI is required. Still, Podman Desktop can be useful for users who prefer graphical environments for common tasks, and for beginners who are learning about containers.

Podman Desktop is modular and extensible. You can use and create extensions that provide additional capabilities. For example, with the *Red Hat OpenShift* extension, Podman Desktop can deploy containers to Red Hat OpenShift Container Platform.

Podman Desktop is available for Linux, MacOS, and Windows. For specific installation instructions, refer to the Podman Desktop documentation.



References

Podman Official Documentation

<https://docs.podman.io/en/latest/>

Formatting output with Podman

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

Podman Desktop Documentation

<https://podman-desktop.io/docs/intro>

► Guided Exercise

Creating Containers with Podman

Create several Podman containers by using different options.

Outcomes

You should be able to run containers in Podman by using the `podman run` command.

Before You Begin

As the student user on the workstation machine, use the `lab` command to prepare your system for this exercise.

```
[student@workstation ~]$ lab start basics-creating
```

Instructions

- 1. Use the `registry.ocp4.example.com:8443/ubi8/ubi-minimal:8.5` image to create a new container that prints the `Hello Red Hat` text.

- 1.1. Use the `podman pull` command to fetch the image from the registry.

```
[student@workstation ~]$ podman pull \
  registry.ocp4.example.com:8443/ubi8/ubi-minimal:8.5
Trying to pull registry.ocp4.example.com:8443/ubi8/ubi-minimal:8.5...
...output omitted...
Writing manifest to image destination
Storing signatures
a0c247bdbd39ebaf1d6c8a2b42573311b3052ed0d67cc8eb7518b47f5e0eeeca6
```

- 1.2. Use the `podman images` command to verify that the image is available locally.

REPOSITORY	TAG	IMAGE ID	CREATED
registry.ocp4.example.com:8443/ubi8/ubi-minimal	8.5	a0c247bdbd39...	4 weeks ago
			107 MB

- 1.3. In a command line terminal, use `podman run` to create a new container. Provide an `echo` command to be executed inside the container.

```
[student@workstation ~]$ podman run --rm \
  registry.ocp4.example.com:8443/ubi8/ubi-minimal:8.5 echo 'Hello Red Hat'
Hello Red Hat
```

- 1.4. Verify that the container is not running after the execution finishes.

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

- ▶ 2. Explore setting and printing environment variables by using the `registry.ocp4.example.com:8443/ubi8/ubi-minimal:8.5` container image.
- 2.1. Use the `-e` option in the `podman run` command to set environment variables. Use the `printenv` command that is packaged in the container image to print the values of the environment variables.

```
[student@workstation ~]$ podman run --rm -e GREET=Hello -e NAME='Red Hat' \
registry.ocp4.example.com:8443/ubi8/ubi-minimal:8.5 printenv GREET NAME
Hello
Red Hat
```

- 2.2. Verify that the container is not running after the execution finishes.

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

- ▶ 3. By using the `registry.ocp4.example.com:8443/ubi8/httpd-24` image, run a new container that creates an Apache HTTP server.
- 3.1. Use the `podman run` command to pull the container image and start the `httpd` container. Use the `-p` option in the `podman run` command to redirect traffic from port 8080 on your machine to port 8080 inside the container.

```
[student@workstation ~]$ podman run --rm -p 8080:8080 \
registry.ocp4.example.com:8443/ubi8/httpd-24
[Thu Apr 21 12:58:57.048491 2022] [ssl:warn] [pid 1:tid 140257793613248] AH01909:
10.0.2.100:8443:0 server certificate does NOT include an ID which matches the
server name
[Thu Apr 21 12:58:57.048899 2022] [:notice] [pid 1:tid 140257793613248]
ModSecurity for Apache/2.9.2 (http://www.modsecurity.org/) configured.
...output omitted...
[Thu Apr 21 12:58:57.136272 2022] [mpm_event:notice] [pid 1:tid 140257793613248]
AH00489: Apache/2.4.37 (Red Hat Enterprise Linux) OpenSSL/1.1.1k configured --
resuming normal operations
[Thu Apr 21 12:58:57.136332 2022] [core:notice] [pid 1:tid 140257793613248]
AH00094: Command line: 'httpd -D FOREGROUND'
```

The container keeps running and the logs are displayed in your terminal.

- 3.2. In a web browser, navigate to `http://localhost:8080`. Verify that the HTTP server is running at the 8080 port.
- 3.3. Go back to your command-line terminal. Press `Ctrl + C` to stop the container.

```
...output omitted...
^C
[Thu Apr 21 13:00:52.516503 2022] [mpm_event:notice] [pid 1:tid 140481583287744]
AH00491: caught SIGTERM, shutting down
```

- 3.4. Create the container again by adding the `-d` option. The container runs in the background.

```
[student@workstation ~]$ podman run --rm -d \
-p 8080:8080 registry.ocp4.example.com:8443/ubi8/httpd-24
c1db10ec1ba61afcc7bb482b8d9cd42995473a4f4523ae15f3253e896a45d61a
```

- 3.5. Verify that the container is running in the background.

```
[student@workstation ~]$ podman ps
c1db10ec1ba6 registry.ocp4.example.com:8443/ubi8/httpd-24:latest /usr/bin/
run-http... About a minute ago Up About a minute ago 0.0.0.0:8080->8080/tcp
distracted_lumiere
```

► 4. Use Podman Desktop to list images and containers.

- 4.1. Open Podman Desktop.

```
[student@workstation ~]$ podman-desktop
```

- 4.2. If a telemetry alert displays, then click **Ok**.

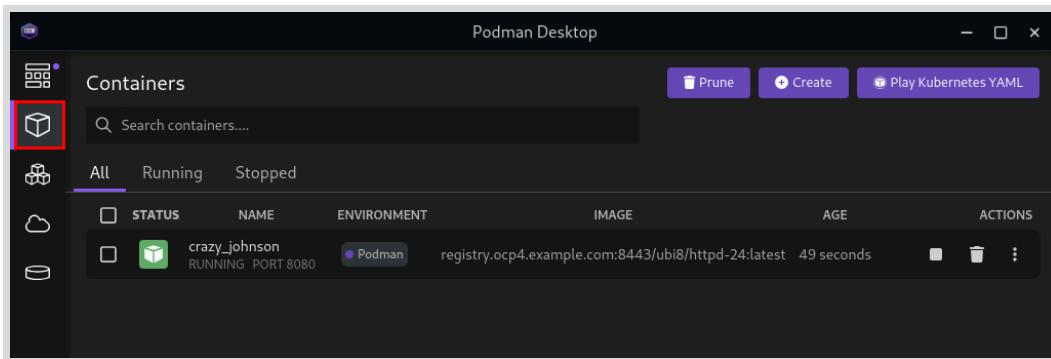


Important

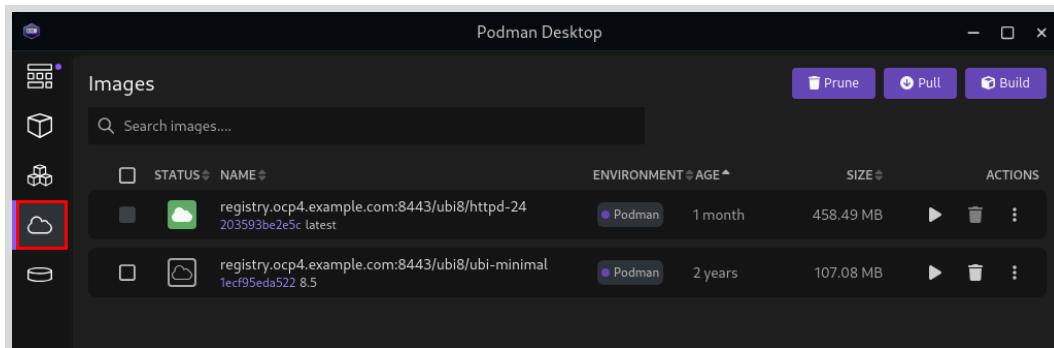
The telemetry dialog might display under Podman Desktop, blocking any interaction with the main window. If you experience this problem, then move the main window to find the telemetry dialog.

This problem is a known bug. For more details, see <https://github.com/containers/podman-desktop/issues/664>.

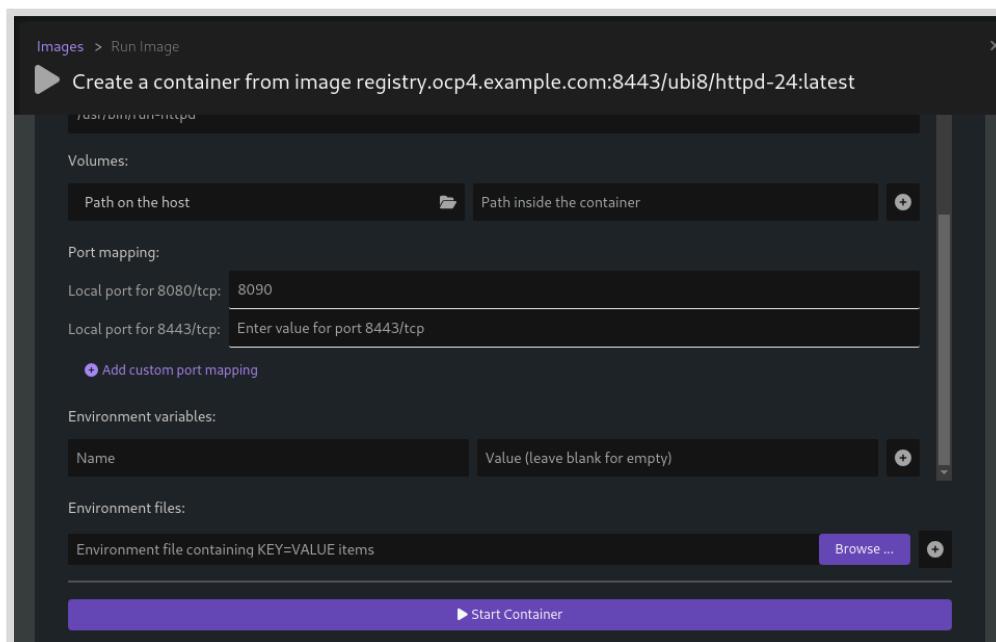
- 4.3. Click **Containers** in the Podman Desktop navigation panel. The list includes the containers that you created in the exercise. Verify that the Apache HTTP container is running.



- 4.4. Click **Images**. The list of local images displays the `ubi8/ubi-minimal` and `ubi8/httpd-24` images.



- ▶ 5. Use Podman Desktop to start another Apache HTTP container that maps its port to port 8090 in the workstation.
 - 5.1. Click the ► icon of the `ubi8/httpd-24` image to run a new container based on this image.
 - 5.2. In the container creation form, enter the following values:
 - Container name: `desktop-test`
 - Local port for 8080/tcp: `8090`
 - Local port for 8443/tcp: Delete the default value and leave blank



- 5.3. Click **Start Container**.

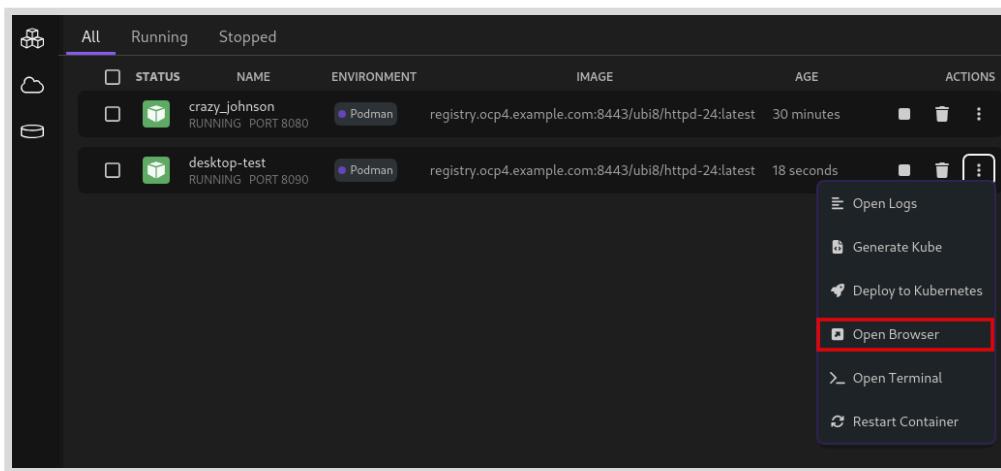


Note

You can safely ignore the SELinux security warning that displays at the top of the desktop.

- 5.4. In the containers list, verify that `desktop-test` is running.

- 5.5. In the **desktop-test** container, scroll to the right if necessary, click **:> Open Browser**, and then click **Yes** in the confirmation dialog box.



- 5.6. Verify that the browser can access the HTTP server at `http://localhost:8090`.
- 5.7. Return to Podman Desktop. In the **desktop-test** container, click the **Delete Container** icon to delete the container, and then close Podman Desktop.

Finish

On the **workstation** machine, use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish basics-creating
```

Container Networking Basics

Objectives

- Describe how containers communicate with each other.

Container Networking Basics

Podman comes with a network called podman. By default, containers are attached to this network and can use it to communicate with one another.

However, you might need to create a new Podman network to better suit the increased communication needs of most applications. For example, the containers running an application API and database can use a separate Podman network to isolate their communication from other containers. Similarly, that same API container can use yet another network to isolate communication with a third container that hosts the application UI.

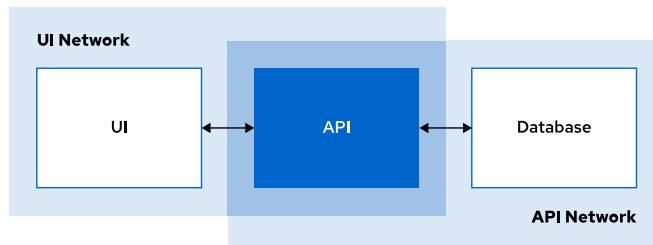


Figure 2.8: Example of isolated communication by using Podman networks

In the preceding example diagram, the UI and API containers are attached to the ui-network Podman network. The API and database containers are attached to the api-network Podman network.

Managing Podman Networks

Podman network management is done via the `podman network` subcommand. This subcommand includes the following operations:

`podman network create`

Creates a new Podman network. This command accepts various options to configure properties of the network, including gateway address, subnet mask, and whether to use IPv4 or IPv6.

`podman network ls`

Lists existing networks and a brief summary of each. Options for this command include various filters and an output format to list other values for each network.

`podman network inspect`

Outputs a detailed JSON object containing configuration data for the network.

`podman network rm`

Removes a network.

podman network prune

Removes any networks that are not currently in use by any running containers.

podman network connect

Connects an already running container to or from an existing network. Alternatively, connect containers to a Podman network on container creation by using the `--net` option. The `disconnect` command disconnects a container from a network.

For example, the following command creates a new Podman network called `example-net`:

```
[user@host ~]$ podman network create example-net
```

To connect a new container to this Podman network, use the `--net` option. The following example command creates a new container called `my-container`, which is connected to the `example-net` network.

```
[user@host ~]$ podman run -d --name my-container \
--net example-net container-image:latest
```

When you create new containers, you can connect them to multiple networks by specifying network names in a comma-separated list. For example, the following command creates a new container called `double-connector` that connects to both the `postgres-net` and `redis-net` networks.

```
[user@host ~]$ podman run -d --name double-connector \
--net postgres-net,redis-net \
container-image:latest
```

Alternatively, if the `my-container` container is already running, then run the following command to connect it to the `example-net` network:

```
[user@host ~]$ podman network connect example-net my-container
```

**Note**

Starting in Podman v4.2.0, the `podman network create` command supports the `isolate` option with the default bridge driver. This option isolates the network by blocking any traffic from it to any other network with the `isolate` option enabled. Use the `podman network create` command with the `-o isolate` option to enable isolation.

**Important**

If a network is not specified with the `podman run` command, then the container connects to the default network. The default network uses the `slirp4netns` network mode, and the networks that you create with the `podman network create` command use the bridge network mode. If you try to connect a bridge network to a container by using the `slirp4netns` network mode, then the command fails.

Enabling Domain Name Resolution

When you use the default Podman network, the domain name system (DNS) for other containers in that network is disabled. To enable DNS resolution between containers, create a Podman network and connect your containers to that network.

When using a network with DNS enabled, a container's hostname or alias is the name assigned to the container. For example, if a container is started with the following command, then the other containers on the `test-net` network can make requests to the first container by using the `basic-container` hostname. The `basic-container` hostname resolves to the current IP address of the `basic-container` container.

```
[user@host ~]$ podman run --net test-net --name basic-container example-image
```

You can modify the default podman network and override the predefined value of disabled DNS, but this is discouraged, and the preferred solution is to create a new network with enabled DNS.

Connecting Containers

You can connect containers to one or more Podman networks. After a container connects to a network, the container can communicate with other containers on that network. However, even though the containers are reachable to one another, other components might prevent connections. For example, firewall rules might block a connection coming from another container. By default, a container is available within any network that the container connects to.

For example, consider a running container called `nginx-host` that uses the `example-net` network. The container exposes an HTTP server on port 8080. Within another container that uses the `example-net` network, the following `curl` command resolves to the root of the HTTP server.

```
[user@host ~]$ curl http://nginx-host:8080
```



References

Basic Networking Guide for Podman

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

Podman 4.0's new network stack: What you need to know

<https://www.redhat.com/sysadmin/podman-new-network-stack>

► Quiz

Container Networking Basics

You are tasked with designing the Podman network layout for an application that uses the following set of containers.

The web UI container for the application is called `app-ui`. This container uses the `app-ui:v2.1.1` image tag and the `app-api` network.

The API container for the application is called `app-api`. This container uses the `app-api:v2.3.4` image tag, and the `app-api` and `app-db` networks.

The PostgreSQL database server container is called `app-db`. This container uses the `postgresql:11` image tag and the `app-db` network.

Each container must communicate with at least one other container via Podman networks. Isolate network communication so that only necessary connections are available. The containers use DNS hostnames to connect to one another.

Given the preceding scenario, choose the correct answers to the following questions:

- ▶ **1. How many Podman networks are necessary to meet the requirements?**
 - a. 0
 - b. 1
 - c. 2
 - d. 3
 - e. 4
 - f. 5
- ▶ **2. Which of the following commands creates a new Podman network called app-ui?**
 - a. `podman network app-ui`
 - b. `podman network create app-ui`
 - c. `podman create app-ui`
 - d. `podman create-network app-ui`
- ▶ **3. Which of the following commands prints metadata about the network called app-api?**
 - a. `podman inspect network app-api`
 - b. `podman network inspect app-api`
 - c. `podman network ls app-api`
 - d. `podman network metadata app-api`

- 4. Which of the following commands creates the app-db container attached to the appropriate networks?
- a. podman run -d --name app-db app-db postgresql
 - b. podman run -d --name app-db --net app-api,app-ui postgresql:11
 - c. podman run -d --name app-db --connect app-db postgresql:11
 - d. podman run -d --name app-db --net app-db postgresql:11
- 5. Which of the following commands creates the app-api container attached to the appropriate networks?
- a. podman run -d --name app-api -p 8080:8080 app-api:v2.3.4
 - b. podman run -d --name app-api --net app-api app-api:v2.3.4
 - c. podman run -d --name app-api --net app-api,app-db app-api:v2.3.4
 - d. podman run -d --name app-api --net app-ui,app-api,app-db app-api:v2.3.4
- 6. Could the default podman network replace one of the networks in this configuration? Why or why not?
- a. Yes, because the default podman network provides DNS by default.
 - b. Yes, but the podman network needs to be overwritten to enable DNS.
 - c. No, because normal users cannot connect containers to the default podman network.
 - d. No, because the default podman network cannot provide DNS.

► Solution

Container Networking Basics

You are tasked with designing the Podman network layout for an application that uses the following set of containers.

The web UI container for the application is called `app-ui`. This container uses the `app-ui:v2.1.1` image tag and the `app-api` network.

The API container for the application is called `app-api`. This container uses the `app-api:v2.3.4` image tag, and the `app-api` and `app-db` networks.

The PostgreSQL database server container is called `app-db`. This container uses the `postgresql:11` image tag and the `app-db` network.

Each container must communicate with at least one other container via Podman networks. Isolate network communication so that only necessary connections are available. The containers use DNS hostnames to connect to one another.

Given the preceding scenario, choose the correct answers to the following questions:

► 1. How many Podman networks are necessary to meet the requirements?

- a. 0
- b. 1
- c. 2
- d. 3
- e. 4
- f. 5

► 2. Which of the following commands creates a new Podman network called `app-ui`?

- a. `podman network app-ui`
- b. `podman network create app-ui`
- c. `podman create app-ui`
- d. `podman create-network app-ui`

► 3. Which of the following commands prints metadata about the network called `app-api`?

- a. `podman inspect network app-api`
- b. `podman network inspect app-api`
- c. `podman network ls app-api`
- d. `podman network metadata app-api`

- 4. Which of the following commands creates the app-db container attached to the appropriate networks?
- a. podman run -d --name app-db app-db postgresql
 - b. podman run -d --name app-db --net app-api,app-ui postgresql:11
 - c. podman run -d --name app-db --connect app-db postgresql:11
 - d. podman run -d --name app-db --net app-db postgresql:11
- 5. Which of the following commands creates the app-api container attached to the appropriate networks?
- a. podman run -d --name app-api -p 8080:8080 app-api:v2.3.4
 - b. podman run -d --name app-api --net app-api app-api:v2.3.4
 - c. podman run -d --name app-api --net app-api,app-db app-api:v2.3.4
 - d. podman run -d --name app-api --net app-ui,app-api,app-db app-api:v2.3.4
- 6. Could the default podman network replace one of the networks in this configuration? Why or why not?
- a. Yes, because the default podman network provides DNS by default.
 - b. Yes, but the podman network needs to be overwritten to enable DNS.
 - c. No, because normal users cannot connect containers to the default podman network.
 - d. No, because the default podman network cannot provide DNS.

Accessing Containerized Network Services

Objectives

- Expose ports to access containerized services.

Port Forwarding

A container's network namespace is isolated, which means that a networked application is only accessible within the container. *Port forwarding* maps a port from the host machine where the container runs to a port inside of a container.

The `-p` option of the `podman run` command forwards a port. The option accepts the form `HOST_PORT:CONTAINER_PORT`.

For example, the following command maps port 80 inside the container to port 8075 on the host machine.

```
[user@host ~]$ podman run -p 8075:80 my-app
```

Without a host specified, the container is assigned the broadcast address (`0.0.0.0`). This means that the container is accessible from all networks on the host machine.

To publish a container to a specific host and to limit the networks it is accessible from, use the following form.

```
[user@host ~]$ podman run -p 127.0.0.1:8075:80 my-app
```

Port 80 in the `my-app` container is available from port 8075 only from the host machine, which is accessible via the localhost `127.0.0.1` IP address.

List Port Mappings

To list port mappings for a container, use the `podman port` command. For example, the following command reveals that port 8010 of the host machine is mapped to port 8008 within the container.

```
[user@host ~]$ podman port my-app
8008/tcp -> 0.0.0.0:8010
```

The `--all` option lists port mappings for all containers.

```
[user@host ~]$ podman port --all
1aacd9cf1c76 8008/tcp -> 0.0.0.0:8010
```



Note

In the preceding example output, `1aacd9cf1c76` refers to the ID of the container.

Networking in Containers

Containers attached to Podman networks are assigned private IP addresses for each network. Other containers in the network can make requests to this IP address.

For example, a container called `my-app` is attached to the `apps` network. The following command retrieves the private IP address of the container within the `apps` network.

```
[user@host ~]$ podman inspect my-app \
-f '{{.NetworkSettings.Networks.apps.IPAddress}}'
10.89.0.2
```

Note that this IP address is only valid within the `apps` network.



References

Basic Networking Guide for Podman

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

Podman Documentation - Port Command

<https://docs.podman.io/en/latest/markdown/podman-port.1.html>

► Guided Exercise

Accessing Containerized Network Services

Run two applications that communicate by using the Podman DNS system.

Outcomes

You should be able to understand how DNS works in Podman networks.

Before You Begin

```
[student@workstation ~]$ lab start basics-exposing
```

This exercise uses two applications: `times` and `cities`. The `times` application returns the current time of a given city, and the `cities` application returns information about a specific city.

To fetch the current time of the city, the `cities` application fetches data from the `times` application.

Instructions

- ▶ 1. Examine the source code of the applications.
 - 1.1. Navigate to the `~/D0188/labs/basics-exposing` directory, where the applications are available.

```
[student@workstation ~]$ cd ~/D0188/labs/basics-exposing  
no output expected
```

- 1.2. View the contents of the `podman-info-times/app/main.go` file.

```
[student@workstation basics-exposing]$ cat podman-info-times/app/main.go  
...output omitted...  
http.ListenAndServe(":8080", r)  
...output omitted...
```

The application exposes an HTTP server on port 8080 that returns the current time for a city.

- 1.3. Examine the Containerfile for the `times` application.

```
[student@workstation basics-exposing]$ cat podman-info-times/Containerfile  
FROM registry.access.redhat.com/ubi8/go-toolset:1.17.7 as build  
WORKDIR /app  
USER root  
COPY app .  
RUN go build
```

```
FROM registry.access.redhat.com/ubi8/ubi-minimal:8.5
WORKDIR /app
COPY --from=build /app/times-app .
EXPOSE 8080
ENTRYPOINT ["/app/times-app"]
```

The `registry.ocp4.example.com:8443/redhattraining/podman-info-times` container image is based on this Containerfile.

- View the `podman-info-cities/app/main.go` file.

```
[student@workstation basics-exposing]$ cat podman-info-cities/app/main.go
...output omitted...
http.ListenAndServe(":8090", r)
...output omitted...
```

The application exposes an HTTP server on port 8090 that returns information for a city.

- Examine the Containerfile for the cities application.

```
[student@workstation basics-exposing]$ cat podman-info-cities/Containerfile
FROM registry.access.redhat.com/ubi8/go-toolset:1.17.7 as build
WORKDIR /app
USER root
COPY app .
RUN go build

FROM registry.access.redhat.com/ubi8/ubi-minimal:8.5
WORKDIR /app
COPY --from=build /app/cities-app .
EXPOSE 8090
ENTRYPOINT ["/app/cities-app"]
```

The `registry.ocp4.example.com:8443/redhattraining/podman-info-cities` container image is based on this Containerfile.

- Navigate to the home directory. The rest of the exercise does not involve local files.

```
[student@workstation basics-exposing]$ cd ~
no output expected
```

▶ 2. Create a Podman network with DNS enabled.

- Observe that DNS is disabled in the default Podman network.

```
[student@workstation ~]$ podman network inspect podman
...output omitted...
  "dns_enabled": false,
...output omitted...
```

- Create a network by using the `podman network create` command.

```
[student@workstation ~]$ podman network create cities
cities
```

- 2.3. Inspect the `cities` network to verify that DNS is enabled.

```
[student@workstation ~]$ podman network inspect cities
...output omitted...
  "dns_enabled": true,
...output omitted...
```

► 3. Start and test the `times` application attached to the `cities` network.

- 3.1. Create a container for the `times` application that forwards port 8080 from the container to the host. Attach the container to the `cities` network.

```
[student@workstation ~]$ podman run --name times-app \
--network cities -p 8080:8080 -d \
registry.ocp4.example.com:8443/redhattraining/podman-info-times:v0.1
...output omitted...
256...ee0
```

- 3.2. Inspect the `times-app` container to find its private IP within the Podman network and copy the IP address.

```
[student@workstation ~]$ podman inspect times-app \
-f '{{.NetworkSettings.Networks.cities.IPAddress}}'
10.89.1.2
```



Note

If the name of the network includes characters such as a dash (-), then the preceding command will not work. To retrieve the IP address of the container attached to such a network, you must use alternative methods to limit the output of the `podman inspect` command. For example, the `grep` or `jq` tools can search or parse the output, respectively.

- 3.3. Use the private IP address to make a request to the `/times` API endpoint from another container. Run the `curl` command inside a `ubi-minimal` container to fetch the current time for Bangkok, which has the code BKK. Attach the container to the `cities` network.

Replace `IP_ADDRESS` with the IP address from the previous step.

```
[student@workstation ~]$ podman run --rm --network cities \
registry.ocp4.example.com:8443/ubi8/ubi-minimal:8.5 \
curl -s http://IP_ADDRESS:8080/times/BKK && echo
2022-06-28T19:59:55.241Z
```

**Note**

Without providing arguments, the echo command prints a new line character, which improves the output formatting.

- 3.4. Use the DNS name to make a request to the /times API endpoint from another container.

```
[student@workstation ~]$ podman run --rm --network cities \
registry.ocp4.example.com:8443/ubi8/ubi-minimal:8.5 \
curl -s http://times-app:8080/times/BKK && echo
2022-06-28T20:04:37.241Z
```

Note that times-app is used as the hostname instead of the IP address.

- ▶ 4. Start and test the cities application attached to the cities network.

- 4.1. Create a container called cities-app and start the application on port 8090. Attach the container to the cities network.

Set the TIMES_APP_URL environment variable to the URL of the times application. Notice that the URL uses the name of the times-app container.

```
[student@workstation ~]$ podman run --name cities-app \
--network cities -p 8090:8090 -d \
-e TIMES_APP_URL=http://times-app:8080/times \
registry.ocp4.example.com:8443/redhattraining/podman-info-cities:v0.1
...output omitted...
dcc...1fd
```

- 4.2. Fetch the city information of Madrid by using the MAD city code.

```
[student@workstation ~]$ curl -s http://localhost:8090/cities/MAD | jq
{
  "name": "Madrid",
  "time": "2022-06-01T12:34:56.123Z",
  "population": 3223000,
  "country": "Spain"
}
```

**Note**

Passing the output of the curl command to the jq command provides better formatting of the JSON response compared to viewing the raw output.

- 4.3. Fetch the city information of San Diego by using the SAN city code.

```
[student@workstation ~]$ curl -s http://localhost:8090/cities/SAN | jq
{
  "name": "San Diego",
  "time": "2022-06-01T12:34:56.123Z",
  "population": 1415000,
  "country": "United States of America"
}
```

Finish

On the **workstation** machine, use the **lab** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish basics-exposing
```

Accessing Containers

Objectives

- Explore running containers.

Container Transparency

Developers commonly package applications as containers to, among other benefits, isolate the application process. However, process isolation also means that developers lose immediate visibility into the state of the containerized process and its environment.

To regain that visibility, containerization tools such as Podman provide a way to start new processes within containers in the running state. This is useful for example when you want to read a log file, verify the value of an environment variable, or debug a process.

An Introduction to Container Layers

Container images are characterized as *immutable* and *layered*. Each image layer consists of a set of file system differences, or *diffs*. A diff signals a file system change from the previous layer, such as adding or modifying a file.

When you start a container, the container creates a new ephemeral layer over its base container image layers called *container layer*. This layer is the only read/write storage available for the container by default, and it is used for any runtime file system operations, such as creating working files, temporary files, and log files.

Files that are created in the container layer are considered volatile, which means that the files are deleted when you delete the container. The container layer is exclusive to the running container, so if you create another container from the same base image, then the new container creates another container layer. This ensures that each container's runtime data is isolated from other containers.

Ephemeral container storage is not sufficient for applications that need to keep data beyond the life of the container, such as databases. You can use persistent storage to support such applications. Persistent container storage is covered later in this course.

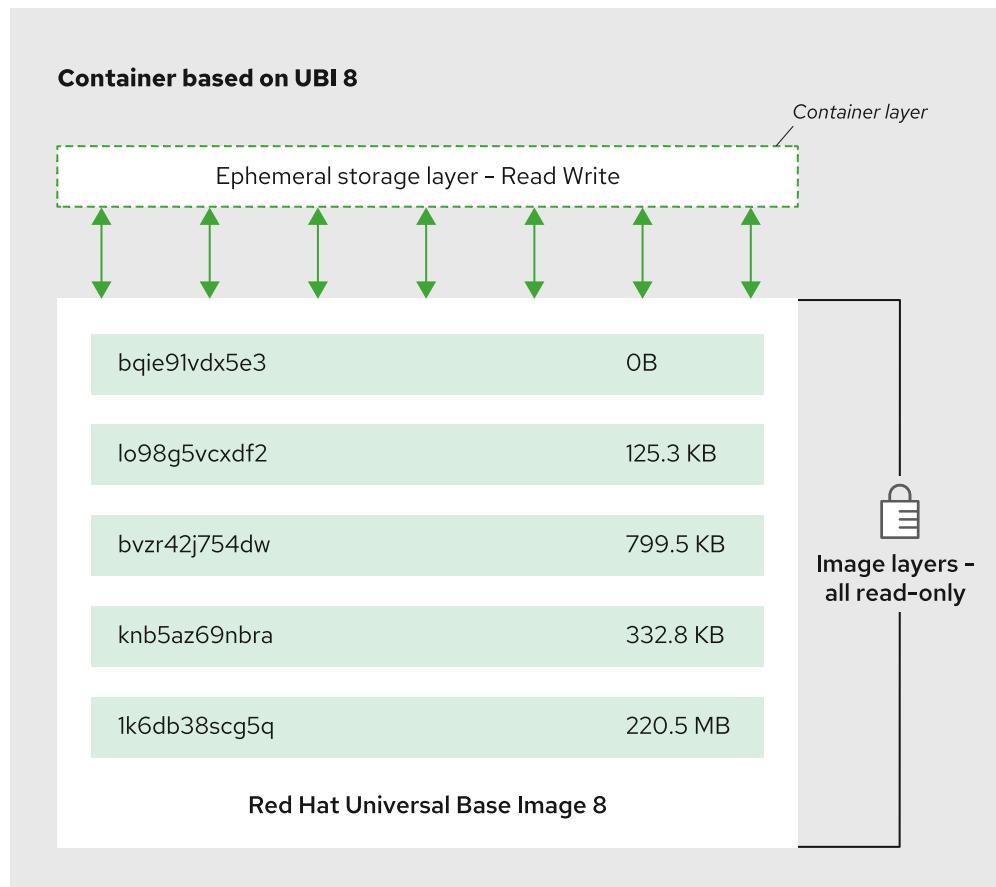


Figure 2.9: Examining example container image layers

Start Processes in Containers

Use the `podman exec` command to start a new process in a running container. The command uses the following syntax:

```
podman exec [options] container [command ...]
```



Note

Most Podman commands accept a container name or a container ID when identifying containers. You can see the IDs of all running containers by executing the `podman ps` command.

In the preceding syntax explanation, parts of the command in square brackets, such as `[options]`, are optional. For example, the following command prints the `/etc/httpd/conf/httpd.conf` file by using the `cat` command in a running container called `httpd`:

```
[user@host ~]$ podman exec httpd cat /etc/httpd/conf/httpd.conf
```

Additionally, `podman exec` provides a number of options, such as:

- Use `--env` or `-e` to specify environment variables.

- Use `--interactive` or `-i` to instruct the container to accept input.
- Use `--tty` or `-t` to allocate a pseudo terminal.
- Use `--latest` or `-l` to execute the command in the last created container.



Note

The `--latest` and `-l` flags are not available when using the Podman remote client. This includes when running via Podman Machine on macOS and Windows, except when using WSL2 on the latter.

The following command sets the `ENVIRONMENT` variable and then executes the `env` command to print all environment variables. In this example, the container name is not necessary because the `-l` option is used

```
[user@host ~]$ podman exec -e ENVIRONMENT=dev -l env
```

After the `env` process finishes, the `ENVIRONMENT` variable is unset. To make the `ENVIRONMENT` variable persistent, stop and remove the running container, and rerun the `podman run` command with the `-e ENVIRONMENT=dev` option.

Open an Interactive Session in Containers

Use the combination of the `--tty` and `--interactive` options to open an interactive shell in a running container.

If you open a shell program such as Bash or PowerShell in a container without providing any options, then the `podman exec` command opens the shell program, receives no input, and exits successfully:

```
[user@host ~]$ podman exec -l /bin/bash  
[user@host ~]$
```

If you open a shell program with the `--interactive` option, the `podman exec` executes the shell program and commands, but the session does not behave like a regular terminal.

For example, features such as command history are not available in this mode and you cannot use programs that require a TTY such as vim. Although command output is visible, it is hard to differentiate from the input. Only use this mode when you do not need a full terminal experience.

The following example shows such an interactive terminal session:

```
[user@host ~]$ podman exec -il /bin/bash  
pwd  
/opt/app-root/src  
ls ../  
etc  
scl_enable  
src
```

**Note**

The preceding example combines multiple options. The `podman exec -il` command is identical to `podman exec -i -l`.

If you open a shell program with the `--tty` option, the `podman exec` executes the shell program and opens a pseudo terminal, but receives no input:

```
[user@host ~]$ podman exec -tl /bin/bash
bash-4.4$ ①
```

- ① The shell program opens in a pseudo terminal, but you cannot pass any input to the shell.

To open an interactive shell in a running container, use the `--interactive` option. Also include the `--tty` option to make the session behave like a regular terminal session. Use the combination of both options to open a remote session in the container and have it behave in a typical way, such as in the following example:

```
[user@host ~]$ podman exec -til /bin/bash
bash-4.4$ pwd ①
/opt/app-root/src
bash-4.4$ ls ../
etc scl_enable src
bash-4.4$ exit ②
[user@host ~]$ ③
```

- ① Execute the `pwd` command to print the current working directory.
- ② Execute the `exit` command to exit the interactive session.
- ③ The container exits successfully.

Copy Files In and Out of Containers

Some containers do not provide the programs necessary to debug a running process. For example, to print the content of a file, you might use the `cat` utility. However, in production-ready containers, a best practice is to package only the libraries required by the application runtime. Such a container might not include basic utilities, such as `cat`, or an editor such as `vi`.

You can work around the issue in several ways, one of which is copying the file you want to modify to your host machine, modifying it, and then replacing the original file.

Use the `podman cp` command to copy files to and from a running container. The command uses the following syntax:

```
podman cp [options] [container:]source_path [container:]destination_path
```

Use the following command to copy the `/tmp/logs` file from a container with ID `a3bd6c81092e` to the current directory:

```
[user@host ~]$ podman cp a3bd6c81092e:/tmp/logs .
```

**Note**

The dot (.) in the previous command signifies the current working directory.

Use the following command to copy the `nginx.conf` file to the `/etc/nginx/` directory in a container called `nginx`:

```
[user@host ~]$ podman cp nginx.conf nginx:/etc/nginx
```

The preceding command assumes that the `nginx.conf` file exists in your working directory.

Use the following command to copy the `nginx.conf` file from the `nginx-test` container to the `nginx-proxy` container:

```
[user@host ~]$ podman cp nginx-test:/etc/nginx/nginx.conf nginx-proxy:/etc/nginx
```

**References**

[podman-exec\(1\) man page](#)

[podman-cp\(1\) man page](#)

► Guided Exercise

Accessing Containers

Use the podman exec and podman cp commands to debug and correct container configuration.

Outcomes

You should be able to:

- Use the podman exec to execute commands inside of a container.
- Use the podman cp command to copy files from and into a container.

Before You Begin

As the student user on the workstation machine, use the lab command to prepare your system for this exercise.

```
[student@workstation ~]$ lab start basics-accessing
```

Instructions

- 1. Create a new container with the following parameters:

- Container name: nginx
- Container image: registry.ocp4.example.com:8443/redhattraining/podman-nginx-helloworld

Route traffic from port 8080 on your machine to port 8080 inside of the container. Use the -d option to run the container in the background.

```
[student@workstation ~]$ podman run --name nginx -d -p 8080:8080 \
  registry.ocp4.example.com:8443/redhattraining/podman-nginx-helloworld
Trying to pull registry.ocp4.example.com:8443/redhattraining/podman-nginx-
helloworld:latest...
...output omitted...
3c2b...fa84
```

- 2. In a web browser, navigate to localhost:8080. You are presented with a 404 Not Found error page.

- 3. Troubleshoot the issue.

- 3.1. Use the podman cp command to copy the /var/log/nginx/error.log log file in the nginx container to your local machine.

In your terminal, execute the following command:

Chapter 2 | Podman Basics

```
[student@workstation ~]$ podman cp nginx:/var/log/nginx/error.log error.log
no output expected
```

- 3.2. View the contents of the log file.

```
[student@workstation ~]$ gedit error.log
2022/04/26 12:19:17 [error] 2#0: *2 "/usr/share/nginx/html/public/index.html" is
not found (2: No such file or directory), client: 10.0.2.100, server: _, request:
"GET / HTTP/1.1", host: "localhost:8080"
2022/04/26 12:19:17 [error] 2#0: *2 open() "/usr/share/nginx/html/public/404.html"
failed (2: No such file or directory), client: 10.0.2.100, server: _, request:
"GET / HTTP/1.1", host: "localhost:8080"
```

The server cannot read the `/usr/share/nginx/html/public/index.html` file.

- 3.3. Verify the contents of the `/usr/share/nginx/html/public` directory.

```
[student@workstation ~]$ podman exec nginx ls /usr/share/nginx/html/public
ls: cannot access '/usr/share/nginx/html/public': No such file or directory
```

The directory does not exist in the container.

- 3.4. Verify the contents of the `/usr/share/nginx/html` directory.

```
[student@workstation ~]$ podman exec nginx ls /usr/share/nginx/html
404.html
50x.html
index.html
nginx-logo.png
poweredby.png
```

The `index.html` page exists in the `/usr/share/nginx/html` directory.

▶ **4.** Correct the server configuration.

- 4.1. Copy the `/etc/nginx/nginx.conf` file from the container to your machine.

```
[student@workstation ~]$ podman cp nginx:/etc/nginx/nginx.conf .
no output expected
```

- 4.2. Open the `nginx.conf` file in a text editor, such as VSCode or Gedit.

In the `server` directive, change the `root` parameter to the `/usr/share/nginx/html/` value.

```
[student@workstation ~]$ gedit nginx.conf
...file omitted...
server {
    listen      8080 default_server;
    server_name _;
    root        /usr/share/nginx/html/;
...file omitted...
```

Save the change in the `nginx.conf` file.

- 4.3. Replace the `/etc/nginx/nginx.conf` file in the container with the modified `nginx.conf` file.

```
[student@workstation ~]$ podman cp nginx.conf nginx:/etc/nginx  
no output expected
```



Note

You can also use the `podman exec` command with the `--interactive` and `--tty` options to open an interactive shell to modify files directly in a running container. For example, you can use the `podman exec -ti nginx vim /etc/nginx/nginx.conf` command to interactively modify the `nginx.conf` file with the `vim` text editor.

Keep in mind that changes applied to a running container are temporary and do not persist if you delete the container. For persistent changes, you must rebuild the container image. Rebuilding container images is covered elsewhere in this course.

- 5. Verify the functionality of the running container.

- 5.1. Reload the server configuration.

```
[student@workstation ~]$ podman exec nginx nginx -s reload  
no output expected
```

- 5.2. In a web browser, navigate to `localhost:8080`. You are presented with the `index.html` page contents.

Finish

On the `workstation` machine, use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish basics-accessing
```

Managing the Container Lifecycle

Objectives

- List, stop, and delete containers with Podman.

Container Lifecycle

Podman provides a set of subcommands to create and manage containers. You can use those subcommands to manage the container and container image lifecycle.

The following figure shows a summary of the most commonly used Podman subcommands that change the container and image state:

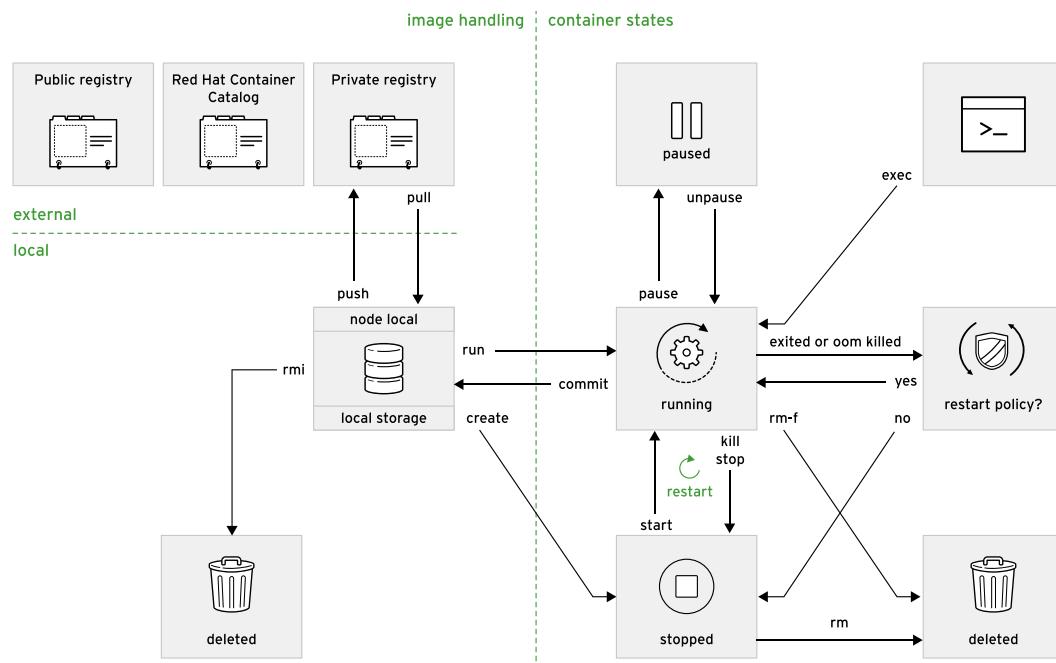


Figure 2.10: Podman lifecycle commands

Podman also provides a set of subcommands to obtain information about running and stopped containers.

You can use these subcommands to extract information from containers and images for debugging, updating, or reporting purposes. The following figure shows a summary of the most commonly used subcommands that query information from containers and images:

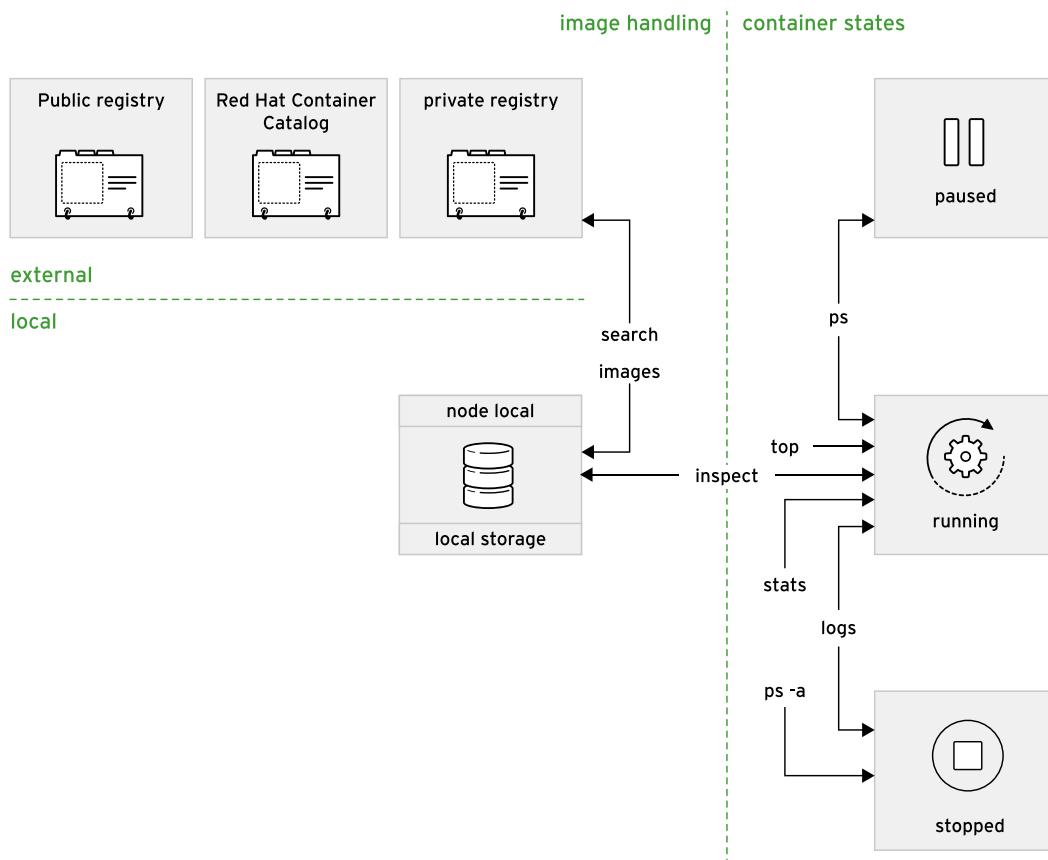


Figure 2.11: Podman query commands

This lecture covers the basic operations that you can use to manage containers. Commands explained in this lecture accept either a container ID or the container name.

Listings Containers

You can list running containers with the `podman ps` command.

```
[user@host ~]$ podman ps
CONTAINER ID  IMAGE          COMMAND           CREATED        STATUS    PORTS     NAMES
0ae7be593698  ...server     /bin/sh -c python...  ...ago         Up...     ...8000/tcp httpd
c42e7dca12d9  ...helloworld /bin/sh -c nginx...  ...ago         Up...     ...8080/tcp nginx
```

Each row describes information about the container, such as the image used to start the container, the command executed when the container started, and the container uptime.

You can include stopped containers in the output by adding the `--all` or `-a` flag to the `podman ps` command.

```
[user@host ~]$ podman ps --all
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS
NAMES
0ae7be593698 ...server /bin/sh -c python... ...ago Up... ...8000/tcp
httpd
bd5ada1b6321 ...httpd-24 /usr/bin/run-http... ...ago Exited... ...8080/tcp
upbeat...
c42e7dca12d9 ...helloworld /bin/sh -c nginx ... ...ago Up... ...8080/tcp
nginx
```

Inspecting Containers

In the context of Podman, inspecting a container means retrieving the full information of the container. The `podman inspect` command returns a JSON array with information about different aspects of the container, such as networking settings, CPU usage, environment variables, status, port mapping, or volumes. The following snippet is a sample output of the command.

```
[user@host ~]$ podman inspect 7763097d11ab
[
  {
    "Id": "7763...cbc0",
    "Created": "2022-05-04T10:00:32.988377257-03:00",
    "Path": "container-entrypoint",
    "Args": [
      "/usr/bin/run-httpd"
    ],
    "State": {
      "OciVersion": "1.0.2-dev",
      "Status": "running",
      "Running": true,
      "Paused": false,
      "Restarting": false,
      "OOMKilled": false,
      "Dead": false,
      "Pid": 9746,
      "...output omitted...
      "Image": "d2b9...fa0a",
      "ImageName": "registry.access.redhat.com/ubi8/httpd-24:latest",
      "Rootfs": "",
      "...output omitted...
      "Env": [
        "PATH=/opt/app-root/src/bin:/opt/app-root/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
        "TERM=xterm",
        "container=oci",
        "HTTPD_VERSION=2.4",
      ],
      "...output omitted...
      "CpuCount": 0,
      "CpuPercent": 0,
      "IOMaximumIOps": 0,
      "IOMaximumBandwidth": 0,
      "CgroupConf": null
    }
  }
]
```

```

    }
}
]
```

The previous JSON output omits most fields. Because the full JSON is long, you might find it difficult to find the specific fields that you want to retrieve. You can pass a `--format` flag to the `podman inspect` command to filter the command output. The `--format` flag expects a Go template expression as a parameter, which you can use to select specific fields in the JSON.

Go template expressions use annotations, which are delimited by double curly braces (`{}{}`), to refer to elements, such as fields or keys, in a data structure. The data structure elements are separated by a dot (`.`) and must start in upper case.

In the following command, the `Status` field of the `State` object is retrieved for a container called `redhat`.

```
[user@host ~]$ podman inspect \
--format='{{.State.Status}}' redhat
running
```

In the preceding example, the `podman inspect` command uses the supplied Go template to navigate the JSON array of the `redhat` container. The template returns the `Status` field value of the `State` object from the JSON array.

Refer to the references section for more information about the Go templating language.

Stopping Containers Gracefully

You can stop a container gracefully by using the `podman stop` command. When you execute the `podman stop` command, Podman sends a `SIGTERM` signal to the container. Processes use the `SIGTERM` signal to implement clean-up procedures before stopping.

The following command stops a container with a container ID `1b982aeb75dd`.

```
[user@host ~]$ podman stop 1b982aeb75dd
1b982aeb75dd
```

You can stop all the running containers by using the `--all` or `-a` flag. In the following example, the command stops three containers.

```
[user@host ~]$ podman stop --all
4aea...0c2a
6b18...54ea
7763...cbc0
```

If a container does not respond to the `SIGTERM` signal, then Podman sends a `SIGKILL` signal to forcefully stop the container. Podman waits 10 seconds by default before sending the `SIGKILL` signal. You can change the default behavior by using the `--time` flag.

```
[user@host ~]$ podman stop --time=100
```

In the previous example, Podman gives the container a grace period of 100 seconds before sending the killing signal.

Chapter 2 | Podman Basics

Stopping a container differs from removing a container. Stopped containers are present on the host machine and can be listed by using the `podman ps --all` command.

Additionally, when the containerized process finishes, the container enters the exited state. Such a process might finish for a number of reasons, such as an error, OOM (out-of-memory) state, or successfully finishing. Podman lists exited containers with other stopped containers.

Stopping Containers Forcefully

You can send the **SIGKILL** signal to the container by using the `podman kill` command. In the following example, a container called `httpd` is stopped forcefully.

```
[user@host ~]$ podman kill httpd
httpd
```

Pausing Containers

Both `podman stop` and `podman kill` commands eventually send a **SIGKILL** signal to the container. The `podman pause` command suspends all processes in the container by sending the **SIGSTOP** signal.

```
[user@host ~]$ podman pause 4f2038c05b8c
4f2038c05b8c
```

The `podman unpause` command resumes a paused container.

```
[user@host ~]$ podman unpause 4f2038c05b8c
4f2038c05b8c
```

Restarting Containers

Execute the `podman restart` command to restart a running container. You can also use the `podman start` command to start stopped containers.

The following command restarts a container called `nginx`.

```
[user@host ~]$ podman restart nginx
1b98...75dd
```

Removing Containers

Use the `podman rm` command to remove a stopped container. The following command removes a stopped container with the container ID `c58cf4b90df`.

```
[user@host ~]$ podman rm c58cf4b90df
c58c...3150
```

You can not remove running containers by default. You must stop the running container first and then remove it. The following command tries to remove a running container.

```
[user@host ~]$ podman rm c58cf4b90df
Error: cannot remove container c58c...3150 as it is running - running or paused
containers cannot be removed without force: container state improper
```

You can add the `--force` (or `-f`) flag to remove the container forcefully.

```
[user@host ~]$ podman rm c58cf4b90df --force
c58c...3150
```

You can also add the `--all` (or `-a`) flag to remove all stopped containers. This flag fails to remove running containers. The following command removes two containers.

```
[user@host ~]$ podman rm --all
6b18...54ea
6c0d...a6fb
```

You can combine the `--force` and `--all` flags to remove all containers, including running containers.

Start a Containerized Service on Boot

In a traditional environment, administrators configure applications such as web servers or databases to start at boot time, and run indefinitely as a systemd service. Systemd is a system and service management tool for Linux operating systems. Systemd uses service unit files to start and stop applications, or to enable them to start at boot time. Typically, an administrator manages these applications with the `systemctl` command. As a regular user, you can create systemd unit files to manage your rootless containers. You can use this configuration to manage your container as a regular systemd service with the `systemctl` command.

Generate a Systemd Unit File

To create a systemd unit file for a specified service container, use the `podman generate systemd` command. Use the `--name` option to specify the container's name. The `--files` option generates files instead of printing to standard output (STDOUT).

```
[user@host ~]$ podman generate systemd --name web --files
/home/user/container-web.service
```

The preceding command creates the `container-web.service` unit file for a container called `web`. After reviewing and adapting the generated service configuration to your requirements, store the unit file in the user's systemd configuration directory (`~/.config/systemd/user/`).

After adding or modifying unit files, you must use the `systemctl` command to reload the systemd configuration.

```
[user@host ~]$ systemctl --user daemon-reload
```

Managing the Containerized Service

To manage a containerized service, use the `systemctl` command.

```
[user@host ~]$ systemctl --user [start, stop, status, enable, disable]  
container-web.service
```

When you use the `--user` option, by default, systemd starts the service at your login, and stops it at your logout. You can start your enabled services at the operating system boot, and stop them on shutdown, by running the `logindctl enable-linger` command.

```
[user@host ~]$ logindctl enable-linger
```

To revert the operation, use the `logindctl disable-linger` command.



References

- [podman-inspect\(1\) man page](#)
- [podman-stop\(1\) man page](#)
- [podman-restart\(1\) man page](#)
- [podman-rm\(1\) man page](#)

Go templates package

<https://pkg.go.dev/text/template>

For more information, refer to the *Running Containers as Systemd Services with Podman* chapter in the *Red Hat Enterprise Linux 9 Building, Running, and Managing Containers* guide at

https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/9/html-single/building_running_and_managing_containers/index

► Guided Exercise

Managing the Container Lifecycle

Manage the lifecycle of a container that runs an Apache HTTP server.

Outcomes

You should be able to:

- Get detailed information about a container.
- Stop containers.
- Restart a stopped container.
- Delete containers.

Before You Begin

As the student user on the workstation machine, use the `lab` command to prepare your system for this exercise.

```
[student@workstation ~]$ lab start basics-lifecycle
```

Instructions

- 1. Create a container that runs an Apache HTTP server in the background.
- 1.1. Execute the `podman run` command to create the container. Expose the `8080` port and use the `registry.ocp4.example.com:8443/ubi8/httpd-24` image.

```
[student@workstation ~]$ podman run --name httpd -d -p \
8080:8080 registry.ocp4.example.com:8443/ubi8/httpd-24
Trying to pull registry.ocp4.example.com:8443/ubi8/httpd-24:latest...
...output omitted...
```

- 2. Verify that the container is running.
- 2.1. Use `podman ps` to list all the running containers.
- | CONTAINER ID | IMAGE | COMMAND | CREATED | STATUS | PORTS | NAMES |
|--------------|-------------|----------------|---------|--------|-------------|--------------|
| ID | registry... | ...run-http... | ...ago | Up... | ...8080/tcp | httpd |
- 2.2. Use the `podman inspect` command to get the `Status` field, which indicates whether the container is running.

```
[student@workstation ~]$ podman inspect --format='{{.State.Status}}' httpd
running
```

Chapter 2 | Podman Basics

- 2.3. Verify that the container is running by using the `Running` field.

```
[student@workstation ~]$ podman inspect --format='{{.State.Running}}' httpd
true
```

- 2.4. In a web browser, navigate to `localhost:8080` to verify that the Apache server is running.

► **3.** Stop the container.

- 3.1. Return to your command-line terminal. Use the container name to stop the container.

```
[student@workstation ~]$ podman stop httpd
httpd
```

- 3.2. Retrieve the `Status` of the container.

```
[student@workstation ~]$ podman inspect --format='{{.State.Status}}' httpd
exited
```

- 3.3. Verify that the container is not running.

```
[student@workstation ~]$ podman inspect --format='{{.State.Running}}' httpd
false
```

- 3.4. In a web browser, verify that the Apache server is no longer accessible at the 8080 port.

► **4.** Restart the container.

- 4.1. In your command-line terminal, use the `podman restart` command to restart the container.

```
[student@workstation ~]$ podman restart httpd
CONTAINER_ID
```

- 4.2. Verify that the container is running again.

```
[student@workstation ~]$ podman ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS              NAMES
ID                  registry...    ...run-http...   ...ago   Up...    ...8080/tcp httpd
```

- 4.3. In a web browser, verify that the Apache server is running by navigating to `http://localhost:8080`.

► **5.** Remove the container.

- 5.1. Try to remove the running container by using the `podman rm` command.

```
[student@workstation ~]$ podman rm httpd
Error: cannot remove container CONTAINER_ID as it is running - running or paused
containers cannot be removed without force: container state improper
```

Podman requires that you stop the container before you can remove it. However, you can force the removal of the container by adding the `--force` flag.

- 5.2. Force the removal of the container.

```
[student@workstation ~]$ podman rm httpd --force
78f3...bc672
```

- ▶ 6. Verify that the container has been removed.

- 6.1. Try to retrieve the status of the container.

```
[student@workstation ~]$ podman inspect --format='{{.State.Running}}' httpd
Error: error inspecting object: no such object: "httpd"
```

Because you removed the container, you can no longer retrieve its status.

- ▶ 7. Forcefully stop a container that does not respond to the SIGTERM signal.

- 7.1. Start a new container in the background by using the `registry.ocp4.example.com:8443/redhattraining/podman-greeter-ignore-sigterm` image. The application ignores SIGTERM signals.

```
[student@workstation ~]$ podman run --name greeter -d \
registry.ocp4.example.com:8443/redhattraining/podman-greeter-ignore-sigterm
...output omitted...
f919...e69b
```

- 7.2. Try to stop the container with a grace period of 5 seconds.

```
[student@workstation ~]$ podman stop greeter --time=5
WARN[0005] StopSignal SIGTERM failed to stop container greeter in 5
seconds, resorting to SIGKILL
greeter
```

Podman sends a SIGTERM signal to stop the container gracefully, but the application ignores the signal. After 5 seconds, Podman sends a SIGKILL signal to the container. The `--time` flag indicates the time that Podman waits before sending a SIGKILL signal to forcefully stop the container.

- 7.3. Restart the container.

```
[student@workstation ~]$ podman restart greeter
f919...e69b
```

- 7.4. Use the `podman kill` command to directly send a SIGKILL signal and stop the container forcefully.

```
[student@workstation ~]$ podman kill greeter  
greeter
```

Finish

On the **workstation** machine, use the **lab** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish basics-lifecycle
```

▶ Lab

Podman Basics

Use Podman to manage local containers.

Outcomes

You should be able to:

- Manage local containers.
- Copy files in and out of containers.
- Run a set of application containers that connect to one another via a Podman network.
- Forward a port from a container so that it is accessible from the host machine.

Before You Begin

As the student user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command starts the `basics-podman-secret` local container, and copies the necessary files for this lab. The command also verifies that Podman is available and can pull from the required registries. You can find the source code for the `basics-podman-secret` container in the `$HOME/D0188/solutions/basics-podman/secret-container` directory.

The lab script continuously evaluates the objectives of this lab. Keep the script running in a terminal window, and complete the objectives of this lab from a new terminal window.

After each objective, return to the lab script evaluation to see if you have finished the objective successfully. When you have finished all of the objectives, the `lab` command prompts you to execute the `finish` function.

```
[student@workstation ~]$ lab start basics-podman
```

Instructions

1. The lab command starts the `basics-podman-secret` container, which contains the `/etc/secret-file` file.
Copy the `/etc/secret-file` file from the container to the `$HOME/D0188/labs/basics-podman/solution` file.
2. Start a new container with the following parameters:
 - Name: `basics-podman-server`
 - Image: `registry.ocp4.example.com:8443/ubi8/httpd-24`
 - Ports: Route traffic from port `8080` on your machine to port `8080` inside of the container
 - Network: `lab-net`

You can start the container in the detached mode for greater convenience.

3. Copy the \$HOME/DO188/labs/basics-podman/index.html file to /var/www/html/ in the basics-podman-server container.
4. Start a new container with the following parameters:
 - Name: basics-podman-client
 - Image: registry.ocp4.example.com:8443/ubi8/httpd-24
 - Network: lab-net

You can start the container in the detached mode for greater convenience.

5. Confirm that the basics-podman-client container can access the basics-podman-server container by its DNS name. Use the podman exec and curl commands to make a request to the basics-podman-server container at port 8080 from the basics-podman-client container.

Finish

As the student user on the workstation machine, change to the student user home directory and use the lab command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

Press y when the lab start command prompts you to execute the finish function.
Alternatively, execute the following command:

```
[student@workstation ~]$ lab finish basics-podman
```

► Solution

Podman Basics

Use Podman to manage local containers.

Outcomes

You should be able to:

- Manage local containers.
- Copy files in and out of containers.
- Run a set of application containers that connect to one another via a Podman network.
- Forward a port from a container so that it is accessible from the host machine.

Before You Begin

As the `student` user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command starts the `basics-podman-secret` local container, and copies the necessary files for this lab. The command also verifies that Podman is available and can pull from the required registries. You can find the source code for the `basics-podman-secret` container in the `$HOME/D0188/solutions/basics-podman/secret-container` directory.

The lab script continuously evaluates the objectives of this lab. Keep the script running in a terminal window, and complete the objectives of this lab from a new terminal window.

After each objective, return to the lab script evaluation to see if you have finished the objective successfully. When you have finished all of the objectives, the `lab` command prompts you to execute the `finish` function.

```
[student@workstation ~]$ lab start basics-podman
```

Instructions

1. The `lab` command starts the `basics-podman-secret` container, which contains the `/etc/secret-file` file.
Copy the `/etc/secret-file` file from the container to the `$HOME/D0188/labs/basics-podman/solution` file.
 - 1.1. Use the `podman ps` command to verify that the `basics-podman-secret` container is running.

```
[student@workstation ~]$ podman ps --format='{{.Names}}'  
basics-podman-secret
```

Chapter 2 | Podman Basics

- 1.2. Use the `ls` command inside of the container to verify that the container contains the `/etc/secret-file` file.

```
[student@workstation ~]$ podman exec basics-podman-secret ls /etc/secret-file  
/etc/secret-file
```

- 1.3. Change into the `$HOME/D0188/labs/basics-podman` directory, for example:

```
[student@workstation ~]$ cd ~/D0188/labs/basics-podman  
[student@workstation basics-podman]$ ls  
index.html
```

- 1.4. Copy the `/etc/secret-file` file from the container as the `solution` file in the `$HOME/D0188/labs/basics-podman` directory.

```
[student@workstation basics-podman]$ podman cp \  
basics-podman-secret:/etc/secret-file solution  
[student@workstation basics-podman]$ ls  
index.html solution
```

The first objective of the lab script is passing.

2. Start a new container with the following parameters:

- Name: `basics-podman-server`
- Image: `registry.ocp4.example.com:8443/ubi8/httpd-24`
- Ports: Route traffic from port `8080` on your machine to port `8080` inside of the container
- Network: `lab-net`

You can start the container in the detached mode for greater convenience.

- 2.1. Create the `lab-net` Podman network.

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

The next objective of the lab script is passing.

- 2.2. Execute the `podman run` command to start the container.

```
[student@workstation basics-podman]$ podman run -d --name basics-podman-server \  
--net lab-net -p 8080:8080 registry.ocp4.example.com:8443/ubi8/httpd-24  
8b747...3616
```

Additional objectives of the lab script are passing.

3. Copy the `$HOME/D0188/labs/basics-podman/index.html` file to `/var/www/html/` in the `basics-podman-server` container.

- 3.1. Verify that you are in the correct directory.

Chapter 2 | Podman Basics

```
[student@workstation basics-podman]$ ls  
index.html solution
```

3.2. Copy the `index.html` file in to the container.

```
[student@workstation basics-podman]$ podman cp index.html \  
basics-podman-server:/var/www/html/
```

3.3. In a web browser, navigate to `localhost:8080` and verify that you see the `Hello from Podman Basics` lab text.

The next objective of the lab script is passing.

4. Start a new container with the following parameters:

- Name: `basics-podman-client`
- Image: `registry.ocp4.example.com:8443/ubi8/httpd-24`
- Network: `lab-net`

You can start the container in the detached mode for greater convenience.

4.1. Execute the `podman run` command to start the container.

```
[student@workstation basics-podman]$ podman run -d --name basics-podman-client \  
--net lab-net registry.ocp4.example.com:8443/ubi8/httpd-24  
8b747...3616
```

5. Confirm that the `basics-podman-client` container can access the `basics-podman-server` container by its DNS name. Use the `podman exec` and `curl` commands to make a request to the `basics-podman-server` container at port `8080` from the `basics-podman-client` container.

5.1. Confirm that DNS is enabled on the `lab-net` network.

```
[student@workstation basics-podman]$ podman network inspect lab-net  
...output omitted...  
    "dns_enabled": true,  
...output omitted...
```

5.2. Use the `curl` command inside of the `basics-podman-client` to confirm that the `basics-podman-server` container is accessible from the `lab-net` network by its DNS name.

```
[student@workstation basics-podman]$ podman exec basics-podman-client \  
curl -s http://basics-podman-server:8080 && echo  
<h1>Hello from Podman Basics lab</h1>
```

All remaining objectives are passing.

Finish

As the **student** user on the **workstation** machine, change to the **student** user home directory and use the **lab** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

Press **y** when the **lab start** command prompts you to execute the **finish** function.
Alternatively, execute the following command:

```
[student@workstation ~]$ lab finish basics-podman
```

Summary

- Use the `podman run` command to start a container.
- Implement container-to-container communication by using Podman networks, which include the following tasks:
 - Managing Podman networks by using `podman network` subcommands.
 - Attaching a container to a network.
- Use port-forwarding to expose a containerized process to the host environment.
- Use the `podman stop` or `podman kill` commands to stop a container.
- Use the `podman ps` command to list containers.
- Use the `podman cp` command to copy files into and out of containers.
- Manage the lifecycle of containers by using the `podman start`, `podman pause`, `podman restart`, and other Podman commands.

Chapter 3

Container Images

Goal

Navigate container registries to find and manage container images.

Objectives

- Navigate container registries.
- Pull and manage container images.

Sections

- Container Image Registries (and Guided Exercise)
- Managing Images (and Guided Exercise)

Lab

- Container Images

Container Image Registries

Objectives

- Navigate container registries.

Container Registries

A container image is a packaged version of your application, with all the dependencies necessary for the application to run. You can use image registries to store container images to later share them in a controlled manner. Some examples of image registries include Quay.io, Red Hat Registry, Docker Hub, or Amazon ECR.

For example, consider the following Podman command.

```
[user@host ~]$ podman pull registry.redhat.io/ubi8/ubi:8.6
Trying to pull registry.redhat.io/ubi8/ubi:8.6...
Getting image source signatures
...output omitted...
Writing manifest to image destination
Storing signatures
3434...8f6b
```

The `registry.redhat.io/ubi8` image is stored in the Red Hat Registry, because the container name uses the `registry.redhat.io` host.

Red Hat Registry

Red Hat distributes container images by using two registries:

- `registry.access.redhat.com`: requires no authentication
- `registry.redhat.io`: requires authentication

However, Red Hat provides a centralized searching utility for both registries: the Red Hat Ecosystem Catalog, available at <https://catalog.redhat.com/>. You can use the Red Hat Ecosystem Catalog to search for images and get technical details about them. Navigate to <https://catalog.redhat.com/software/containers/explore> to search for container images.

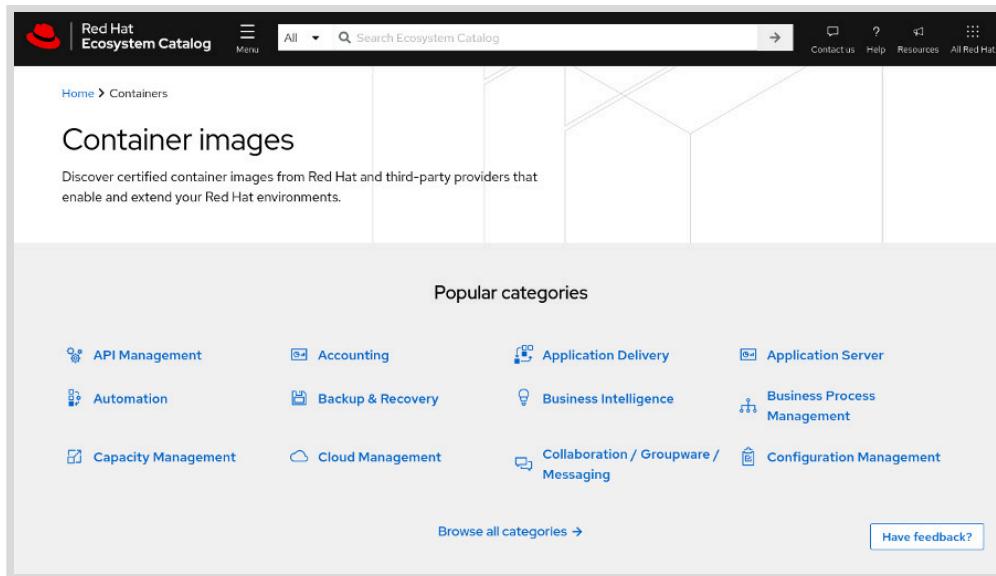


Figure 3.1: The Red Hat Ecosystem Catalog

The container image details page gives you relevant information about the container image, such as the Containerfile used to create the image, the packages installed within the image, or a security scanning. You can also change the image version by selecting a specific tag.

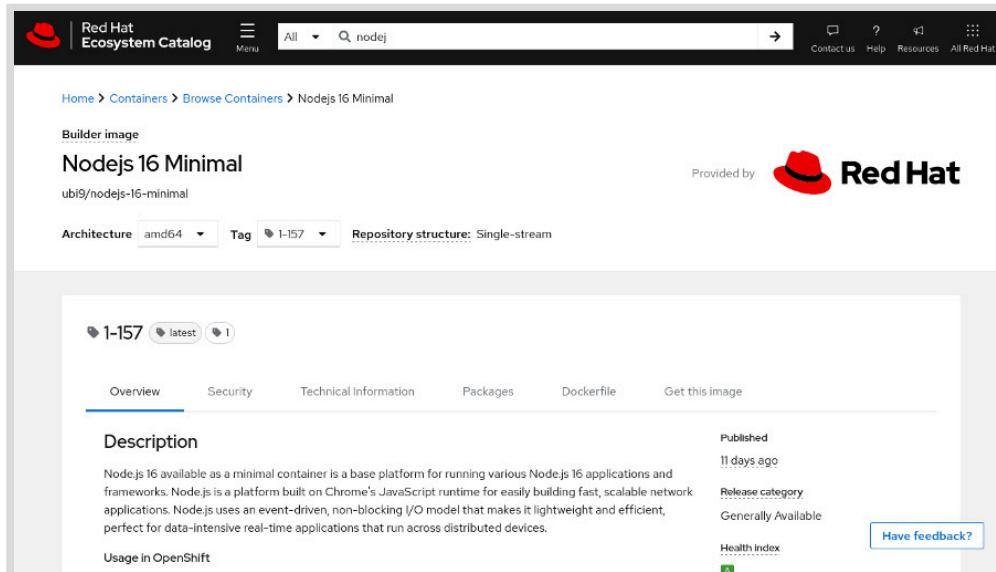


Figure 3.2: The Node.js 16 image based on RHEL 8

Useful Container Images

The following table introduces some useful container images.

Image	Provides	Description
<code>registry.access.redhat.com/ubi9</code>	Universal Base Image (UBI), Version 9	A base image to create other images that is based on RHEL 9.

Image	Provides	Description
registry.access.redhat.com/ubi9/python-39	Python 3.9	A UBI-based image with the Python 3.9 runtime.
registry.access.redhat.com/ubi9/nodejs-18	Node.js 18	A UBI-based image with the Node.js 18 runtime.
registry.access.redhat.com/ubi9/go-toolset	Go Toolset	A UBI-based image with the Go runtime. The Go version depends on the image tag.

Red Hat UBIs are *Open Container Initiative (OCI)* compliant enterprise grade container images that provide the base operating system layer for your containerized applications. UBIs include a subset of Red Hat Enterprise Linux (RHEL) components. UBIs can additionally provide a set of pre-built language runtimes. UBIs are freely distributable, and you can use UBIs on both Red Hat and non-Red Hat platforms or container registries.

You do not need a Red Hat subscription to use or distribute UBI-based images. However, Red Hat only provides full support for containers built on UBI if the containers are deployed to a Red Hat platform, such as Red Hat OpenShift Container Platform (RHOC) or RHEL.

Quay.io

Although the Red Hat Registry only stores images from Red Hat and certified providers, you can use the Quay.io registry to store your custom images. Storing public images in Quay.io is free, and paying customers receive further benefits, such as private repositories. Developers can also deploy an on-premise Quay instance, which you can use to set up an image registry on your infrastructure.

To log in to Quay.io, you can use your Red Hat developer account.

Figure 3.3: The Quay.io welcome page

Manage Registries with Podman

When you pull a container image, you provide a number of details. For example, the `registry.access.redhat.com/ubi9/nodejs-18:latest` image name consists of the following information:

- Registry URL: `registry.access.redhat.com`
- User or organization: `ubi9`
- Image repository: `nodejs-18`
- Image tag: `latest`

Developers can specify a shorter, unqualified name, which omits the registry URL. For example, you might shorten the `registry.redhat.io/rhel7/rhel:7.9` to `rhel7/rhel:7.9`.

```
[user@host ~]$ podman pull ubi8/python-39
```

If you do not provide the registry URL, then Podman uses the `/etc/containers/registries.conf` file to search other container registries that might contain the image name. This file contains registries that Podman searches to find the image, in order of preference.

For example, with the following configuration, Podman searches the Red Hat Registry first. If the image is not found in the Red Hat Registry, then Podman searches in the Docker Hub registry.

```
unqualified-search-registries == ['registry.redhat.io', 'docker.io']
```

You can also block a registry. For example, the following configuration blocks pulling from the Docker Hub.

```
[[registry]]
location="docker.io"
blocked=true
```

Some systems do not have a `/etc/containers/registries.conf` file, such as Microsoft Windows. In such cases, the `registries.conf` file might exist in a different location.

For example, on Microsoft Windows, execute `podman machine ssh` to connect to the Linux-based virtual machine that starts your containers. In the virtual machine, you can find the `/etc/containers/registries.conf` file:

```
[user@host ~]$ podman machine ssh
[user@DESKTOP-AA1A111 ~]$ ls /etc/containers/containers.conf
/etc/containers/containers.conf
```

See the references section for more details about the `registries.conf` file.

Red Hat recommends that you always use a fully qualified container image name to avoid duplicate container images in multiple container registries. For example, depending on your Podman configuration, the `rhel7/rhel:7.9` container image might resolve to a potentially unsupported or malicious `docker.io/library/rhel7/rhel:7:9` container image.

Manage Registries with Skopeo

Skopeo is a command-line tool for working with container images. Developers can use Skopeo in a number of ways, for example:

- Inspect remote container images.
- Copy a container image between registries.
- Sign an image with OpenPGP keys.
- Convert image format, for example from Docker to the OCI format.

Skopeo can inspect remote images or transfer images between registries without using local storage. The skopeo command uses the `transport:image` format, such as `docker://remote_image`, `dir:path`, or `oci:path:tag`.

Use the `skopeo inspect` command to read image metadata.

```
[user@host ~]$ skopeo inspect \
  docker://registry.access.redhat.com/ubi9/nodejs-18
{
  "Name": "registry.access.redhat.com/ubi9/nodejs-18",
  "Digest": "sha256:741b...22e0",
  "RepoTags": [
    ...output omitted...
```

Use the `skopeo copy` command to copy images between registries. The following example copies the `registry.access.redhat.com/ubi9/nodejs-18:latest` image into the `quay.io/myuser/nodejs-18` Quay.io repository.

```
[user@host ~]$ skopeo copy \
  docker://registry.access.redhat.com/ubi9/nodejs-18 \
  docker://quay.io/myuser/nodejs-18
Getting image source signatures
...output omitted...
```

The following example changes the transport format to download an image into a local directory.

```
[user@host ~]$ skopeo copy \
  docker://registry.access.redhat.com/ubi9/nodejs-18 \
  dir:/var/lib/images/nodejs-18
Getting image source signatures
...output omitted...
```

See the references section for more details about Skopeo.

Manage Registry Credentials with Podman

Some registries require users to authenticate, such as the `registry.redhat.io` registry.

```
[user@host ~]$ podman pull registry.redhat.io/rhel8/httpd-24
Trying to pull registry.redhat.io/rhel8/httpd-24:latest...
Error: initializing source docker://registry.redhat.io/rhel8/
httpd-24:latest: unable to retrieve auth token: invalid username/password:
  unauthorized: Please login to the Red Hat Registry using your Customer Portal
  credentials. Further instructions can be found here: https://access.redhat.com/
RegistryAuthentication
```

You might choose a different image that does not require authentication, such as the UBI 8 image from the `registry.access.redhat.com` registry:

```
[user@host ~]$ podman pull registry.access.redhat.com/ubi8:latest
Trying to pull registry.access.redhat.com/ubi8:latest...
Getting image source signatures
Checking if image destination supports signatures
...output omitted...
```

Alternatively, authenticate your calls by executing the `podman login` command.

```
[user@host ~]$ podman login registry.redhat.io
Username: YOUR_USER
Password: YOUR_PASSWORD
Login Succeeded!
[user@host ~]$ podman pull registry.redhat.io/rhel8/httpd-24
Trying to pull registry.redhat.io/rhel8/httpd-24:latest...
Getting image source signatures
...output omitted...
```

Podman stores the credentials in the `${XDG_RUNTIME_DIR}/containers/auth.json` file, where the `${XDG_RUNTIME_DIR}` refers to a directory specific to the current user. The credentials are encoded in the base64 format:

```
[user@host ~]$ cat ${XDG_RUNTIME_DIR}/containers/auth.json
{
  "auths": {
    "registry.redhat.io": {
      "auth": "dXNlcjpodW50ZXIy"
    }
  }
}
[user@host ~]$ echo -n dXNlcjpodW50ZXIy | base64 -d
user:hunter2
```



Note

For security reasons, the `podman login` command does not show your password in the interactive session. Although you do not see what you are typing, Podman registers every key stroke. Press enter when you have typed your full password in the interactive session to initiate the login.

Skopeo uses the same `${XDG_RUNTIME_DIR}/containers/auth.json` file to access authentication details for each registry.



References

Red Hat - Manage Container Registries

<https://www.redhat.com/sysadmin/manage-container-registries>

`skopeo(1)` man page

`skopeo-copy(1)` man page

► Guided Exercise

Container Image Registries

Interact with container images in multiple container registries.

Outcomes

You should be able to:

- Log in to container image registries.
- Move container images between registries.
- Test Quay images.

Before You Begin

As the student user on the workstation machine, use the `lab` command to prepare your system for this exercise.

```
[student@workstation ~]$ lab start images-basics
```

Instructions

In this exercise, your team must ensure that the internal Dev and Test pipelines use the same Python container image as the one that is in Red Hat OpenShift Container Platform (RHOCP).

To do so, copy the Python image from the RHOCP registry to the internal registry:

- Source: `default-route-openshift-image-registry.apps.ocp4.example.com/default/python:3.9-ubi8`
- Destination: `registry.ocp4.example.com:8443/developer/python:3.9-ubi8`

► 1. Log in to the container image registries.

1.1. Log in to RHOCP as the `admin` user.

```
[student@workstation ~]$ oc login -u admin -p redhatocp \
https://api.ocp4.example.com:6443
Login successful.
...output omitted...
```



Note

RHOCP and the `oc` command are explored later in the course.

1.2. Log in to the RHOCP registry with Podman.

```
[student@workstation ~]$ podman login -u $(oc whoami) -p $(oc whoami -t) \
default-route-openshift-image-registry.apps.ocp4.example.com
Login Succeeded!
```

- 1.3. Log in to the `registry.ocp4.example.com:8443` registry with Podman.

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

- ▶ 2. Use `skopeo` to copy the `default-route-openshift-image-registry.apps.ocp4.example.com/default/python:3.9-ubi8` image to the `registry.ocp4.example.com:8443` registry.

```
[student@workstation ~]$ \
RHOCP_REGISTRY="default-route-openshift-image-registry.apps.ocp4.example.com"
[student@workstation ~]$ skopeo copy --dest-tls-verify=false \
docker://${RHOCP_REGISTRY}/default/python:3.9-ubi8 \
docker://registry.ocp4.example.com:8443/developer/python:3.9-ubi8
Getting image source signatures
...output omitted...
```

- ▶ 3. Make the image public in the internal registry.

- 3.1. In a web browser, navigate to `https://registry.ocp4.example.com:8443` and log in with the user `developer` and password `developer`.
- 3.2. Type `developer` in the **Filter Repositories** field, then click the `developer/python` repository.
- 3.3. Click the **Settings** icon at the bottom of the page. Scroll to the **Repository Visibility** settings and click **Make Public**. Then, click **OK**.

- ▶ 4. Test the image as an unauthenticated user.

- 4.1. In a terminal, log out from all container image registries.

```
[student@workstation ~]$ podman logout --all
Removed login credentials for all registries
```

- 4.2. Pull the `registry.ocp4.example.com:8443/developer/python:3.9-ubi8` image.

```
[student@workstation ~]$ podman pull \
registry.ocp4.example.com:8443/developer/python:3.9-ubi8
Trying to pull registry.ocp4.example.com:8443/developer/python:3.9-ubi8...
...output omitted...
e972...ac821
```

- 4.3. Start a container that uses the image.

```
[student@workstation ~]$ podman run --rm \
registry.ocp4.example.com:8443/developer/python:3.9-ubi8 python3 --version
Python 3.9.13
```

Finish

On the `workstation` machine, use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish images-basics
```

Managing Images

Objectives

- Pull and manage container images.

Image Management

Image management includes different operations:

- Tagging image versions so that they map to product versions and updates.
- Pulling images into your system.
- Building images.
- Pushing images to an image repository.
- Inspecting images to get metadata.
- Removing images to recover storage space.

For the complete list of image-related commands, execute `podman image --help` in a terminal window.

Image Versioning and Tags

Because container images package software, developers often consider images as deployment artifacts. To keep images up to date, developers often map the image versions to the versions of the packaged software. Keeping images up to date also means updating the OS libraries within the image to receive improvements and security fixes.

One way to version images relative to their packaged software product is to use semantic versioning. Semantic version numbers form a string with the format `MAJOR.MINOR.PATCH` meaning:

- MAJOR: backward incompatible changes
- MINOR: backward compatible changes
- PATCH: bug fixes

Because versioning has no enforced structure, it is up to the image maintainers to follow good versioning practices. This is one reason why you should use trusted image registries and repositories.

Image versions can be used in the image name or in the image tag. An image tag is a string that you specify after the image name. Also, the same image can have multiple tags.

```
[<image repository>/<namespace>/]<image name>[:<tag>]
```

Using a tag in Podman is optional. When you do not specify a tag in a Podman command, Podman uses the `latest` tag by default.

```
[user@host ~]$ podman pull quay.io/argoproj/argocd
Trying to pull quay.io/argoproj/argocd:latest...
...output omitted...
```

Using the `latest` tag is considered a bad practice. Because the `latest` tag also represents the latest version of the image, it can include backwards-incompatible changes and cause containers that use the image to break.

To create additional tags for local images, use the `podman image tag` command.

```
[user@host ~]$ podman image tag LOCAL_IMAGE:TAG LOCAL_IMAGE:NEW_TAG
```

Pulling Images

To search for images in different image registries, use a web browser to navigate to the registry URL and use the web UI.

Alternatively, use the `podman search` command to search for images in all the registries present in the `unqualified-search-registries` list in your `registries.conf` file. This enables you to search multiple registries.

```
[user@host ~]$ podman search nginx
NAME                                     DESCRIPTION
registry.fedoraproject.org/f29/nginx
...output omitted...
registry.access.redhat.com/ubi8/nginx-118      Platform for running nginx 1.18 or
                                                 building...
...output omitted...
docker.io/library/nginx                      Official build of Nginx.
...output omitted...
quay.io/linuxserver.io/baseimage-alpine-nginx
...output omitted...
```

To retrieve an image, run `podman image pull IMAGE_NAME`, which downloads the image from a registry. Alternatively, `podman pull IMAGE_NAME` provides the same functionality.

```
[user@host ~]$ podman pull registry.redhat.io/rhel8/mariadb-103:1
```

When you pull an image as a non-root user, Podman stores container images in the `~/ .local/share/containers` directory.

To find which images your user has available locally, use the `podman image ls` or `podman images` command.

```
[user@host ~]$ podman image ls
REPOSITORY                                     TAG      IMAGE ID      CREATED        SIZE
registry.redhat.io/rhel9/python-39            1-52    d336a3191d35  3 weeks ago   995 MB
registry.access.redhat.com/ubi8/python-39      latest   6b7a42c9d513  4 weeks ago   894 MB
...output omitted...
```

If an image is pulled by the root user, then it is stored in the `/var/lib/containers` directory. This image is only listed when `podman image ls` is run as root.

Building Images

You can also build an image from a Containerfile, which describes the steps used to build an image. Run the `podman build --file CONTAINERFILE --tag IMAGE_REFERENCE` to build a container image.

For example, to build an image that you can later push to Red Hat Quay.io, execute the following command:

```
[user@host ~]$ podman build --file Containerfile \
--tag quay.io/YOUR_QUAY_USER/IMAGE_NAME:TAG
```

Pushing Images

After you build an image, share it by pushing it to a remote registry. To push an image, you must be logged in to the registry. Run the `podman login REGISTRY` to log in to the specified registry. Then, you can use the `podman push IMAGE` command to push a local image to the remote registry.

For example, to push an image to Quay.io, run the following command:

```
[user@host ~]$ podman push quay.io/YOUR_QUAY_USER/IMAGE_NAME:TAG
Getting image source signatures
Copying blob fb3154998920 done
...output omitted...
Writing manifest to image destination
Storing signatures
```

Inspecting Images

The `podman image inspect` command provides useful information about a locally available image in your system.

The following example usage shows information about a `mariadb` image.

```
[user@host ~]$ podman image inspect registry.redhat.io/rhel8/mariadb-103:1
[
  {
    "Id": "6683...98ea",
    ...output omitted...
    "Config": {
      "User": "27", ①
      "ExposedPorts": { ②
        "3306/tcp": {}
      },
      "Env": [ ③
        "PATH=/opt/app-root/src/bin:/opt/app-root/bin:/usr/local/sbin:/usr/
local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
        ...output omitted...
      ],
      "Entrypoint": [ ④
        "container-entrypoint"
      ],
      "Cmd": [ ⑤
        ...
      ]
    }
  }
]
```

```

        "run-mysqld"
],
"WorkingDir": "/opt/app-root/src", 6
"Labels": { 7
    ...output omitted...
    "release": "177.1654147959",
    "summary": "MariaDB 10.3 SQL database server",
    "url": "https://access.redhat.com/containers/#/
registry.access.redhat.com/rhel8/mariadb-103/images/1-177.1654147959",
    ...output omitted...
    "Architecture": "amd64", 8
    "Os": "linux",
    "Size": 573593952,
    ...output omitted...

```

- 1** The default user for the image.
- 2** The port that the application exposes.
- 3** The environment variables used by the image.
- 4** The entrypoint, a command that runs when the container starts.
- 5** The command that the `container-entrypoint` script runs.
- 6** The working directory for the commands in the image.
- 7** Labels providing extra metadata.
- 8** The architecture where this image can be used.

The output from `podman inspect` is verbose, which makes it hard to find information. To select a specific part of the output use the Go templating feature in Podman by providing the `--format` option. Use the `podman inspect` output keys preceded by dots and surrounded by double curly braces.

```
--format="{{.Key.Subkey}}"
```

For example, the following command extracts the `CMD` instruction from the `rhel8/mariadb-103` image:

```
[user@host ~]$ podman image \
inspect registry.redhat.io/rhel8/mariadb-103:1 \
--format="{{.Config.Cmd}}"
[run-mysqld]
```

To inspect a remote image, you can use the Skopeo tool.

Image Removal

You can delete local images that are no longer used by any container. Run the `podman image rm` or the `podman rmi` command to delete a container image.

```
[user@host ~]$ podman image rm REGISTRY/NAMESPACE/IMAGE_NAME:TAG
```

Chapter 3 | Container Images

If the image is in use by a container, then Podman fails to remove it. You must first remove any containers using the image by running `podman stop container-name`. Alternatively, force Podman to remove the image by providing the `-f` option. This automatically stops and removes any containers that use the image and then removes the image.

```
[user@host ~]$ podman image rm -f REGISTRY/NAMESPACE/IMAGE_NAME:TAG
```

With the `--all` option, you can delete all images in the local storage.

```
[user@host ~]$ podman rmi --all
```

```
[user@host ~]$ podman image rm --all
```

Images without tags and that are not referenced by other images are considered dangling images. Use the `podman image prune` command to delete dangling images from your local storage. When executing the `podman image prune` command, Podman displays an interactive prompt to confirm image removal.

```
[user@host ~]$ podman image prune
```

WARNING! This command removes all dangling images.

Are you sure you want to continue? [y/N]

To delete both dangling and unused images, provide the `--all` or `-a` option.

```
[user@host ~]$ podman image prune -a
```

WARNING! This command removes all images without at least one container associated with them.

Are you sure you want to continue? [y/N]

You can include the `-f` option to force the removal and to avoid the interactive prompt.

```
[user@host ~]$ podman image prune -af
```

Export and Import File Systems

The `podman export` command exports a container's filesystem to a .tar file on your local machine. This command creates a snapshot of an existing container, so you can use it later. For example, if you inadvertently make changes to your container's filesystem and do not know how to fix it, then use the snapshot to return to a known starting point. By default, the `podman export` command writes to the standard output (STDOUT). To redirect the output to a file use the `--output` or `-o` option, specifying the name for the archive to create, and the container name or ID to export as arguments.

```
[user@host ~]$ podman export -o mytarfile.tar fb601b05cd3b
```

To import a .tar file containing a container file system, and save the file system as a container image, use the `podman import` command. The `podman import` command requires the image name and tag as arguments.

```
[user@host ~]$ podman import mytarfile.tar httpdcustom:2.4
Getting image source signatures
Copying blob 47662b708e31 done  |
Copying config 9af04983ef done  |
Writing manifest to image destination
sha256:9af0...4c8f
```

After importing a file system, you can verify the creation of the container image by using the `podman images` command.

```
[user@host ~]$ podman images
REPOSITORY           TAG      IMAGE ID      CREATED       SIZE
localhost/httpdcustom    2.4      9af04983ef93  18 minutes ago  305 MB
registry.../rhscl/httpd-24-rhel7  latest   699f5c8b7fd3  2 months ago   330 MB
```



References

semver.org

<https://semver.org/>

[podman-export\(1\) man page](#)

[podman-import\(1\) man page](#)

For more information, refer to the *Exporting and importing containers* section in the *Building, running, and managing containers* at

https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/9/html-single/building_running_and_managing_containers/index

For more information, refer to the *Image tags and versions* section in the *Red Hat Ecosystem Catalog - Help* at

<https://redhat-connect.gitbook.io/catalog-help/container-images/container-image-details/image-tags-and-versions>

► Guided Exercise

Managing Images

Learn to manage container images.

Outcomes

You should be able to manage images by performing common operations such as:

- Creating images
- Listing images
- Inspecting images
- Tagging images
- Removing images
- Searching images
- Pulling images

Before You Begin

As the student user on the workstation machine, use the `lab` command to prepare your system for this exercise.

This command copies an example Containerfile that you use to build the `simple-server` image.

```
[student@workstation]$ lab start images-managing
```

Instructions

- 1. Verify that the `simple-server` image is not present on your machine.

```
[student@workstation ~]$ podman image ls --format "{{.Repository}}"
no output expected
```



Note

You might see images from previous exercises if you did not run the `lab finish` scripts.

- 2. Build and run the `simple-server` image by using the `~/D0188/labs/images-managing/Containerfile` file.

- 2.1. Change to the exercise directory and examine its contents.

```
[student@workstation ~]$ cd ~/D0188/labs/images-managing
no output expected
```

- 2.2. Use the exercise Containerfile and the `podman build` command to create the image. Use the `-t` option to call the image `simple-server`.

```
[student@workstation images-managing]$ podman build -f Containerfile -t \
simple-server
...output omitted...
Successfully tagged localhost/simple-server:latest
87dc...fc02
```

- 2.3. Use the `podman image ls` command to verify that the `simple-server` image is present.

```
[student@workstation images-managing]$ podman image ls
REPOSITORY          TAG      IMAGE ID      CREATED       SIZE
localhost/simple-server    latest    87dc46f07c8c  5 minutes ago  892 MB
```

Because you did not provide a server name and a tag to the `simple-server` image name, the image uses the `localhost` registry server and the `latest` tag by default.

- 2.4. Create a container from the `simple-server` image and call it `http-server`. Use the `-d` option to run it in the background and the `-p` option to map your local host port 8080 to the container port 8000.

```
[student@workstation images-managing]$ podman run -d \
-p 8080:8000 \
--name http-server \
simple-server
2b81..0207
```

- 2.5. Use a tool such as `curl` or a web browser to send a request to `http://localhost:8080/hello.html`.

```
[student@workstation images-managing]$ curl http://localhost:8080/hello.html
Hello from the container
```

► 3. Inspect the `simple-server` image to see which command it runs by default.

Verify that `python -m http.server` is the default command for this image. This Python command comes from the `CMD` instruction in the exercise Containerfile.

```
[student@workstation images-managing]$ podman image inspect simple-server \
--format="{{.Config.Cmd}}"
[/bin/sh -c python -m http.server]
```

► 4. Tag the `simple-server` image with the `0.1` tag and list the images to verify the result.

```
[student@workstation images-managing]$ podman image tag simple-server \
simple-server:0.1
no output expected
```

When you list the images in your machine you can see that `simple-server` shows for the `latest` tag, and also for the `0.1` tag. Verify that the `IMAGE ID` has the same value because both tags point to the same image.

```
[student@workstation images-managing]$ podman image ls
REPOSITORY          TAG      IMAGE ID   CREATED     SIZE
localhost/simple-server    latest    87dc46f07c8c  10 minutes ago  892 MB
localhost/simple-server    0.1      87dc46f07c8c  10 minutes ago  892 MB
```

- 5. Remove the `simple-server` image completely from your system.

- 5.1. Use the `podman image rm` command to remove the `simple-server`.

```
[student@workstation images-managing]$ podman image rm simple-server
Untagged: localhost/simple-server:latest
```

The previous command only removed the `latest` tag because you did not specify the tag in the command and you have two tags pointing to the same image.

- 5.2. Try to delete the `simple-server:0.1` tag to remove the `simple-server` image completely.

```
[student@workstation images-managing]$ podman image rm simple-server:0.1
Error: Image used by 1bd7...250d: image is in use by a container
```

The error message indicates that you cannot remove an image that is currently being used by a container. Before removing the image you must stop and remove the container.

- 5.3. Force the container and container image deletion.

```
[student@workstation images-managing]$ podman image rm -f simple-server:0.1
WARN[0010] StopSignal SIGTERM failed to stop container http-server in 10 seconds,
resorting to SIGKILL
Untagged: localhost/simple-server:0.1
Deleted: 87dc...fc02
Deleted: 2318...7bbb
```

Finish

On the workstation machine, use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation]$ lab finish images-managing
```

► Lab

Container Images

Use Podman to create and pull an image from a container registry.

Outcomes

You should be able to:

- Create a container image from a Containerfile by using Quay.
- Pull an image from the Quay container registry.
- Add a tag to a container image.

Before You Begin

As the student user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command verifies that Podman is available and provides a Containerfile template.

The `lab` script continuously evaluates the objectives of this lab. Keep the script running in a terminal window and complete the objectives of this lab from a new terminal window.

After each objective, return to the lab script evaluation to see if you have finished the objective successfully. When you finish all objectives, the `lab` command prompts you to execute the `finish` function.

```
[student@workstation ~]$ lab start images-lab
```

Instructions

1. Build a container image that uses the `~/D0188/labs/images-lab/Containerfile` file. Call the resulting image `images-lab` and push the image into the `registry.ocp4.example.com:8443` registry in the `developer` user repository. Use the `developer` user with the `developer` password to authenticate with the `registry.ocp4.example.com:8443` registry.
Optional: use the `curl` command to verify that the HTTP server is running.
2. Add the `grue` tag to the `images-lab` container image, and push it to the `registry.ocp4.example.com:8443` registry.
3. Create a container by using the `images-lab:grue` image. Use the `images-lab` container name, and bind the `8080` container port to the `8080` host port. Start the container in the background.

Finish

As the student user on the `workstation` machine, use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

Press **y** when the **lab start** command prompts you to execute the **finish** function.
Alternatively, execute the following command:

```
[student@workstation ~]$ lab finish images-lab
```

► Solution

Container Images

Use Podman to create and pull an image from a container registry.

Outcomes

You should be able to:

- Create a container image from a Containerfile by using Quay.
- Pull an image from the Quay container registry.
- Add a tag to a container image.

Before You Begin

As the student user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command verifies that Podman is available and provides a Containerfile template.

The lab script continuously evaluates the objectives of this lab. Keep the script running in a terminal window and complete the objectives of this lab from a new terminal window.

After each objective, return to the lab script evaluation to see if you have finished the objective successfully. When you finish all objectives, the `lab` command prompts you to execute the `finish` function.

```
[student@workstation ~]$ lab start images-lab
```

Instructions

1. Build a container image that uses the `~/D0188/labs/images-lab/Containerfile` file. Call the resulting image `images-lab` and push the image into the `registry.ocp4.example.com:8443` registry in the `developer` user repository. Use the `developer` user with the `developer` password to authenticate with the `registry.ocp4.example.com:8443` registry.
 - 1.1. Authenticate Podman with the `registry.ocp4.example.com:8443` registry.

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

- 1.2. In a terminal, change into the `~/D0188/labs/images-lab` directory.

```
[student@workstation ~]$ cd ~/D0188/labs/images-lab
no output expected
```

Chapter 3 | Container Images

- 1.3. Build the Containerfile with the `registry.ocp4.example.com:8443/developer/images-lab` container image name.

```
[student@workstation images-lab]$ podman build --file Containerfile \
--tag registry.ocp4.example.com:8443/developer/images-lab
...output omitted...
Successfully tagged registry.ocp4.example.com:8443/developer/images-lab:latest
8d14...dd5a
```

- 1.4. Push the image to the `registry.ocp4.example.com:8443` registry.

```
[student@workstation images-lab]$ podman push \
registry.ocp4.example.com:8443/developer/images-lab
...output omitted...
Writing manifest to image destination
Storing signatures
```

2. Add the `grue` tag to the `images-lab` container image, and push it to the `registry.ocp4.example.com:8443` registry.

- 2.1. Add the `grue` tag to the image.

```
[student@workstation images-lab]$ podman tag \
registry.ocp4.example.com:8443/developer/images-lab \
registry.ocp4.example.com:8443/developer/images-lab:grue
no output expected
```

- 2.2. Push the new image tag.

```
[student@workstation images-lab]$ podman push \
registry.ocp4.example.com:8443/developer/images-lab:grue
...output omitted...
Writing manifest to image destination
Storing signatures
```

3. Create a container by using the `images-lab:grue` image. Use the `images-lab` container name, and bind the `8080` container port to the `8080` host port. Start the container in the background.

Optionally, use the `curl` command to verify that the HTTP server is running.

- 3.1. Start the container.

```
[student@workstation images-lab]$ podman run -d --name images-lab \
-p 8080:8080 images-lab:grue
2422...e7b1
```

- 3.2. Connect to the HTTP server by using `curl`. Note that this step is not required to make the lab script pass.

```
[student@workstation images-lab]$ curl localhost:8080
It is pitch black. You are likely to be eaten by a grue.
```

Finish

As the **student** user on the **workstation** machine, use the **lab** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

Press **y** when the **lab start** command prompts you to execute the **finish** function.
Alternatively, execute the following command:

```
[student@workstation ~]$ lab finish images-lab
```

Summary

- Use container registries to store and share container images.
- Red Hat provides several container registries for both official Red Hat images and for uploading your own.
- Configure container registries for Podman.
- Manage container images with Podman, including the following image operations:
 - Listing
 - Pulling
 - Building
 - Pushing
 - Inspecting
 - Tagging
 - Removing

Chapter 4

Custom Container Images

Goal

Build custom container images to containerize applications.

Objectives

- Create a Containerfile by using basic commands.
- Create a Containerfile that uses best practices.
- Run rootless containers with Podman.

Sections

- Create Images with Containerfiles (and Guided Exercise)
- Build Images with Advanced Containerfile Instructions (and Guided Exercise)
- Rootless Podman (and Guided Exercise)

Lab

- Custom Container Images

Create Images with Containerfiles

Objectives

- Create a Containerfile by using basic commands.

Creating Images with Containerfiles

A *Containerfile* lists a set of instructions that a container runtime uses to build a container image. You can use the image to create any number of containers.

Each instruction causes a change that is captured in a resulting image layer. These layers are stacked together to form the resulting container image.



Note

You might have seen or used Dockerfiles instead of Containerfiles. These files are largely the same and mainly differ in historical context.

This course uses Containerfiles because they are not associated with any specific container runtime engine. Podman supports both Dockerfiles and Containerfiles.

Choosing a Base Image

When you build an image, podman executes the instructions in the Containerfile and applies the changes on top of a container *base image*. A base image is the image from which your Containerfile and its resulting image is built.

The base image you choose determines the Linux distribution and its components, such as:

- Package manager
- Init system
- Filesystem layout
- Preinstalled dependencies and runtimes

The base image can also influence other factors, such as image size, vendor support, and processor compatibility.

Red Hat provides container images intended as a common starting point for containers known as *universal base images (UBI)*. These UBIs come in four variants: `standard`, `init`, `minimal`, and `micro`. Additionally, Red Hat also provides UBI-based images that include popular runtimes, such as Python and Node.js.

These UBIs use Red Hat Enterprise Linux (RHEL) at their core and are available from the Red Hat Container Catalog. The main differences are as follows:

Standard

This is the primary UBI, which includes DNF, systemd, and utilities such as `gzip` and `tar`.

Init

Simplifies running multiple applications within a single container by managing them with `systemd`.

Minimal

This image is smaller than the `init` image and still provides nice-to-have features. This image uses the `microdnf` minimal package manager instead of the full-sized version of DNF.

Micro

This is the smallest available UBI because it only includes the bare minimum number of packages. For example, this image does not include a package manager.

Containerfile Instructions

Containerfiles use a small domain-specific language (DSL) consisting of basic instructions for crafting container images. The following are the most common instructions.

FROM

Sets the base image for the resulting container image. Takes the name of the base image as an argument.

WORKDIR

Sets the current working directory within the container. Instructions that follow the `WORKDIR` instruction run within this directory.

COPY and ADD

Copy files from the build host into the file system of the resulting container image. Relative paths use the host current working directory, known as the build context. Both instructions use the working directory within the container as defined by the `WORKDIR` instruction.

The `ADD` instruction adds the following functionality:

- Copying files from URLs.
- Unpacking `tar` archives in the destination image.

Because the `ADD` instruction adds functionality that might not be obvious, developers tend to prefer the `COPY` instruction for copying local files into the container image.

RUN

Runs a command in the container and commits the resulting state of the container to a new layer within the image.

ENTRYPOINT

Sets the executable to run when the container is started.

CMD

Runs a command when the container is started. This command is passed to the executable defined by `ENTRYPOINT`. Base images define a default `ENTRYPOINT`, which is usually a shell executable, such as Bash.

**Note**

Neither `ENTRYPOINT` nor `CMD` run when building a container image. Podman executes them when you start a container from the image.

USER

Changes the active user within the container. Instructions that follow the `USER` instruction run as this user, including the `CMD` instruction. It is a good practice to define a different user other than `root` for security reasons.

LABEL

Adds a key-value pair to the metadata of the image for organization and image selection.

Chapter 4 | Custom Container Images

EXPOSE

Adds a port to the image metadata indicating that an application within the container binds to this port. This instruction does not bind the port on the host and is for documentation purposes.

ENV

Defines environment variables that are available in the container. You can declare multiple ENV instructions within the Containerfile. You can use the env command inside the container to view each of the environment variables.

ARG

Defines build-time variables, typically to make a customizable container build. Developers commonly configure the ENV instructions by using the ARG instruction. This is useful for preserving the build-time variables for runtime.

VOLUME

Defines where to store data outside of the container. The value configures the path where Podman mounts persistent volume inside of the container. You can define more than one path to create multiple volumes.

Each Containerfile instruction runs in an independent container by using an intermediate image built from every previous command. This means each instruction is independent from other instructions in the Containerfile. The following is an example Containerfile for building a simple Apache web server container:

```
# This is a comment line ①
FROM      registry.redhat.io/ubi8/ubi:8.6 ②
LABEL     description="This is a custom httpd container image" ③
RUN       yum install -y httpd ④
EXPOSE   80 ⑤
ENV       LogLevel "info" ⑥
ADD       http://someserver.com/filename.pdf /var/www/html ⑦
COPY      ./src/  /var/www/html/ ⑧
USER     apache ⑨
ENTRYPOINT ["/usr/sbin/httpd"] ⑩
CMD      ["-D", "FOREGROUND"] ⑪
```

- ① Lines that begin with a hash, or pound, sign (#) are comments.
- ② The FROM instruction declares that the new container image extends the `registry.redhat.io/ubi8/ubi:8.6` container base image.
- ③ The LABEL instruction adds metadata to the image.
- ④ RUN executes commands in a new layer on top of the current image. The shell that is used to execute commands is `/bin/sh`.
- ⑤ EXPOSE configures metadata indicating that the container listens on the specified network port at runtime.
- ⑥ ENV is responsible for defining environment variables that are available in the container.
- ⑦ The ADD instruction copies files or directories from a local or remote source and adds them to the container's file system.
- ⑧ COPY copies files from a path relative to the working directory and adds them to the container's file system.

- ⑨ USER specifies the username or the UID to use when running the container image for the RUN, CMD, and ENTRYPOINT instructions.
- ⑩ ENTRYPOINT specifies the default command to execute when users instantiate the image as a container.
- ⑪ CMD provides the default arguments for the ENTRYPOINT instruction.

Container Image Tags

When building a container image, you can specify an image name to later identify the image. The image name is a string composed of letters, numbers, and some special characters.

An *image tag* comes after the image name and is delimited by a colon (:). When you omit an image tag, podman uses the default `latest` tag.



Note

It is a best practice to specify tags in addition to `latest` tag. Because `latest` is the default tag applied to new images, references that use the `latest` tag can change unintentionally.

The full name of the image includes both a name and an optional image tag.

For example, in the full image name `my-app:1.0`, the name is `my-app` and the tag is `1.0`.



References

What is a container image?

<https://opensource.com/article/21/8/container-image>

Best practices for building images that pass Red Hat Container Certification

<https://developers.redhat.com/articles/2021/11/11/best-practices-building-images-pass-red-hat-container-certification/#>

For more information on the differences between UBI variants, refer to the *Characteristics of UBI Images* chapter in the *Building, running, and managing containers* guide for RHEL 9 at

https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/9/html-single/building_running_and_managing_containers/index#con_characteristics-of-ubi-images_assembly_types-of-container-images

The difference between an image *name* and an image *tag* is explained in the Podman documentation at

<https://docs.podman.io/en/latest/markdown/podman-tag.1.html>

► Guided Exercise

Create Images with Containerfiles

Create a basic Containerfile for an example Node.js application. Throughout the exercise, you gradually improve the Containerfile and the resulting container image.

Outcomes

You should be able to:

- Create a Containerfile by using an appropriate base image.
- Use common Containerfile instructions, such as FROM, COPY, RUN, and CMD.
- Optimize instructions to reduce image size and number of layers.

Before You Begin

As the student user on the workstation machine, use the lab command to prepare your system for this exercise.

```
[student@workstation ~]$ lab start custom-containerfiles
```

Instructions

- 1. Create a Containerfile for the hello-server Node.js application.



Important

The Containerfile you create in this initial step does not follow Red Hat guidance for creating container images. You improve this Containerfile throughout the exercise.

- 1.1. Change into the ~/D0188/labs/custom-containerfiles/hello-server directory.

```
[student@workstation ~]$ cd ~/D0188/labs/custom-containerfiles/hello-server  
no output expected
```

- 1.2. Create a file called Containerfile with the following contents:

```
FROM registry.ocp4.example.com:8443/redhattraining/podman-ubi9.2  
  
COPY . /tmp/hello-server  
  
RUN dnf module enable -y nodejs:18  
  
RUN dnf install -y nodejs
```

```
RUN cd /tmp/hello-server && npm install  
  
CMD cd /tmp/hello-server && npm start
```

**Note**

This exercise uses a customized UBI image that replaces the default UBI repositories with an internal classroom repository to eliminate the impact of network errors.

By default, Red Hat serves UBI repositories at the <https://cdn-ubi.redhat.com/content/public/ubi/> public CDN. Classroom environments might experience problems when connecting to this CDN.

Under reliable network conditions, use the default UBI image.

For similar reasons, the `npm install` command uses an internal NPM registry, specified in the `.npmrc` file.

- 1.3. Build a container image tagged `hello-server:bad` by using the `Containerfile` you created.

```
[student@workstation hello-server]$ podman build -t hello-server:bad .  
...output omitted...  
Successfully tagged localhost/hello-server:bad  
...output omitted...
```

- 1.4. Create a container called `hello-bad` that uses the `hello-server:bad` image and binds the container port 3000 to the host port 3000.

```
[student@workstation hello-server]$ podman run -d --rm --name hello-bad \  
-p 3000:3000 hello-server:bad  
ecb8...b5af
```

The command prints the ID of the running container, which differs on your machine.

- 1.5. Make a request to the running container by using the `curl` command.

```
[student@workstation hello-server]$ curl http://localhost:3000/greet ;echo  
{"hello":"world"}
```

- 1.6. Stop the `hello-bad` container.

```
[student@workstation hello-server]$ podman stop hello-bad  
hello-bad
```

- 2. Update the `Containerfile` to reduce the number of image layers it produces and simplify directory usage.

- 2.1. Inspect the number of image layers in the `hello-server:bad` image.

```
[student@workstation hello-server]$ podman image tree hello-server:bad
...output omitted...
Image Layers
└─ ID: 6f740e943089 Size: 217MB Top Layer of: [registry.ocp4.example.com:8443/
ubi9/ubi9.2]
  └─ ID: 9aa75dbb96f3 Size: 5.12kB Top Layer of: [registry.ocp4.example.com:8443/
redhattraining/podman-ubi9.2:latest]
    └─ ID: a91b00af1e08 Size: 9.216kB
    └─ ID: 227220780963 Size: 19.01MB
    └─ ID: d61636a98dee Size: 192.8MB
    └─ ID: 7f801d74e40b Size: 162.7MB Top Layer of: [localhost/hello-server:bad]
```

The layer IDs and sizes differ in your environment from the preceding example.

Notice that the Containerfile creates several layers on top of the Red Hat UBI base image.

- 2.2. Open the Containerfile that you created and refactor the file to use a **WORKDIR** instruction. Update the paths in the rest of the file to match.

```
FROM registry.ocp4.example.com:8443/redhattraining/podman-ubi9.2

WORKDIR /tmp/hello-server

COPY . .

RUN dnf module enable -y nodejs:18

RUN dnf install -y nodejs

RUN npm install

CMD npm start
```

By using **WORKDIR**, the other instructions no longer need to change directories. This makes reading the Containerfile easier and reduces potential mistakes in file paths.

The **COPY . .** instruction copies the content of the current directory (`~/DO188/labs/custom-containerfiles/hello-server`) into the current directory in the container (`/tmp/hello-server`).

- 2.3. Update the Containerfile by merging the RUN instructions into a single instruction. The final Containerfile contains the following instructions:

```
FROM registry.ocp4.example.com:8443/redhattraining/podman-ubi9.2

WORKDIR /tmp/hello-server

COPY . .
```

```
RUN dnf module enable -y nodejs:18 && \
dnf install -y nodejs && \
npm install

CMD npm start
```

- 2.4. Build the container image by using the updated Containerfile. Use the `hello-server:better` image tag.

```
[student@workstation hello-server]$ podman build -t hello-server:better .
...output omitted...
Successfully tagged localhost/hello-server:better
...output omitted...
```

- 2.5. Inspect the number of image layers in the `hello-server:better` image.

```
[student@workstation hello-server]$ podman image tree hello-server:better
...output omitted...
Image Layers
└─ ID: 6f740e943089 Size: 217MB Top Layer of: [registry.ocp4.example.com:8443/
ubi9/ubi9:2]
└─ ID: 9aa75dbb96f3 Size: 5.12kB Top Layer of: [registry.ocp4.example.com:8443/
redhattraining/podman-ubi9.2:latest]
└─ ID: eaeec6f5f18b Size: 9.216kB
└─ ID: 73539cf6de6f Size: 365.2MB Top Layer of: [localhost/hello-server:better]
```

Notice that the Containerfile creates two layers on top of the `podman-ubi9.2` image. Reducing the number of RUN instructions also reduces the number of resulting image layers.

- 2.6. Run and test the container to verify that it works.

```
[student@workstation hello-server]$ podman run -d --rm --name hello-better \
-p 3000:3000 hello-server:better
d6c3...9f21
[student@workstation hello-server]$ curl http://localhost:3000/greet ;echo
{"hello":"world"}
[student@workstation hello-server]$ podman stop hello-better
hello-better
```

- 3. Update the Containerfile to use a better base image and use environment variables to adjust server settings.
- 3.1. Open the Containerfile and change the FROM instruction to use the Node.js runtime image provided by Red Hat. This image is based on the Red Hat Universal Base Image (UBI) and includes the Node.js runtime.
- ```
FROM registry.ocp4.example.com:8443/ubi9/nodejs-18-minimal:1
```
- 3.2. Because the base image includes the Node.js runtime, remove the `dnf module enable -y nodejs:18` and `dnf install -y nodejs` part of the RUN instruction.

```
COPY . .

RUN npm install

CMD npm start
```

This update does not change the number of image layers because the number of RUN instructions remains the same.

- 3.3. Add ENV instructions to set environment variables within the container that the application uses.

```
FROM registry.ocp4.example.com:8443/ubi9/nodejs-18-minimal:1

ENV SERVER_PORT=3000
ENV NODE_ENV="production"

WORKDIR /tmp/hello-server
```



### Note

Setting the NODE\_ENV environment variable to production instructs NPM to ignore the dependencies listed in the devDependencies section of the package.json file.

Because the devDependencies packages are not necessary to run the application, setting the NODE\_ENV variable to production reduces the image size.

The application uses the SERVER\_PORT environment variable to determine the port to which it binds.

- 3.4. Update the working directory to /opt/app-root/src, which is a better location than /tmp/hello-server.

```
WORKDIR /opt/app-root/src
```

## ► 4. Apply appropriate metadata to the image.

- 4.1. Add a LABEL instruction to indicate who is responsible for maintaining the image.

```
FROM registry.ocp4.example.com:8443/ubi9/nodejs-18-minimal:1

LABEL org.opencontainers.image.authors="Your Name"

ENV SERVER_PORT=3000
```

- 4.2. Add further LABEL instructions to provide hints as to the version and intended usage of the container image.

```
LABEL org.opencontainers.image.authors="Your Name"
LABEL com.example.environment="production"
LABEL com.example.version="0.0.1"

ENV SERVER_PORT=3000
```

- 4.3. Add an EXPOSE instruction to indicate that the application within the container binds to the port defined in the SERVER\_PORT environment variable.

```
ENV SERVER_PORT=3000
ENV NODE_ENV="production"

EXPOSE $SERVER_PORT

WORKDIR /opt/app-root/src
```

The EXPOSE instruction serves for documentation purposes. It does not bind the port on the host running the container.

- 4.4. The final Containerfile contains the following instructions:

```
FROM registry.ocp4.example.com:8443/ubi9/nodejs-18:1

LABEL org.opencontainers.image.authors="Your Name"
LABEL com.example.environment="production"
LABEL com.example.version="0.0.1"

ENV SERVER_PORT=3000
ENV NODE_ENV="production"

EXPOSE $SERVER_PORT

WORKDIR /opt/app-root/src

COPY . .

RUN npm install

CMD npm start
```

- 4.5. Build the container image by using the updated Containerfile. Use the hello-server:best image tag.

```
[student@workstation hello-server]$ podman build -t hello-server:best .
...output omitted...
Successfully tagged localhost/hello-server:best
...output omitted...
```

- 4.6. Inspect the container image to observe the added metadata.

```
[student@workstation hello-server]$ podman inspect hello-server:best \
-f '{{.Config.Env}}'
...output omitted... SERVER_PORT=3000 NODE_ENV=production]
```

4.7. Verify that the application works.

```
[student@workstation hello-server]$ podman run -d --rm --name hello-best \
-p 3000:3000 hello-server:best
d6c3...9f21
[student@workstation hello-server]$ curl http://localhost:3000/greet ;echo
{"hello":"world"}
[student@workstation hello-server]$ podman stop hello-best
hello-best
```

4.8. Change into the home directory.

```
[student@workstation hello-server]$ cd
no output expected
```

## Finish

On the **workstation** machine, use the **lab** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish custom-containerfiles
```

# Build Images with Advanced Containerfile Instructions

## Objectives

- Create a Containerfile that uses best practices.

## Advanced Containerfile Instructions

You can package your application in a container with the application runtime dependencies. However, developers commonly create containers that have a number of downsides, for example:

- The container is bound to a specific environment, and requires a rebuild before deploying to a production environment.
- The container contains developer tools, such as a debugger, text editors, or compilers.
- The container contains build-time dependencies that are not necessary at runtime.
- The container generates a large volume of files stored on the copy-on-write file system, which limits the performance of the application.

This section focuses on advanced container patterns that help you limit previously mentioned issues, such as:

- Reducing image storage footprint by using the multistage container build pattern.
- Customizing container runtime with environment variables.
- Using volumes to decrease the container size and increase the performance of writing files.

## The ENV Instruction

The ENV instruction lets you specify environment dependent configuration, for example, hostnames, ports or usernames. The containerized application can use the environment variables at runtime.

To include an environment variable, use the key=value format. The following example declares a DB\_HOST variable with the database hostname.

```
ENV DB_HOST="database.example.com"
```

Then you can retrieve the environment variable in your application. The following example is a Python script that retrieves the DB\_HOST environment variable.

```
from os import environ

DB_HOST = environ.get('DB_HOST')

Connect to the database at DB_HOST...
```

## The ARG Instruction

Use the ARG instruction to define build-time variables, typically to make a customizable container build.

You can optionally configure a default build-time variable value if the developer does not provide it at build time. Use the following syntax to define a build-time variable:

```
ARG key[=default value]
```

When you build the container image, use the `--build-arg` flag to set the value, such as `podman build --build-arg key=value` for the preceding example.

Consider the following Containerfile example:

```
ARG VERSION="1.16.8" \
 BIN_DIR=/usr/local/bin/

RUN curl "https://dl.example.io/${VERSION}/example-linux-amd64" \
 -o ${BIN_DIR}/example
```

If you do not provide the `--build-arg` flag, then Podman uses the default values during the build process. However, you can change the values during the build process without changing the Containerfile by using the ARG instruction.

Developers commonly configure the ENV instructions by using the ARG instruction. This is useful for preserving the build-time variables for runtime.

Consider the following Containerfile example:

```
ARG VERSION="1.16.8" \
 BIN_DIR=/usr/local/bin/

ENV VERSION=${VERSION} \
 BIN_DIR=${BIN_DIR}

RUN curl "https://dl.example.io/${VERSION}/example-linux-amd64" \
 -o ${BIN_DIR}/example
```

In the preceding example, the ENV instruction uses the value of the ARG instruction to configure the run-time environment variables.

If you do not configure the ARG default values but you must configure the ENV default values, then use the  `${VARIABLE:-DEFAULT_VALUE}` syntax, such as:

```
ARG VERSION \
 BIN_DIR

ENV VERSION=${VERSION:-1.16.8} \
 BIN_DIR=${BIN_DIR:-/usr/local/bin/}

RUN curl "https://dl.example.io/${VERSION}/example-linux-amd64" \
 -o ${BIN_DIR}/example
```

## The VOLUME Instruction

Use the VOLUME instruction to persistently store data. The value is the path where Podman mounts a persistent volume inside of the container. The VOLUME instruction accepts more than one path to create multiple volumes.

For example, the following Containerfile creates a volume to store PostgreSQL data.

```
FROM registry.redhat.io/rhel9/postgresql-13:1
VOLUME /var/lib/pgsql/data
```

In the previous Containerfile, if you add instructions that update `/var/lib/pgsql/data` after the VOLUME instruction, then those instructions are ignored.

To retrieve the local directory used by a volume, you can use the `podman inspect VOLUME_ID` command.

```
[user@host ~]$ podman inspect VOLUME_ID
[
 {
 "Name": "VOLUME_ID",
 "Driver": "local",
 "Mountpoint": "/home/your-name/.local/share/containers/storage/volumes/VOLUME_ID/_data",
 ...output omitted...
 "Anonymous": true
 }
]
```

The Mountpoint field gives you the absolute path to the directory where the volume exists on your host file system. Volumes created from the VOLUME instruction have a random ID in the Name field and are considered Anonymous volumes.

To remove unused volumes, use the `podman volume prune` command.

```
[user@host ~]$ podman volume prune
WARNING! This will remove all volumes not used by at least one container. The
following volumes will be removed:
8c0c...bcb8c
Are you sure you want to continue? [y/N] y
8c0c...bcb8c
```

You can create a *named* volume by using the `podman volume create` command.

```
[user@host ~]$ podman volume create VOLUME_NAME
VOLUME_NAME
```

You can also format the `podman volume ls` command to include the Mountpoint field of every volume. Persistent data storage with volumes is covered in depth later in the course.

```
[user@host ~]$ podman volume ls \
--format="{{.Name}}\t{{.Mountpoint}}"
0a8c...82c2 /home/your-name/.local/share/containers/storage/volumes/0a8c...82c2/
_data
252d...b2ed /home/your-name/.local/share/containers/storage/volumes/252d...b2ed/
_data
```

## The ENTRYPPOINT and CMD Instructions

The ENTRYPPOINT and CMD instructions specify the command to execute when the container starts. A valid Containerfile must have at least one of these instructions.

The ENTRYPPOINT instruction defines an executable, or command, that is always part of the container execution. This means that additional arguments are passed to the provided command.

Consider the following example:

```
FROM registry.access.redhat.com/ubi8/ubi-minimal:8.5
ENTRYPOINT ["echo", "Hello"]
```

Running a container from the previous Containerfile with no arguments prints "Hello".

```
[user@host ~]$ podman run my-image
Hello
```

If you provide the container with the "Red" and "Hat" arguments, then it prints "Hello Red Hat".

```
[user@host ~]$ podman run my-image Red Hat
Hello Red Hat
```

If you only use the CMD instruction, then passing arguments to the container overrides the command provided in the CMD instruction.

```
FROM registry.access.redhat.com/ubi8/ubi-minimal:8.5
CMD ["echo", "Hello", "Red Hat"]
```

Running a container from the previous Containerfile with no arguments prints Hello Red Hat.

```
[user@host ~]$ podman run my-image
Hello Red Hat
```

Running a container with the argument whoami overrides the echo command.

```
[user@host ~]$ podman run my-image whoami
root
```

When a Containerfile specifies both ENTRYPPOINT and CMD then CMD changes its behavior. In this case the values provided to CMD are passed as default arguments to the ENTRYPPOINT.

```
FROM registry.access.redhat.com/ubi8/ubi-minimal:8.5
ENTRYPOINT ["echo", "Hello"]
CMD ["Red", "Hat"]
```

The previous example prints Hello Red Hat when you run the container.

```
[user@host ~]$ podman run my-image
Hello Red Hat
```

If you provide the argument Podman to the container, then it prints Hello Podman.

```
[user@host ~]$ podman run my-image Podman
Hello Podman
```

The ENTRYPPOINT and CMD instructions have two formats for executing commands:

### **Text array**

The executable takes the form of a text array, such as:

```
ENTRYPOINT ["executable", "param1", ... "paramN"]
```

In this form you must provide the full path to the executable.

### **String form**

The command and parameters are written in a text form, such as:

```
CMD executable param1 ... paramN
```

The string form wraps the executable in a shell command such as sh -c "executable param1 ... paramN". This is useful when you require shell processing, for example for variable substitution.

You might need to change the container entrypoint at runtime, for example for troubleshooting. Consider the following Containerfile:

```
FROM registry.access.redhat.com/ubi8/ubi-minimal:8.5

LABEL GREETING="World"

ENTRYPOINT echo Hello "${GREETING}"
```

When you run the container, you notice unexpected output:

```
[user@host ~]$ podman run my-image
Hello
```

You can execute the env command to print the environment variables in the container. However, the preceding container uses the ENTRYPPOINT instruction, which means that you cannot change the echo command by adding an argument to the container execution:

```
[user@host ~]$ podman run my-image env
Hello
```

In that case, you can overwrite the entrypoint by using the `podman run --entrypoint` command.

```
[user@host ~]$ podman run --entrypoint env my-image
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
TERM=xterm
container=oci
HOME=/root
HOSTNAME=ba00a7663a93
```

Consequently, you verify that the `GREETING` environment variable is not set. This is because the developer that created the `Containerfile` used the `LABEL` instruction instead of the `ENV` instruction.

## Podman Secrets

A secret is a blob of sensitive information required by containers at runtime. This can be usernames, passwords, or keys. For example, if your containerized application requires credentials for connecting to a database, then you must store those credentials as a secret. After creating the secret, you must instruct the application container to make the credentials available to the application when it starts. To manage this process, use the `podman secret` subcommands.

### Creating Podman Secrets

With Podman, you can create secrets by using either a file, or by passing the sensitive information to the standard input (STDIN). To create a secret from a file, run the `podman secret create` subcommand specifying the name of the file containing the sensitive information, and the name of the secret to create as arguments. The output of the `podman secret create` subcommand displays the secret ID.

```
[user@host ~]$ echo "R3d4ht123" > dbsecretfile
[user@host ~]$ podman secret create dbsecret dbsecretfile
9c2400836ee16ed07d86a3122
```

Run the `podman secret create` subcommand specifying the secret name and `-` as arguments. The `-` argument instructs podman to read the sensitive information from standard input.

```
[user@host ~]$ printf "R3d4ht123" | podman secret create dbsecret2 -
875a1e46fa64639756968c644
```

The `podman secret create` command supports different drivers to store the secrets. You can use the `--driver` or `-d` option to specify one of the supported secret drivers:

#### **file (default)**

Stores the secret in a read-protected file

#### **pass**

Stores the secret in a GPG-encrypted file.

**shell**

Manages the secret storage by using a custom script.

To list stored secrets, use the `podman secret ls` subcommand.

```
[user@host ~]$ podman secret ls
ID NAME DRIVER CREATED UPDATED
875a1e46fa64639756968c644 dbsecret2 file About a minute ago About a
 minute ago
9c2400836ee16ed07d86a3122 dbsecret file 14 minutes ago 14 minutes
```

When you no longer need a secret, you can remove it by running the `podman secret rm` command, and providing the name of the secret as argument.

```
[user@host ~]$ podman secret rm dbsecret2
875a1e46fa64639756968c644
```

## Running Containers With Podman Secrets

To make secrets available for use to a container, execute the `podman run` command with the `--secret` option, and specify the name of the secret as parameter. You can use the `--secret` option multiple times for multiple secrets.

```
[user@host ~]$ podman run -it --secret dbsecret \
--name myapp registry.access.redhat.com/ubi8/ubi /bin/bash
[root@f9fedddbc81a /]# cat /run/secrets/dbsecret
R3d4ht123
```

When you use secrets, Podman retrieves the secret and places it on a tmpfs volume and then mounts the volume inside the container in the `/run/secrets` directory as a file based on the name of the secret.

To prevent secrets from being stored in an image, neither the `podman commit` nor `podman export` commands copy the secret data to an image or `.tar` file.

## Multistage Builds

A multistage build uses multiple `FROM` instructions to create multiple independent container build processes, also called stages. Every stage can use a different base image and you can copy files between stages. The resulting container image consists of the last stage.

Multistage builds can reduce the image size by only keeping the necessary runtime dependencies. For example, consider the following example, where a Node application is containerized.

```
FROM registry.redhat.io/ubi8/nodejs-14:1

WORKDIR /app
COPY . .

RUN npm install
RUN npm run build

RUN serve build
```

The `npm install` command installs the required NPM packages, which includes packages that are only needed at build-time. The `npm run build` command uses the packages to create an optimized production-ready application build. Then, the container uses an HTTP server to expose the application by using the `serve` command.

If you build an image from the previous Containerfile, then the image contains both the build-time and runtime dependencies, which increases the image size. The resulting image also contains the Node.js runtime, which is not used at container runtime but might increase the attack surface of the container.

To avoid this issue, you can define two stages:

- **First stage:** Build the application.
- **Second stage:** Copy and serve the static files by using an HTTP server, such as NGINX or Apache Server.

The following example implements the two-stage build process.

```
First stage
FROM registry.access.redhat.com/ubi8/nodejs-14:1 as builder ①
COPY ./ /opt/app-root/src/
RUN npm install
RUN npm run build ②

Second stage
FROM registry.access.redhat.com/ubi8/nginx-120 ③
COPY --from=builder /opt/app-root/src/ /usr/share/nginx/html ④
```

- ① Define the first stage with an alias. The second stage uses the `builder` alias to reference this stage.
- ② Build the application.
- ③ Define the second stage without an alias. It uses the `ubi8/nginx-120` base image to serve the production-ready version of the application.
- ④ Copy the application files to a directory in the final image. The `--from` flag indicates that Podman copies the files from the `builder` stage.

## Examine Container Data Layers

Container images use a copy-on-write (COW), layered file system. When you create a Containerfile, the `RUN`, `COPY`, and `ADD` instructions create *layers* (sometimes referred to as *blobs*).

The layered COW file system ensures that a container remains immutable. When you start a container, Podman creates and mounts a thin, ephemeral, writable layer on top of the container image layers. When you delete the container, Podman deletes the writable thin layer. This means that all container layers stay identical, except for the thin writable layer.

## Cache Image Layers

Because all container layers are identical, multiple containers can share the layers. This means Podman can cache layers, and build only those layers that are modified or not cached.

You can use caching to decrease build time. For example, consider a Node.js application Containerfile:

```
...content omitted...
COPY . /app/
...content omitted...
```

The previous instruction copies every file and directory in the Containerfile directory to the /app directory in the container. This means that any changes to the application result in the rebuilding of every layer after the COPY layer.

You can change the instructions as follows:

```
...content omitted...
COPY package.json /app/
RUN npm ci --production
COPY src ./src
...content omitted...
```

This means that if you change your application source code in the src directory and rebuild your container image, then the dependency layer is cached and skipped, which reduces the build time. Podman rebuilds the dependency layer only when you change the package.json file.

## Reduce Image Layers

You can reduce the number of container image layers, for example, by chaining instructions. Consider the following RUN instructions:

```
RUN mkdir /etc/gitea
RUN chown root:gitea /etc/gitea
RUN chmod 770 /etc/gitea
```

You can reduce the number of layers to one by chaining the commands. In Linux, you can chain commands by using the double ampersand (&&). You can also use the backslash character (\) to break a long command into multiple lines.

Consequently, the resulting RUN command looks as follows:

```
RUN mkdir /etc/gitea && \
 chown root:gitea /etc/gitea && \
 chmod 770 /etc/gitea
```

The advantage of chaining commands is that you create less container image layers, which typically results in smaller images.

However, chained commands are more difficult to debug and cache.

You can also create Containerfiles that do not use chained commands, and configure Podman to squash the layers. Use the --squash option to squash layers declared in the Containerfile. Alternatively, use the --squash-all option to also squash the layers from the parent image.

For example, consider the following three builds of one Containerfile.

```
[user@host ~]$ podman build -t localhost/not-squashed .
...output omitted...
[user@host ~]$ podman build --squash -t localhost/squashed .
```

```
...output omitted...
[user@host ~]$ podman build --squash-all -t localhost/squashed-all .
...output omitted...
[user@host ~]$ podman images --format="{{.Names}}\t{{.Size}}"
[localhost/not-squashed:latest] 419 MB ①
[localhost/squashed:latest] 419 MB ②
[localhost/squashed-all:latest] 394 MB ③
```

- ① The base image size is 419MB.
- ② When Podman squashed the image layers, the image size stayed the same but the number of layers is lower.
- ③ When Podman squashed the layers of the container image and its parent image, the size reduced by 25MB.

Developers commonly reduce the number of layers by using multistage builds in combination with chaining some commands.



## References

### **Podman Official Documentation**

<https://docs.podman.io/en/latest/markdown/podman-build.1.html>

### **Containerfile Documentation**

<https://github.com/containers/common/blob/main/docs/Containerfile.5.md>

### **Exploring the new Podman secret command**

<https://www.redhat.com/sysadmin/new-podman-secrets-command>

`podman-secret-create(1)` man page

## ► Guided Exercise

# Build Images with Advanced Containerfile Instructions

Use a simple Python application to create a Containerfile with advanced instructions.

## Outcomes

You should be able to work with:

- Multistage builds.
- The `USER` instruction.
- The `WORKDIR` instruction.
- The `ENV` instruction.
- The `VOLUME` instruction.

## Before You Begin

As the `student` user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

```
[student@workstation ~]$ lab start custom-advanced
```

## Instructions

### ► 1. Examine the exercise application.

- 1.1. Navigate to the `~/D0188/labs/custom-advanced` directory.

```
[student@workstation ~]$ cd ~/D0188/labs/custom-advanced
no output expected
```

- 1.2. Examine the `main.py` file, which contains a basic Python application. The application reads the `numbers.txt` file, prints its content, and appends another number to the file. The application uses an environment variable to locate the `numbers.txt` file.

```
...output omitted...
```

```
FILE = environ.get('FILE')
```

```
...output omitted...
```

- 1.3. Examine the incomplete `Containerfile` file.

### ► 2. Use a multistage build to complete the `Containerfile` file.

**Chapter 4 |** Custom Container Images

Add a stage that uses the `registry.ocp4.example.com:8443/redhattraining/podman-random-numbers` base image. The base image contains the `random_generator.py` script. Use the script to generate a `numbers.txt` file and copy the file to the final stage.

- 2.1. Create a stage by adding a `FROM` instruction.

```
FROM registry.ocp4.example.com:8443/redhattraining/podman-random-numbers as
generator

FROM registry.ocp4.example.com:8443/ubi8/python-38:1-96
...output omitted...
```

- 2.2. Generate the `numbers.txt` file.

```
FROM registry.ocp4.example.com:8443/redhattraining/podman-random-numbers as
generator
RUN python3 random_generator.py

FROM registry.ocp4.example.com:8443/ubi8/python-38:1-96
...output omitted...
```

- 2.3. In the second build stage, copy the `numbers.txt` file by using the `--from=generator` option of the `COPY` instruction. Use the `--chown` option to make the user `default` the new owner.

```
...output omitted...
WORKDIR /redhat

COPY --from=generator --chown=default /app/numbers.txt materials/numbers.txt
COPY main.py .
...output omitted...
```

- 3. Add the `FILE` environment variable to set the path of the `numbers.txt` file.

```
FROM registry.ocp4.example.com:8443/redhattraining/podman-random-numbers as
generator
RUN python3 random_generator.py

FROM registry.ocp4.example.com:8443/ubi8/python-38:1-96

ENV FILE="/redhat/materials/numbers.txt"
USER default
WORKDIR /redhat

COPY --from=generator --chown=default /app/numbers.txt materials/numbers.txt
COPY main.py .

CMD python3 main.py
```

- 4. Build and test the image.

**Chapter 4 |** Custom Container Images

- 4.1. In a command-line terminal, build the image from the Containerfile.

```
[student@workstation custom-advanced]$ podman build -t \
redhat-local/custom-advanced .
...output omitted...
Successfully tagged localhost/redhat-local/custom-advanced:latest
3360...cc21
```

- 4.2. Run the image.

```
[student@workstation custom-advanced]$ podman run --rm \
--name=custom-advanced redhat-local/custom-advanced
Current content: ['17 72 97 8 32 15 63 97 57']
Writing another number...
Current content: ['17 72 97 8 32 15 63 97 57 4']
```

The application uses the environment variable to read from the file, and write content to the file. The numbers that you get might differ because they are generated randomly, however, a 4 is always appended to the end.

► **5.** Add a mount point to the /redhat/materials directory.

- 5.1. In the Containerfile, add a VOLUME instruction.

```
...output omitted...
COPY main.py .

VOLUME /redhat/materials

CMD python3 main.py
```

► **6.** Build and test the image.

- 6.1. In your command-line terminal, build the image from the Containerfile.

```
[student@workstation custom-advanced]$ podman build -t \
redhat-local/custom-advanced .
...output omitted...
Successfully tagged localhost/redhat-local/custom-advanced:latest
4872...0ee0
```

- 6.2. Create a container from the image.

```
[student@workstation custom-advanced]$ podman run --name=custom-advanced \
redhat-local/custom-advanced
Current content: ['46 37 98 39 70 53 81 44 59']
Writing another number...
Current content: ['46 37 98 39 70 53 81 44 59 4']
```

**Note**

The previous command does not have the `--rm` option. If you use the `--rm` option, then Podman removes the anonymous volumes that the container uses.

- 7. Read the `numbers.txt` file on your local file system.

- 7.1. Inspect the container to identify the anonymous volume path on the file system.

```
[student@workstation custom-advanced]$ podman inspect custom-advanced
...output omitted...
"Mounts": [
 {
 "Type": "volume",
 "Name": "3aa7...22ac",
 "Source": "/home/student/.local/share/containers/storage/volumes/3aa7...22ac/_data",
 "Destination": "/redhat/materials",
 "Driver": "local",
...output omitted...
```

You can limit the output of the `podman inspect` command by using the `--format` option.

```
[student@workstation custom-advanced]$ podman inspect custom-advanced \
--format="{{ (index .Mounts 0).Source }}"
/home/student/.local/share/containers/storage/volumes/40ed...96bf/_data
```

- 7.2. Use the `Source` path to read the `numbers.txt` file from your local file system.

```
[student@workstation custom-advanced]$ cat SOURCE/numbers.txt
46 37 98 39 70 53 81 44 59 4
```

## Finish

On the `workstation` machine, use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish custom-advanced
```

# Rootless Podman

---

## Objectives

- Run rootless containers with Podman.

## Container Workload Isolation

With container technology becoming increasingly popular, the structure of deploying applications is rapidly changing. Previously, a single application might be separated into tiers, such as front end, API or back end, and database tiers. Developers often deployed each of these application parts to virtualized machines that were dedicated to a single application.

Virtualized machines then ran an operating system with a kernel separate from the host operating system. This meant that one physical machine could host several virtual machines but each of the virtual machines contained a kernel and typically only a single application.

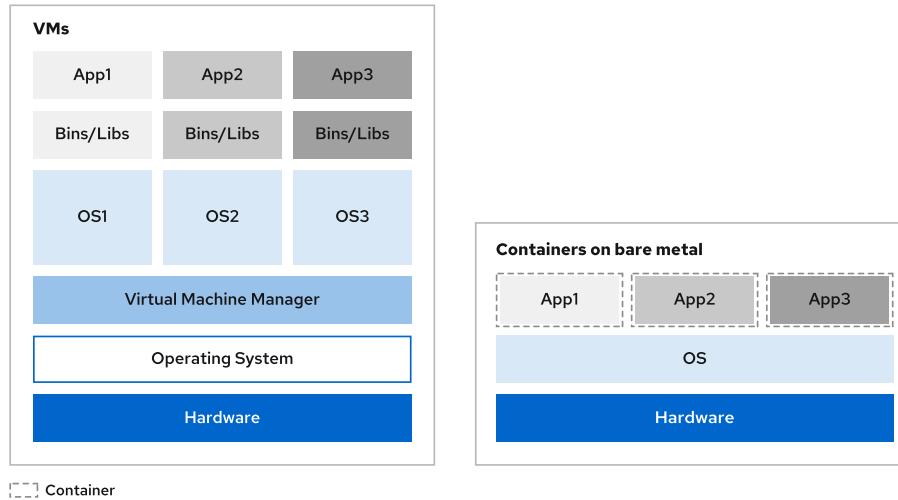


Figure 4.1: Container and VM application isolation

Container technology encourages developers to split a single application tier, such as back end, into multiple smaller microservices. Additionally, containerized processes use the kernel of the host operating system. Typically, due to the smaller resource requirements of a single containerized process, one host can run more containerized processes than it can virtual machines.

Because one machine can host multiple applications, exploiting a single application to gain superuser privilege on the host machine can lead to large-scale outages. Consequently, developers and administrators strive to create and maintain applications that use the principle of least privilege, which minimizes potential attack vectors.

## Analyzing Rootless Containers

*Rootless containers*, or unprivileged containers, are containers that do not require administrator privileges.

A container is rootless only when it meets the following conditions:

- The containerized process does not use the root user, which is a special privileged user in Linux and UNIX systems. Such a user is for administrative purposes and has the ID 0.
- The root user inside of the container is not the root user outside of the container.
- The container runtime does not use the root user. For example, if your container runtime runs as the root user, then containers managed by such runtime are not rootless containers.

Because Podman starts each container as a new process, the runtime does not require elevated privileges.



#### Note

The Podman process exits after creating a container. Then the container process ID attaches to the systemd parent process ID.

The following subsections examine the remaining requirements of rootless containers.

## Prerequisites for Rootless Containers

Depending on your operating system, Podman might require host operating system setup to run rootless containers. The following list briefly describes the most common prerequisite setup.

### cgroup v2

Cgroup v2 is a Linux kernel feature that Podman uses to limit container resource use without requiring elevating privileges.

Podman on Red Hat Enterprise Linux 9 (RHEL 9) uses cgroup v2 by default, with the runc container runtime implementation.

### slirp4netns

Podman uses the slirp4netns package to implement user-mode networking for unprivileged network namespaces.

### fuse-overlays

Podman uses the fuse-overlays package to manage the Copy-On-Write (COW) file system. Though Podman does not require this package, Red Hat recommends using it for performance reasons. COW file system is discussed later in the course.

See the references section for more information about rootless Podman setup.

## Changing the Container User

When you create a Containerfile, the user tends to be root. This is because you require elevated privileges for certain operations, such as installing packages or making configuration changes.

Determine the current user by running the id command:

```
[user@host ~]$ podman run registry.access.redhat.com/ubi9/ubi id
uid=0(root) gid=0(root) groups=0(root)
```

The following container image uses the root user to start an HTTP server:

```
FROM registry.access.redhat.com/ubi9/ubi

CMD ["python3", "-m", "http.server"]
```

This is a security risk, because an attacker could exploit the application, get access to the container, and exploit further vulnerabilities to escape from the containerized environment into the host system. Attackers might escape the containerized environment by exploiting bugs and vulnerabilities typically in the kernel or container runtime, such as runc.

Consider the following Containerfile change:

```
FROM registry.access.redhat.com/ubi9/ubi

RUN adduser \
 --no-create-home \
 --system \
 --shell /usr/sbin/nologin \
 python-server

USER python-server

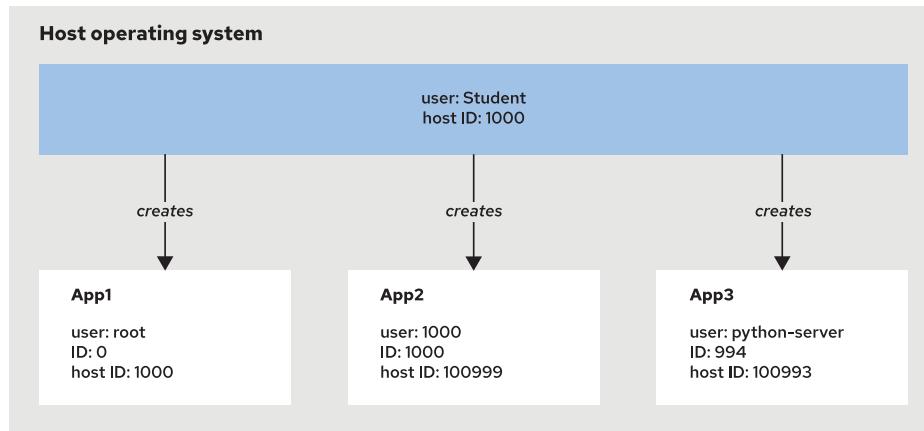
CMD ["python3", "-m", "http.server"]
```

Such a container starts the same process without elevated privileges, which makes it harder for an attacker to exploit a vulnerability and access the host system. The RUN instruction runs as the root user, because it precedes the USER instruction.

## Explaining User Mapping

Traditionally, when an attacker gains access to the container file system by using an exploit, the root user inside the container corresponds to the root user outside of the container. This means that if an attacker escapes the container isolation, then they have elevated privileges on the host system, thus potentially causing more damage.

Podman maps users inside of the container to unprivileged users on the host system by using *subordinate ID* ranges. Podman defines the allowed ID ranges in the */etc/subuid* and */etc/subgid* files.



**Figure 4.2: Examples of users with different IDs inside and outside of containers**

Consider the following example:

```
[user@host ~]$ cat /etc/subuid /etc/subgid
student:100000:65536
student:100000:65536
```

On the system from the preceding example, the `student` user can allocate 65536 user IDs starting with the ID 100000. This leads to the `HostUserID = 100000 + ContainerUserID - 1` ID mapping. Consequently, user ID 1 in a container maps to the host user ID 100000, and so on.

The user ID 0 (`root`) is an exception, because the `root` user maps to the user ID that started the container. For example, if a user with the ID 1000 starts a container that uses the `root` user, then the `root` user maps to the host user ID 1000.

To generate the subordinate ID ranges, use the `usermod` command.

```
[user@host ~]$ sudo usermod --add-subuids 100000-165535 \
--add-subgids 100000-165535 student
[user@host ~]$ grep student /etc/subuid /etc/subgid
/etc/subuid:student:100000:65536
/etc/subgid:student:100000:65536
```

The `/etc/subuid` and `/etc/subgid` files must exist before you define the subordinate ID ranges with the `usermod` command. After you define the ranges, you must execute the `podman system migrate` for the new subordinate ID ranges to take effect.

You can verify the mapped user by using the `podman top` command. For example, the following command starts a container and uses the `id` command to verify that the user inside of the container is `root`.

```
[user@host ~]$ podman run -it registry.access.redhat.com/ubi9/ubi bash
[root@e6116477c5c9 /]# id
uid=0(root) gid=0(root) groups=0(root)
```

Then, you must verify the mapping of host user (huser) to the container user (user). The following example uses the container ID e6116477c5c9:

```
[user@host ~]$ podman top e6116477c5c9 huser user
HUSER USER
1000 root
```

The preceding example shows that the user inside the container, `root`, is mapped to a user with ID `1000` on the host system.

Alternatively, you can verify the same ID mapping by printing the `/proc/self/uid_map` and `/proc/self/gid_map` files inside of the container:

```
[root@e6116477c5c9 /]# cat /proc/self/uid_map /proc/self/gid_map
0 1000 1
0 1000 1
```



### Warning

When you execute a container with elevated privileges on the host machine, the root mapping does not take place even when you define subordinate ID ranges, for example:

```
[user@host ~]$ sudo podman run -it registry.access.redhat.com/ubi9/ubi bash
[root@4746207beab7 /]# id
uid=0(root) gid=0(root) groups=0(root)
```

In a new terminal, verify the `podman top` output:

```
[user@host ~]$ sudo podman top 4746207beab7 huser user
HUSER USER
root root
```

## Limitations of Rootless Containers

Rootless containers have limitations that make some applications unsuitable to be containerized as rootless containers. The following list describes some limitations of rootless containers.

### Non-trivial Containerization

Some applications might require the root user. Depending on the application architecture, some applications might not be suitable for rootless containers or might require a deeper understanding to containerize.

For example, applications such as HTTPd and Nginx start a bootstrap process and then spawn a new process with a non-privileged user, which interacts with external users. Such applications are non-trivial to containerize for rootless use.

Red Hat provides containerized versions of HTTPd and Nginx that do not require root privileges for production usage. You can find the containers in the Red Hat container registry [<https://catalog.redhat.com/software/containers/explore>].

## Required Use of Privileged Ports or Utilities

Rootless containers cannot bind to privileged ports, such as ports 80 or 443. Red Hat recommends that you do not use privileged ports, and use port forwarding instead. However, if you require the use of privileged ports, then you can configure the unprivileged port range:

```
[user@host ~]$ sudo sysctl -w "net.ipv4.ip_unprivileged_port_start=79"
```

This means rootless containers can bind to port 80 and higher.

Similarly, rootless containers cannot use utilities that require the root user, such as the `ping` utility. This is because the `ping` utility requires elevated privileges to establish raw sockets, which is an action that requires the `cap_net_raw` privilege.

To solve such a requirement, verify whether you can grant the privilege to a non-root user. For example, you can specify a range of group IDs that are allowed to use the `ping` utility by using the `net.ipv4.ping_group_range` kernel parameter:

```
[user@host ~]$ sudo sysctl -w "net.ipv4.ping_group_range=0 2000000"
```



### References

#### **Understanding root inside and outside a container**

<https://www.redhat.com/en/blog/understanding-root-inside-and-outside-container>

#### **Application Container Security Guide**

<https://nvlpubs.nist.gov/nistpubs/specialpublications/nist.sp.800-190.pdf>

#### **Podman Setup**

[https://github.com/containers/podman/blob/v4.4.1/docs/tutorials/rootless\\_tutorial.md](https://github.com/containers/podman/blob/v4.4.1/docs/tutorials/rootless_tutorial.md)

#### **Podman Troubleshooting**

<https://github.com/containers/podman/blob/v4.4.1/troubleshooting.md>

#### **Shortcomings of Rootless Podman**

<https://github.com/containers/podman/blob/v4.4.1/rootless.md>

#### **Container Security Workshop**

[http://redhatgov.io/workshops/security\\_containers/](http://redhatgov.io/workshops/security_containers/)

## ► Guided Exercise

# Rootless Podman

Analyze a non-functioning rootless Podman environment.

## Outcomes

You should be able to:

- Analyze the symptoms of a Podman environment that lacks group and user ID mapping.
- Provide the group and user ID mapping to fix the environment issues.

## Before You Begin

As the student user on the workstation machine, use the `lab` command to prepare your system for this exercise.

```
[student@workstation ~]$ lab start custom-rootless
```

## Instructions

► 1. Explore and build the Gitea container image.

- Change into the `$HOME/D0188/labs/custom-rootless/gitea` directory.

```
[student@workstation ~]$ cd $HOME/D0188/labs/custom-rootless/gitea
no output expected
```

- Explore the `Containerfile` file. Note that the image creates and uses the `default` user.

- Build the container image with the `localhost/gitea` tag.

```
[student@workstation gitea]$ podman build -t gitea .
...output omitted...
Successfully tagged localhost/gitea:latest
e92a...390b
```

- Repeat the last command for the `root` user. Use the double exclamation mark (`!!`) to refer to the previously executed command.

```
[student@workstation gitea]$ sudo !!
...output omitted..
Successfully tagged localhost/gitea:latest
a247...2e58
```

This guided exercise requires that you use the `gitea` image to start a container by using the `root` user in a later step. Consequently, because the `gitea` image is local

to your environment, you must build the gitea image for the root user. Users do not share local images by default.

- ▶ 2. As the root user, execute the \$HOME/D0188/labs/custom-rootless/ids.sh script to remove the user and group ID system mapping.

```
[student@workstation ~]$ sudo $HOME/D0188/labs/custom-rootless/ids.sh
Backing up /etc/subgid and /etc/subuid
OK
```

- ▶ 3. Start the containerized Gitea process.

- 3.1. Start a container that uses the gitea image in rootless mode.

```
[student@workstation gitea]$ podman run --rm -p 3030:3030 gitea
ERRO[0000] cannot find UID/GID for user student: cannot read subids - check
rootless mode in man pages.
WARN[0000] Using rootless single mapping into the namespace. This might break
some images. Check /etc/subuid and /etc/subgid for adding sub*ids if not using a
network user
Error: OCI runtime error: runc: container_linux.go:380: starting container process
caused: setup user: invalid argument
```

Podman fails to start the container because the image uses multiple users and Podman does not have enough ID mapping to map the user and group IDs inside the container to non-root users and group IDs on your system.

- 3.2. Start a container that uses the gitea image as the root user.

```
[student@workstation gitea]$ sudo podman run --name root-gitea \
-p 3030:3030 --rm gitea
...output omitted...
2022/06/02 09:22:11 ...s/graceful/server.go:61>NewServer() [I] Starting new Web
server: tcp:0.0.0.0:3030 on PID: 1
```

- 3.3. In a new terminal, verify the user and group ID mapping.

```
[student@workstation ~]$ sudo podman exec root-gitea \
cat /proc/self/uid_map /proc/self/gid_map
0 0 4294967295
0 0 4294967295
```

When you start the container as root, Podman uses your system's root user for the root user inside the container. Consequently, if an attacker gains root permissions in your container, then they can potentially access your host system.

- 3.4. Stop the root-gitea container.

```
[student@workstation ~]$ sudo podman stop root-gitea
root-gitea
```

- ▶ 4. Provide the subuid and subguid ID ranges.

- 4.1. Create the /etc/subuid and /etc/subgid files.

```
[student@workstation ~]$ sudo touch /etc/{subuid,subgid}
no output expected
```

- 4.2. Starting with the ID 100000, add 65536 IDs to map for the student user.

```
[student@workstation ~]$ sudo usermod --add-subuids 100000-165536 \
--add-subgids 100000-165536 student
no output expected
```

- 4.3. Verify the ID ranges.

```
[student@workstation ~]$ cat /etc/subuid /etc/subgid
student:100000:65537
student:100000:65537
```

5. Start a container that uses the gitea image in a rootless mode.

- 5.1. Attempt to start the gitea container.

```
[student@workstation ~]$ podman run --rm -p 3030:3030 --name gitea gitea
Error: OCI runtime error: runc: container_linux.go:380: starting container process
caused: setup user: invalid argument
```

The error occurs because you did not inform the Podman runtime about the new ID ranges.

- 5.2. Migrate the Podman ID ranges.

```
[student@workstation ~]$ podman system migrate
no output expected
```

- 5.3. Start a container that uses the gitea image. Expose port 3030 to access the application.

```
[student@workstation ~]$ podman run --name gitea --rm -p 3030:3030 gitea
...output omitted...
2022/06/03 11:29:49 cmd/web.go:212:listen() [I] AppURL(ROOT_URL): http://
localhost:3030/
2022/06/03 11:29:49 ...s/graceful/server.go:61>NewServer() [I] Starting new Web
server: tcp:0.0.0.0:3030 on PID: 1
```

- 5.4. Navigate to localhost:3030 in a web browser to verify the application functionality.

- 5.5. Terminate the gitea container.

```
[student@workstation ~]$ podman stop gitea
gitea
```

- 6. Restore the original user and group IDs by using the \$HOME/DO188/labs/custom-rootless/ids.sh script.

```
[student@workstation ~]$ sudo $HOME/DO188/labs/custom-rootless/ids.sh
Restoring /etc/subgid and /etc/subuid
OK
```

## Finish

On the **workstation** machine, use the **lab** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish custom-rootless
```

## ► Lab

# Custom Container Images

Complete the Containerfile for an application that generates a QR code from a given text.

## Outcomes

You should be able to:

- Understand multistage builds.
- Run commands within a container.
- Set environment variables.
- Set a working directory.
- Set an entry point.
- Change the user that executes commands.

## Before You Begin

As the student user on the workstation machine, use the `lab` command to prepare your system for this exercise.

```
[student@workstation ~]$ lab start custom-lab
```

The `start` command copies a Node.js application, which generates a QR code from a given text, to the `labs/custom-lab` directory of your workspace. The command also generates an `.npmrc` file that configures the Node.js application to use an internal NPM registry.

The `lab` script continuously evaluates the objectives of this lab. Keep the script running in a terminal window and complete the objectives of this lab from a new terminal window.

The application contains a `Containerfile` that you must complete throughout this exercise. The `Containerfile` uses a multistage build. The first stage uses the `registry.ocp4.example.com:8443/redhattraining/podman-certificate-generator` image to generate self-signed certificates.

In the second stage, the application uses the certificates to enable a TLS connection.

## Instructions

1. Navigate to the `~/D0188/labs/custom-lab` directory, which contains the application that converts a text into a QR code image. Then, run the app on the host machine by using the `npm install` and `npm start` commands. Verify that the application fails gracefully because an environment variable is missing.
  - 1.1. Navigate to the `~/D0188/labs/custom-lab` directory.

```
[student@workstation ~]$ cd ~/DO188/labs/custom-lab
no output expected
```

- 1.2. Install the application dependencies.

```
[student@workstation custom-lab]$ npm install
added 201 packages, and audited 202 packages in 1s
...output omitted...
```

- 1.3. Start the application. The application exits because the HTTP port is not set.

```
[student@workstation custom-lab]$ npm start
> custom-images-lab@1.0.0 start
> node index.js

HTTP PORT not found. Set the env variable to proceed.
```

2. In the build stage of the Containerfile, generate the TLS certificates by using the `./gen_certificates.sh` command.

The `./gen_certificates.sh` command is included in the provided container.

3. In the final stage of the Containerfile, set the following environment variables:

- `TLS_PORT=8443` (the port for TLS traffic)
- `HTTP_PORT=8080` (the port for HTTP traffic)
- `CERTS_PATH=/etc/pki/tls/private/certs` (the path that contains the TLS certificates)

Build the container image with the name `localhost/podman-qr-app`.

4. In the final stage of the Containerfile, set the working directory of the application to the `/app` path.

Then, build the container image with the name `localhost/podman-qr-app`.

5. In the final stage of the Containerfile, set the `student` user as the user that runs the application. The `student` user exists in the Containerfile.

Then, build the container image with the name `localhost/podman-qr-app`.

6. In the final stage of the Containerfile, run the `npm install --omit=dev` command to install the production dependencies of the Node.js application.

Then, build the container image with the name `localhost/podman-qr-app`.

7. In the final stage of the Containerfile, make `npm start` the default command for this image. Additional runtime arguments should not override the default command.

Then, build the container image with the name `localhost/podman-qr-app`.

8. Start the `podman-qr-app` container. Call the container `custom-lab` and expose ports `8080` and `8443`.

## Finish

As the **student** user on the **workstation** machine, use the **lab** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

Press **y** when the **lab start** command prompts you to execute the **finish** function. Alternatively, execute the following command:

```
[student@workstation ~]$ lab finish custom-lab
```

## ► Solution

# Custom Container Images

Complete the Containerfile for an application that generates a QR code from a given text.

### Outcomes

You should be able to:

- Understand multistage builds.
- Run commands within a container.
- Set environment variables.
- Set a working directory.
- Set an entry point.
- Change the user that executes commands.

### Before You Begin

As the student user on the workstation machine, use the `lab` command to prepare your system for this exercise.

```
[student@workstation ~]$ lab start custom-lab
```

The `start` command copies a Node.js application, which generates a QR code from a given text, to the `labs/custom-lab` directory of your workspace. The command also generates an `.npmrc` file that configures the Node.js application to use an internal NPM registry.

The `lab` script continuously evaluates the objectives of this lab. Keep the script running in a terminal window and complete the objectives of this lab from a new terminal window.

The application contains a `Containerfile` that you must complete throughout this exercise. The `Containerfile` uses a multistage build. The first stage uses the `registry.ocp4.example.com:8443/redhattraining/podman-certificate-generator` image to generate self-signed certificates.

In the second stage, the application uses the certificates to enable a TLS connection.

### Instructions

1. Navigate to the `~/D0188/labs/custom-lab` directory, which contains the application that converts a text into a QR code image. Then, run the app on the host machine by using the `npm install` and `npm start` commands. Verify that the application fails gracefully because an environment variable is missing.
  - 1.1. Navigate to the `~/D0188/labs/custom-lab` directory.

```
[student@workstation ~]$ cd ~/DO188/labs/custom-lab
no output expected
```

- 1.2. Install the application dependencies.

```
[student@workstation custom-lab]$ npm install
added 201 packages, and audited 202 packages in 1s
...output omitted...
```

- 1.3. Start the application. The application exits because the HTTP port is not set.

```
[student@workstation custom-lab]$ npm start
> custom-images-lab@1.0.0 start
> node index.js

HTTP PORT not found. Set the env variable to proceed.
```

2. In the build stage of the Containerfile, generate the TLS certificates by using the `./gen_certificates.sh` command.

The `./gen_certificates.sh` command is included in the provided container.

- 2.1. Use the RUN instruction to generate the TLS certificates.

```
FROM registry.ocp4.example.com:8443/redhattraining/podman-certificate-generator as
certs

RUN ./gen_certificates.sh

FROM registry.ocp4.example.com:8443/ubi9/nodejs-18:1
USER root
RUN groupadd -r student && useradd -r -m -g student student && \
 npm config set cache /tmp/.npm --global

COPY --from=certs --chown=student:student /app/*.pem /etc/pki/tls/private/certs/
COPY --chown=student:student . /app/
```

3. In the final stage of the Containerfile, set the following environment variables:

- `TLS_PORT=8443` (the port for TLS traffic)
- `HTTP_PORT=8080` (the port for HTTP traffic)
- `CERTS_PATH=/etc/pki/tls/private/certs` (the path that contains the TLS certificates)

Build the container image with the name `localhost/podman-qr-app`.

- 3.1. Use the ENV instruction to add the environment variables to the Containerfile.

```

FROM registry.ocp4.example.com:8443/redhattraining/podman-certificate-generator as
certs

RUN ./gen_certificates.sh

FROM registry.ocp4.example.com:8443/ubi9/nodejs-18:1
USER root
RUN groupadd -r student && useradd -r -m -g student student && \
 npm config set cache /tmp/.npm --global

COPY --from=certs --chown=student:student /app/*.pem /etc/pki/tls/private/certs/
COPY --chown=student:student . /app/

ENV TLS_PORT=8443 \
 HTTP_PORT=8080 \
 CERTS_PATH="/etc/pki/tls/private/certs"

```

3.2. Build the container image.

```

[student@workstation custom-lab]$ podman build -t localhost/podman-qr-app .
...output omitted...
Successfully tagged localhost/podman-qr-app:latest
201...cc8

```

4. In the final stage of the Containerfile, set the working directory of the application to the /app path.

Then, build the container image with the name localhost/podman-qr-app.

4.1. Use the WORKDIR instruction to define the working directory.

```

FROM registry.ocp4.example.com:8443/redhattraining/podman-certificate-generator as
certs

RUN ./gen_certificates.sh

FROM registry.ocp4.example.com:8443/ubi9/nodejs-18:1
USER root
RUN groupadd -r student && useradd -r -m -g student student && \
 npm config set cache /tmp/.npm --global

COPY --from=certs --chown=student:student /app/*.pem /etc/pki/tls/private/certs/
COPY --chown=student:student . /app/

ENV TLS_PORT=8443 \
 HTTP_PORT=8080 \
 CERTS_PATH="/etc/pki/tls/private/certs"

WORKDIR /app

```

4.2. Build the container image.

```
[student@workstation custom-lab]$ podman build -t localhost/podman-qr-app .
...output omitted...
Successfully tagged localhost/podman-qr-app:latest
201...cc8
```

5. In the final stage of the Containerfile, set the `student` user as the user that runs the application. The `student` user exists in the Containerfile.

Then, build the container image with the name `localhost/podman-qr-app`.

- 5.1. Use the `USER` instruction.

```
FROM registry.ocp4.example.com:8443/redhattraining/podman-certificate-generator as
certs

RUN ./gen_certificates.sh

FROM registry.ocp4.example.com:8443/ubi9/nodejs-18:1
USER root
RUN groupadd -r student && useradd -r -m -g student student && \
 npm config set cache /tmp/.npm --global

COPY --from=certs --chown=student:student /app/*.pem /etc/pki/tls/private/certs/
COPY --chown=student:student . /app/

ENV TLS_PORT=8443 \
 HTTP_PORT=8080 \
 CERTS_PATH="/etc/pki/tls/private/certs"

WORKDIR /app

USER student
```

- 5.2. Build the container image.

```
[student@workstation custom-lab]$ podman build -t localhost/podman-qr-app .
...output omitted...
Successfully tagged localhost/podman-qr-app:latest
201a...ecc8
```

6. In the final stage of the Containerfile, run the `npm install --omit=dev` command to install the production dependencies of the Node.js application.

Then, build the container image with the name `localhost/podman-qr-app`.

- 6.1. Use the `RUN` instruction to execute the command.

```
FROM registry.ocp4.example.com:8443/redhattraining/podman-certificate-generator as
certs

RUN ./gen_certificates.sh

FROM registry.ocp4.example.com:8443/ubi9/nodejs-18:1
```

```

USER root
RUN groupadd -r student && useradd -r -m -g student student && \
 npm config set cache /tmp/.npm --global

COPY --from=certs --chown=student:student /app/*.pem /etc/pki/tls/private/certs/
COPY --chown=student:student . /app/

ENV TLS_PORT=8443 \
 HTTP_PORT=8080 \
 CERTS_PATH="/etc/pki/tls/private/certs"

WORKDIR /app

USER student

RUN npm install --omit=dev

```

## 6.2. Build the container image.

```

[student@workstation custom-lab]$ podman build -t localhost/podman-qr-app .
...output omitted...
Successfully tagged localhost/podman-qr-app:latest
201...cc8

```

- In the final stage of the Containerfile, make `npm start` the default command for this image. Additional runtime arguments should not override the default command.

Then, build the container image with the name `localhost/podman-qr-app`.

- Use the `ENTRYPOINT` instruction to execute the command when the container is started.

```

FROM registry.ocp4.example.com:8443/redhattraining/podman-certificate-generator as
certs

RUN ./gen_certificates.sh

FROM registry.ocp4.example.com:8443/ubi9/nodejs-18:1
USER root
RUN groupadd -r student && useradd -r -m -g student student && \
 npm config set cache /tmp/.npm --global

COPY --from=certs --chown=student:student /app/*.pem /etc/pki/tls/private/certs/
COPY --chown=student:student . /app/

ENV TLS_PORT=8443 \
 HTTP_PORT=8080 \
 CERTS_PATH="/etc/pki/tls/private/certs"

WORKDIR /app

USER student

```

```
RUN npm install --omit=dev

ENTRYPOINT npm start
```

7.2. Build the container image.

```
[student@workstation custom-lab]$ podman build -t localhost/podman-qr-app .
...output omitted...
Successfully tagged localhost/podman-qr-app:latest
201...cc8
```

8. Start the podman-qr-app container. Call the container `custom-lab` and expose ports 8080 and 8443.

8.1. Use the `podman run` command to start the application and bind the corresponding ports.

```
[student@workstation custom-lab]$ podman run --name=custom-lab \
-p 8080:8080 -p 8443:8443 podman-qr-app
...output omitted...
TLS Server running on port 8443
Server running on port 8080
```

8.2. Optionally, test the application by navigating to `http://localhost:8080` in a web browser.

## Finish

As the `student` user on the `workstation` machine, use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

Press `y` when the `lab start` command prompts you to execute the `finish` function. Alternatively, execute the following command:

```
[student@workstation ~]$ lab finish custom-lab
```

# Summary

---

- Containerize a process by choosing a base image and by using the Containerfile instructions.
- Create rootless containers by creating Containerfiles that do not run as a privileged user.
- You can implement advanced container build and runtime patterns, which includes:
  - The use of the ENV instruction to create containers that can be deployed in different environments.
  - The use of the VOLUME instruction to store container data outside the container layered file system.
  - The use of ENTRYPPOINT, CMD, and their interactions.
  - Creating multistage Containerfiles to build smaller images that do not contain build-time dependencies.

## Chapter 5

# Persisting Data

### Goal

Run database containers with persistence.

### Objectives

- Describe the process for mounting volumes and common use cases.
- Build containerized databases.

### Sections

- Volume Mounting (and Guided Exercise)
- Working with Databases (and Guided Exercise)

### Lab

- Persisting Data

# Volume Mounting

## Objectives

- Describe the process for mounting volumes and common use cases.

## Copy-on-write (COW) File System

When you build a container image, each instruction that modifies the container file system creates a new read-only data layer. Because you cannot modify data in an existing layer, each layer contains a set of changes, or *diffs*, from the previous layer.

Consider the following Containerfile:

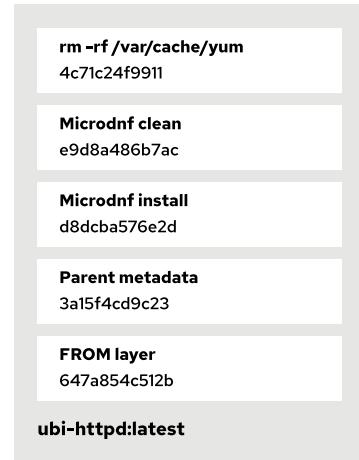
```
FROM registry.access.redhat.com/ubi8/ubi-minimal

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

CMD httpd -DFOREGROUND
```

You can inspect the image layers by using the podman `image tree` command:

```
[user@host ~]$ podman image tree ubi-httpd
Image ID: bdd298c12db7
Tags: [localhost/ubi-httpd:latest]
Size: 166.8MB
Image Layers
└─ ID: 647a854c512b Size: 94.77MB
└─ ID: 3a15f4cd9c23 Size: 20.48kB Top Layer of: [registry.access.redhat.com/ubi8/
ubi-minimal:latest]
└─ ID: d8dcba576e2d Size: 68.45MB
└─ ID: e9d8a486b7ac Size: 3.581MB
└─ ID: 4c71c24f9911 Size: 4.608kB Top Layer of: [localhost/ubi-httpd:latest]
```



**Figure 5.1: The ubi-httppd container layers**

Note that because the `microdnf clean all` command creates a new layer, the resulting container image still contains the data that the `microdnf clean all` command removed. Though not accessible at runtime, the data is contained in the previous layer, the `d8dcba576e2d` layer, and contributes to the overall container size.

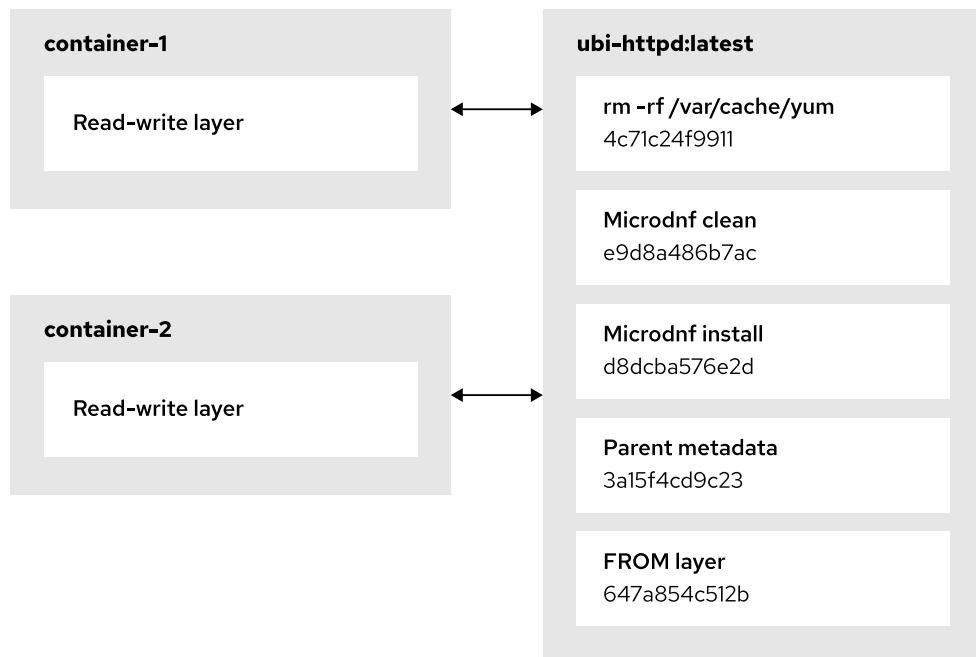
To remove unnecessary data from the `microdnf install` command, use the `microdnf clean all` and the `rm -rf /var/cache/yum` commands in the same container layer. For example, chain the commands in one RUN instruction:

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

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

CMD httpd -DFOREGROUND
```

When you instantiate the container image, for example by using the `podman run` command, Podman creates a *thin* read/write container layer on top of the previous layers. This means that multiple containers that use one container image share read-only layers and differ in the runtime read/write layer. When you delete a container, Podman destroys the read/write layer. This means container runtime data is ephemeral.



**Figure 5.2: Multiple containers share immutable read-only layers**

See the references section for more information about Podman's implementation of the overlay union file system.

## Implications of a COW File System

Files that are in the container image layers are available to the read/write data layer by using a union file system. At runtime, the container file system consists of a union of files in the defined layers.

Modifying a file or its attributes at runtime is possible in several steps:

- Podman locates the requested file in the closest (highest) layer and copies it to the runtime layer.
- In the runtime layer, when the file is available for both read and write operations, a containerized process can modify the file.

The implementation of how files behave when a process attempts to read or write a file depends on the storage driver.

Using a union file systems provides efficient read operations. The COW data architecture also promotes the sharing and reuse of data layers between separate containers, because the layers are immutable.

However, union file systems introduce a performance bottleneck for write-intensive containerized processes.

## Store Data on Host Machine

Developers often write applications that require storing data persistently. For such cases, Podman implements external mounts by using *volumes* and *bind mounts*.

This is useful for the following reasons:

## Persistence

Mounted data is persistent across container deletions. Because containers are ephemeral, data from the runtime read/write layer is not accessible after container deletion.

## Use of Host File System

Mounted data typically does not implement the COW file system. Consequently, write-heavy containerized processes can write data to a mount without the limitations of the COW file systems for write operations.

## Ease of Sharing

Mounted data can be shared between multiple containers at the same time. This means that one container can write to a mount and another container can read the same data.

Additionally, mounts are not limited to the container host machine. You can host data mounts over the network, for example by using the NFS protocol.

Volumes are data mounts managed by Podman. Bind mounts are data mounts managed by the user.

Both volumes and bind mounts can use the `--volume` (or `-v`) parameter.

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

In the preceding syntax, the `:OPTIONS` part is optional. Note that you can specify host paths by using absolute paths, such as `/home/user/www`, or relative paths, such as `./www`, which refers to the `www` directory in the current working directory.

Use `-v volume_name:/path/in/container` to refer to a volume.

Alternatively, you can use the `--mount` parameter with the following syntax:

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

The `--mount` parameter explicitly specifies the volume type, such as:

- `bind` for bind mounts.
- `volume` for volume mounts.
- `tmpfs` for creating memory-only, ephemeral mounts.

The `--mount` parameter is the preferred way of mounting directories in a container. However, because the `-v` parameter is still widely used, this course uses both styles.

Developers commonly use bind mounts for testing, or for mounting environment-specific files, such as property files. Because Podman manages the volume file system, use volumes to ensure consistent mount behavior across systems. Additionally, use volumes for advanced uses, such as mounting a remote volume over NFS.

## Storing Data with Bind Mounts

Bind mounts can exist anywhere on the host system.

For example, to mount the `/www` directory on your host machine to the `/var/www/html` directory inside the container with the read-only option, use the following `podman run` command:

```
[user@host ~]$ podman run -p 8080:8080 --volume /www:/var/www/html:ro \
registry.access.redhat.com/ubi8/httpd-24:latest
```

## Troubleshoot Bind Mounts

When you use bind mounts, you must configure file permissions and SELinux access manually. SELinux is an additional security mechanism used by Red Hat Enterprise Linux (RHEL) and other Linux distributions.

Consider the following bind mount example:

```
[user@host ~]$ podman run -p 8080:8080 --volume /www:/var/www/html \
registry.access.redhat.com/ubi8/httpd-24:latest
```

By default, the `Httpd` process has insufficient permissions to access the `/var/www/html` directory. This can be a file permission issue or an SELinux issue.

To troubleshoot file permission issues, use the `podman unshare` command to execute the `ls -l` command. The `podman unshare` command executes provided Linux commands in a new namespace such as the one Podman creates for the container. This process maps user IDs as they are mapped in a new container, which is useful for troubleshooting user permissions.

```
[user@host ~]$ podman unshare ls -l /www/
total 4
-rw-rw-r--. 1 root root 21 Jul 12 15:21 index.html
[user@host ~]$ podman unshare ls -ld /www/
drwxrwxr-x. 1 root root 20 Jul 12 15:21 /www/
```

The previous example reveals that the `/www` directory is accessible to all users:

- The `/www/index.html` file provides read permissions to all users.
- The `/www` directory is viewed as owned by the `root` user and the `root` group in a new namespace.
- The `/www` directory provides the execute permissions to all users, which gives all users access to the directory contents.

This means that the file and directory permissions are correct in the bind mount.

To troubleshoot SELinux permission issues, inspect the `/www` directory SELinux configuration by running the `ls` command with the `-Z` option. Use the `-d` option to print only the directory information.

```
[user@host ~]$ ls -Zd /www
system_u:object_r:default_t:s0:c228,c359 /www
```

The output shows the SELinux context label `system_u:object_r:default_t:s0:c228,c359`, which has the `default_t` type. A container must have the `container_file_t` SELinux type to have access to the bind mount. SELinux is out of scope for this course.

To fix the SELinux configuration, add the `:z` or `:Z` option to the bind mount:

## Chapter 5 | Persisting Data

- Lower case z lets different containers share access to a bind mount.
- Upper case Z provides the container with exclusive access to the bind mount.

```
[user@host ~]$ podman run -p 8080:8080 --volume /www:/var/www/html:z \
registry.access.redhat.com/ubi8/httpd-24:latest
```

After adding the corresponding option, run the ls -Zd command and notice the right SELinux type.

```
[user@host ~]$ ls -Zd /www
system_u:object_r:container_file_t:s0:c240,c717 /www
```

The container\_file\_t SELinux type allows the container to access the bind mount files.



### Important

Changing the SELinux label for system directories might lead to unexpected issues.

## Storing Data with Volumes

Volumes let Podman manage the data mounts. You can manage volumes by using the podman volume command.

To create a volume called http-data, use the following command:

```
[user@host ~]$ podman volume create http-data
d721d941960a2552459637da86c3074bbba12600079f5d58e62a11caf6a591b5
```

You can inspect the volume by using the podman volume inspect command:

```
[user@host ~]$ podman volume inspect http-data
[
 {
 "Name": "http-data",
 "Driver": "local",
 "Mountpoint": "/home/user/.local/share/containers/storage/volumes/http-
data/_data",
 "CreatedAt": "2022-07-12T17:10:12.709259987+02:00",
 "Labels": {},
 "Scope": "local",
 "Options": {}
 }
]
```

For rootless containers, Podman stores local volume data in the \$HOME/.local/share/containers/storage/volumes/ directory.

To mount the volume into a container, refer to the volume by the volume name:

```
[user@host ~]$ podman run -p 8080:8080 --volume http-data:/var/www/html \
registry.access.redhat.com/ubi8/httpd-24:latest
```

Because Podman manages the volume, you do not need to configure SELinux permissions.

## Exporting and Importing Data with Volumes

You can import data from a tar archive into an existing Podman volume by using the `podman volume import VOLUME_NAME ARCHIVE_NAME` command.

```
[user@host ~]$ podman volume import http_data web_data.tar.gz
...no output expected...
```

You can also export data from an existing Podman volume and save it as a tar archive on the local machine by using the `podman volume export VOLUME_NAME --output ARCHIVE_NAME` command.

```
[user@host ~]$ podman volume export http_data --output web_data.tar.gz
...no output expected...
```

## Storing Data with a tmpfs Mount

Some applications cannot use the default COW file system in a specific directory for performance reasons, but use persistence or data sharing for that directory.

For such cases, you can use the `tmpfs` mount type, which means that the data in a mount is ephemeral but does not use the COW file system:

```
[user@host ~]$ podman run -e POSTGRESQL_ADMIN_PASSWORD=redhat --network lab-net \
--mount type=tmpfs,tmpfs-size=512M,destination=/var/lib/pgsql/data \
registry.redhat.io/rhel9/postgresql-13:1
```



### References

#### Overlay.go

<https://github.com/containers/podman/blob/v4.4.1/vendor/github.com/containers/storage/drivers/overlay/overlay.go>

#### SELinux and Container File Permissions

<https://www.redhat.com/sysadmin/user-namespaces-selinux-rootless-containers>

#### Podman is gaining rootless overlay support

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

## ► Guided Exercise

# Volume Mounting

Inject files into a container by using a bind mount and a volume.

## Outcomes

You should be able to:

- Use bind mounts with your containers.
- Use `podman unshare` to troubleshoot permission issues with bind mounts.
- Create named volumes.
- Import files into named volumes.

## Before You Begin

As the `student` user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

```
[student@workstation ~]$ lab start persisting-mounting
```

## Instructions

- 1. Examine the Containerfile for the application.

```
[student@workstation ~]$ cat \
~/DO188/labs/persisting-mounting/podman-python-server/Containerfile
FROM registry.access.redhat.com/ubi9/ubi:9.0.0-1468

RUN adduser \
 --no-create-home \
 --system \
 --shell /usr/sbin/nologin \
 python-server && \
 mkdir /server && \
 chown -R 'python-server:python-server' /server

WORKDIR /server

USER python-server

CMD ["python3", "-m", "http.server"]
```

The resulting container image uses the `/server` directory as the web root directory for the Python HTTP server. The `registry.ocp4.example.com:8443/redhattraining/podman-python-server` container image is based on this Containerfile.

- 2. Copy the `index.html` file to the `~/www` directory.

The `~/www` directory serves as a bind mount that contains the HTML for the container.

```
[student@workstation ~]$ cp ~/DO188/labs/persisting-mounting/index.html ~/www
no output expected
```

- 3. Test the `podman-python-server` container with the `~/www` directory mounted as a bind mount.

3.1. Start a container with the following parameters:

- Bind the `~/www` directory on the host system to the `/server` directory inside the container.
  - Use the `:Z` option to set the correct SELinux label on the bind mount.
- Name the container `podman-server`.
- Use the `--rm` option.
- Use the `-ti` options to display container output.
- Use the `registry.ocp4.example.com:8443/redhattraining/podman-python-server` image.
- Bind the port `8000` on the local machine to port `8000` inside the container.

```
[student@workstation ~]$ podman run -ti --rm --name podman-server \
--volume ~/www:/server:Z -p 8000:8000 \
registry.ocp4.example.com:8443/redhattraining/podman-python-server
...output omitted...
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/)
```

3.2. In a web browser, navigate to `localhost:8000`. You are presented with an error.

3.3. Check the container logs:

```
10.0.2.100 - - [28/Jun/2022 13:21:08] code 404, message No permission to list
directory
10.0.2.100 - - [28/Jun/2022 13:21:08] "GET / HTTP/1.1" 404 -
...output omitted...
```

The container does not have permission to access the `index.html` file. Keep the container running.

- 4. Correct the permission for the `~/www` directory.

4.1. In a new terminal, verify the directory permissions in a new user namespace.

```
[student@workstation ~]$ podman unshare ls -l --directory ~/www
drwxrwx---. 1 root root 20 Jun 28 14:56 /home/student/www
```

From the perspective of a new container, the directory is owned by the `root` user and group, and other users have no permissions in the directory.

4.2. Verify the group ID inside of the `podman-server` container.

```
[student@workstation ~]$ podman run --rm \
registry.ocp4.example.com:8443/redhattraining/podman-python-server id
uid=994(python-server) gid=994(python-server) groups=994(python-server)
```

- 4.3. Change the group of the ~/www directory and its content to the python-server group ID.

```
[student@workstation ~]$ podman unshare chgrp -R 994 ~/www
no output expected
```

- 4.4. Verify the directory permissions in a new user namespace.

```
[student@workstation ~]$ podman unshare ls -ln --directory ~/www
drwxrwx--- 1 0 994 20 Jun 28 14:56 /home/student/www
```

- 5. Retest the podman-server container with the ~/www directory mounted as a bind mount.

- 5.1. In a web browser, access localhost:8000. You are presented with the index.html page.
- 5.2. Stop the container by pressing Ctrl+c.

- 6. Create a named volume with the index.html page.

- 6.1. Create a volume called html-vol.

```
[student@workstation ~]$ podman volume create html-vol
html-vol
```

- 6.2. Change into the persisting-mounting lab directory.

```
[student@workstation ~]$ cd ~/D0188/labs/persisting-mounting
no output expected
```

- 6.3. Import the index.tar.gz archive file, which contains index.html, into the html-vol volume.

```
[student@workstation persisting-mounting]$ podman volume import \
html-vol index.tar.gz
no output expected
```

- 7. Start a new container that uses the podman-python-server image. Use a volume mount instead of the bind mount.

- 7.1. Start the podman-server container.

Bind the html-vol volume as a read-only /server directory inside the container. The rest of the parameters remain the same.

```
[student@workstation ~]$ podman run -ti --rm --name podman-server -p 8000:8000 \
--mount 'type=volume,source=html-vol,destination=/server,ro' \
registry.ocp4.example.com:8443/redhattraining/podman-python-server
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/)
```

- 7.2. In a web browser, access `localhost:8000`. You are presented with the `index.html` page.
- 7.3. Stop the container by pressing `Ctrl+c`.

## Finish

On the `workstation` machine, use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish persisting-mounting
```

# Working with Databases

## Objectives

- Build containerized databases.

## Stateful Database Containers

Database containers typically maintain state because they must persist data. For this reason, they are called *stateful containers*. These containers differ from other types of containerized software, such as web APIs or proxies, which do not need to maintain state and therefore are called stateless containers.



### Note

Database containers can be stateless if there is no need to persist data. Memory-based caches or ephemeral testing databases are examples of stateless database containers.

Compared to stateless containers, stateful containers present the following challenges:

### Portability

The ability to move the container between different environments. Creating a container in the new environment is no longer enough and you must provide the container with up-to-date data.

### Scalability

The ability to improve performance by increasing the number of replicas of a container. Many database technologies are not designed to have data operated by more than one database process at a time.

### Availability

The ability to keep the system working in the event of one container crashing as long as that container has another replica running. Even though database systems include configurations for high availability, Podman alone does not provide the features needed for this type of configuration.

To achieve portability with stateful containers, you must provide them with a valid state across different environments. For example, your containers might need sample data for development and testing environments. In production, however, your containers typically require real, up-to-date data.

Scalability and availability are not requirements for development or testing environments. They are addressed in production environments by using features offered by orchestration systems such as Red Hat OpenShift, along with specialized database operators.

## Good Practices for Database Containers

The following practices provide benefits when you containerize database processes:

## Use the VOLUME instruction in Containerfiles

When you build a custom database container image, use the VOLUME instruction to create mount points at the database storage directories. Mount points avoid using the copy-on-write (COW) file system, which does not perform well with stateful data.

When you create a container from an image that uses the VOLUME instruction, Podman creates an anonymous volume and attaches the volume to the container.

## Mount the data directory to a named volume

Use a named volume to mount the data directory of your database to add data persistence across container recreations. Volumes are also more portable than bind mounts because the source directory does not need to exist in the host machine.

## Create a database network

If only containerized applications access your database, then there is no need to expose your database to the host network. You can omit exposing the database port and use a Podman network to access the database.

This provides better isolation for the database, because only applications that share the network have access to the database.

## Import Database Data

Setting up the development environment sometimes requires populating the database with sample data for development purposes. Testing environments might require larger data sets than development environments, or even production-like data.

Depending on the database container that you choose, there might be different approaches to load the database with data.

## Database Containers with Data-loading Features

Database container images usually configure a directory where you can place scripts to initialize the database. You can mount your database scripts into that directory. The database container executes the scripts at container creation or at container start.

You can also migrate the data from a running database. Some database containers include an export feature to extract the data from the container while the database is running.

## Load Data with a Database Client

You can also load the data by using a database client that is compatible with your database server. Provide the client with a file containing the data to load and the configuration to connect to the database server.

The container that runs the database might already include that client. In that case, you must provide the database scripts to the container. You can copy the scripts into the container by using the podman cp command.

```
[user@host ~] podman cp SQL_FILE TARGET_DB_CONTAINER:CONTAINER_PATH
```

The preceding command copies *SQL\_FILE*, the file containing the data, from the host to the container.

## Chapter 5 | Persisting Data

After you have copied the scripts into the container, run the database client command to load the data. For example, for a PostgreSQL database the following command executes the *SQL\_FILE* to create and populate the database.

```
[user@host ~] podman exec -it DATABASE_CONTAINER \
 psql -U DATABASE_USER -d DATABASE_NAME \
 -f CONTAINER_PATH/SQL_FILE
```

If a database image does not include this feature, then you can create a container that includes a database client, and use this container to load the data.

For example, the following command creates an ephemeral container to load data into a PostgreSQL database:

```
[user@host ~] podman run -it --rm \
 -e PGPASSWORD=DATABASE_PASSWORD \
 -v ./SQL_FILE:/tmp/SQL_FILE:Z \
 --network DATABASE_NETWORK \
 registry.redhat.io/rhel8/postgresql-12:1-113 \
 psql -U DATABASE_USER -h DATABASE_CONTAINER \
 -d DATABASE_NAME -f /tmp/SQL_FILE
```

This command uses the SELinux option Z to set the SELinux context for the container to have access to the host *SQL\_FILE*.

The command also uses the *DATABASE\_NETWORK* network, which allows the `psql` client in this container to use the *DATABASE\_CONTAINER* name as the hostname for the database container. For the DNS to work, the *DATABASE\_NETWORK* network must have DNS enabled.

## Export Database Data

To export the database data, you can use database backup commands present in the database container image. For example, MySQL provides the `mysqldump` command and PostgreSQL provides the `pg_dump` command.

You can run the following command in a PostgreSQL container to export the database called *DATABASE* to a *BACKUP\_DUMP* file.

```
[student@workstation ~]$ podman exec POSTGRES_CONTAINER \
 pg_dump -Fc DATABASE -f BACKUP_DUMP
no output expected
```

Then, you can copy the dump file out of the database container to import the data in a new container.

## Red Hat Database Containers

The registries in the Red Hat Ecosystem Catalog contain a list of database container images supported by Red Hat.

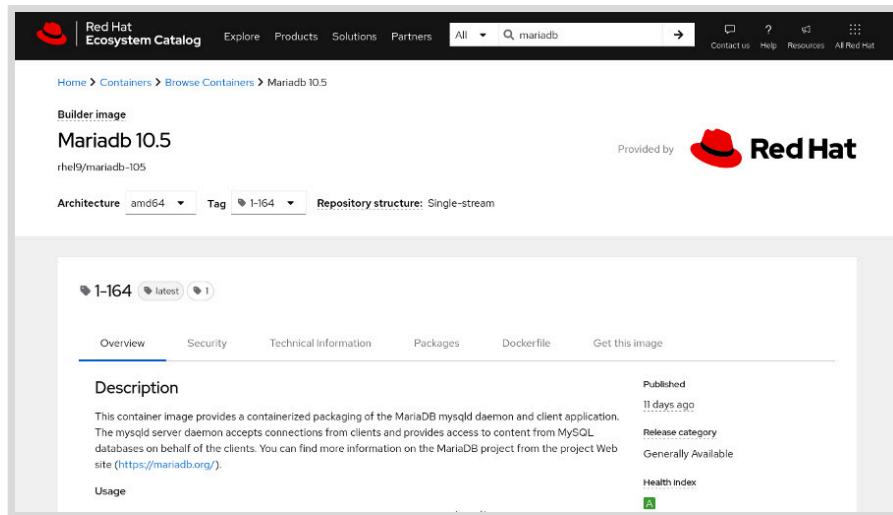
The screenshot shows the Red Hat Ecosystem Catalog interface. At the top, there's a navigation bar with links for Home, Explore, Products, Solutions, Partners, and a search bar labeled "Search Ecosystem Catalog". Below the navigation is a breadcrumb trail: Home > Containers. The main title is "Container images", with a subtitle: "Discover certified container images from Red Hat and third-party providers that enable and extend your Red Hat environments." To the right of the title is a stylized graphic of overlapping triangles. Below the title, there's a section titled "Popular categories" featuring a grid of icons and category names: API Management, Accounting, Application Delivery, Application Server, Automation, Backup & Recovery, Business Intelligence, Business Process Management, Capacity Management, Cloud Management, Collaboration / Groupware / Messaging, and Configuration Management. At the bottom of this section are links to "Browse all categories" and "Have feedback?".

Find the database products containerized by Red Hat and third-party vendors by selecting the **Database & Data Management** category.

The screenshot shows the Red Hat Ecosystem Catalog interface, specifically the "Certified container images" page for the "Database & Data Management" category. The top navigation bar includes a "Containers" dropdown, a search bar, and other standard links. The breadcrumb trail shows "Home > Containers > Browse Containers". The main title is "Certified container images", with a subtitle: "Container images offer lightweight and self-contained software to enable deployment at scale." On the left, there are filters for "Provider" (Red Hat) and "Category" (Database & Data Management). A sidebar lists other categories like Developer Tools, Enterprise Resource Planning, Identity Management, Integration, Logging & Metrics, and Management. The main content area shows a grid of four container images, each with a Red Hat logo: "PostgreSQL APB" (by Red Hat), "MySQL APB" (by Red Hat), "MariaDB APB" (by Red Hat), and "Red Hat Data Grid 7.3 for OpenShift" (by Red Hat). Each card provides a brief description and an "Ansible Playbook" link.

After picking the database container, pull the container image and follow the usage instructions.

For example, if you select the `rhel9/mariadb-105` image, then you can pull the image by navigating to the `Get this image` tab and following the instructions under the `Using podman` login section. The usage instructions are available under the `Overview` tab.



## References

### **Red Hat Ecosystem Catalog**

<https://catalog.redhat.com/software/containers/explore>

### **Red Hat Container Registry Authentication**

<https://access.redhat.com/RegistryAuthentication>

### **Source for the PostgreSQL Container**

<https://github.com/sclorg/postgresql-container>

### **Source for the Mariadb Container**

<https://github.com/sclorg/mariadb-container>

### **Source for the Mongoddb Container**

<https://github.com/sclorg/mongodb-container>

### **Source for the Redis Container**

<https://github.com/sclorg/redis-container>

## ► Guided Exercise

# Working with Databases

Set up a containerized PostgreSQL database for development and testing environments.

## Outcomes

You should be able to:

- Create a containerized database that contains ephemeral data.
- Load the database schema and data on container creation.
- Use the PostgreSQL client in the PostgreSQL container to interact with the database.
- Create a containerized persistent database for a testing environment.
- Migrate the testing environment data to a container running a newer PostgreSQL version.

## Before You Begin

As the `student` user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command copies the database initialization scripts of a Raspberry Pi store to the `lab` directory in your system.

```
[student@workstation ~]$ lab start persisting-databases
```

## Instructions

- 1. Create a PostgreSQL 12 container and call it `persisting-pg12`. Use the `load_db.sh` script in the `lab` directory to load the data.
- 1.1. Examine the `~/D0188/labs/persisting-databases/load_db.sh` script, which loads the database schema definitions and the data into the `model` and `inventory` tables.
  - 1.2. Create a PostgreSQL 12 container that runs interactively and call it `persisting-pg12`.  
Use the `registry.ocp4.example.com:8443/rhel8/postgresql-12:1-113` image.  
Mount the `~/D0188/labs/persisting-databases` host directory into the `/opt/app-root/src/postgresql-start` container directory as a bind mount, with the `Z` option for SELinux.  
Provide the following environment variables:
    - `POSTGRESQL_USER=backend`
    - `POSTGRESQL_PASSWORD=secret_pass`
    - `POSTGRESQL_DATABASE=rpi-store`

```
[student@workstation ~]$ podman run -it --rm \
--name persisting-pg12 \
-e POSTGRESQL_USER=backend \
-e POSTGRESQL_PASSWORD=secret_pass \
-e POSTGRESQL_DATABASE=rpi-store \
-v ~/D0188/labs/persisting-databases:/opt/app-root/src/postgresql-start:z \
registry.ocp4.example.com:8443/rhel8/postgresql-12:1-113
...output omitted...
server started
/var/run/postgresql:5432 - accepting connections
=> sourcing /opt/app-root/src/postgresql-start/load_db.sh ...
...output omitted...
```

On start, containers based on the `rhel8/postgresql-12` image run any scripts that end in `.sh`, found in the `/opt/app-root/src/postgresql-start` directory.

- 1.3. Open a new terminal and run the `psql` PostgreSQL client in the container to query the `model` table.

```
[student@workstation ~]$ podman exec -it persisting-pg12 \
 psql -d rpi-store -c "select * from model"

id| name | model | soc | memory_mb | ethernet |release_date
--+
1 | Raspberry Pi | B | BCM2835 | 256 | t | 2012
2 | Raspberry Pi Zero | Zero | BCM2835 | 512 | f | 2015
3 | Raspberry Pi Zero | 2W | BCM2710A1 | 512 | f | 2021
4 | Raspberry Pi 4 | B | BCM2711 | 4096 | t | 2019
6 | Raspberry Pi 4 | 400 | BCM2711 | 4096 | t | 2020
(5 rows)
```

- ▶ 2. Recreate the previous container to verify that the data loaded into it is not persistent.
    - 2.1. Exit the **persisting-pg12** container by pressing **Ctrl+C**. The container is removed automatically because of the **--rm** option provided in the container creation.
    - 2.2. Recreate the **persisting-pg12** container without loading the database scripts.

```
[student@workstation ~]$ podman run -it --rm \
--name persisting-pg12 \
-e POSTGRESQL_USER=backend \
-e POSTGRESQL_PASSWORD=secret_pass \
-e POSTGRESQL_DATABASE=rpi-store \
registry.ocp4.example.com:8443/rhel8/postgresql-12:1-113
...output omitted...
```

- 2.3. In the previous terminal window, verify that the data you loaded in the preceding step is not present by querying the `model` table.

```
[student@workstation ~]$ podman exec -it persisting-pg12 \
 psql -d rpi-store -c "select * from model"
ERROR: relation "model" does not exist
LINE 1: select * from model
```

- 3. Create a containerized PostgreSQL database for a testing environment. Use a volume to avoid losing data if the container is recreated.
- 3.1. Exit the `persisting-pg12` container by pressing `Ctrl+C`. The container is removed automatically because of the `--rm` option.
  - 3.2. Create a volume called `rpi-store-data` to store the database files.

```
[student@workstation ~]$ podman volume create rpi-store-data
rpi-store-data
```

- 3.3. Create the `persisting-pg12` container in detached mode and map the `rpi-store-data` volume to the `/var/lib/pgsql/data` directory in the container. Bind mount the `postgresql-start` directory like you did previously.

```
[student@workstation ~]$ podman run -d \
 --name persisting-pg12 \
 -e POSTGRESQL_USER=backend \
 -e POSTGRESQL_PASSWORD=secret_pass \
 -e POSTGRESQL_DATABASE=rpi-store \
 -v rpi-store-data:/var/lib/pgsql/data \
 -v ~/D0188/labs/persisting-databases:/opt/app-root/src/postgresql-start:z \
 registry.ocp4.example.com:8443/rhel8/postgresql-12:1-113
c99b...7b7c
```

The `/var/lib/pgsql/data` is the directory where the `rhel8/postgresql-12` image is configured to store the database files.

- 3.4. Verify that the `rpi-store` database contains data by querying the `model` table.

```
[student@workstation ~]$ podman exec -it persisting-pg12 \
 psql -d rpi-store -c "select * from model"
...output omitted...
```

- 4. Recreate the container to test that the data inserted in the `rpi-store` database persists after container recreation.
- 4.1. Remove the container called `persisting-pg12`. Provide the `-f` option to stop and remove the container with the same command.
- ```
[student@workstation ~]$ podman rm -f persisting-pg12
c99b...7b7c
```
- 4.2. Recreate the `persisting-pg12` container and attach the `rpi-store-data` volume without binding the data loading scripts.

```
[student@workstation ~]$ podman run -d \
--name persisting-pg12 \
-e POSTGRESQL_USER=backend \
-e POSTGRESQL_PASSWORD=secret_pass \
-e POSTGRESQL_DATABASE=rpi-store \
-v rpi-store-data:/var/lib/pgsql/data \
registry.ocp4.example.com:8443/rhel8/postgresql-12:1-113
e37a...d0cb
```

4.3. Query the `model` table to verify that the data is preserved.

```
[student@workstation ~]$ podman exec -it persisting-pg12 \
psql -d rpi-store -c "select * from model"
...output omitted...
(5 rows)
```

► 5. Start a containerized pgAdmin interface to manage the database by using a web UI.

5.1. Create a network to allow DNS name resolution between the pgAdmin and database containers.

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

5.2. Delete the existing `persisting-pg12` container.

```
[student@workstation ~]$ podman rm -f persisting-pg12
e37a...d0cb
```

5.3. Recreate the `persisting-pg12` container. The container must use the network called `persisting-network`.

```
[student@workstation ~]$ podman run -d \
--name persisting-pg12 \
-e POSTGRESQL_USER=backend \
-e POSTGRESQL_PASSWORD=secret_pass \
-e POSTGRESQL_DATABASE=rpi-store \
-v rpi-store-data:/var/lib/pgsql/data \
--network persisting-network \
registry.ocp4.example.com:8443/rhel8/postgresql-12:1-113
ab8f...ce8c
```

5.4. Create the pgAdmin container and call it `persisting-pgadmin`. Attach the container to the `persisting-network` to resolve the database container by using the `persisting-pg12` name.

Use the `registry.ocp4.example.com:8443/crunchydata/crunchy-pgadmin4:ubi8-4.30-1` image.

Map the `5050` container port to same host port.

Use the following environment variables for the container:

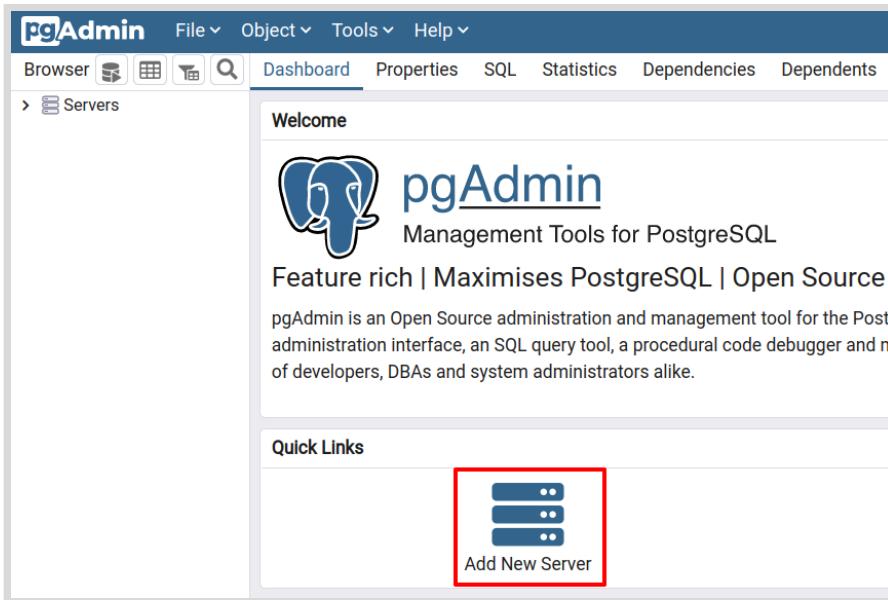
- PGADMIN_SETUP_EMAIL=gls@example.com
- PGADMIN_SETUP_PASSWORD=pga_secret_pass

```
[student@workstation ~]$ podman run -d \
--name persisting-pgadmin \
-e PGADMIN_SETUP_EMAIL=gls@example.com \
-e PGADMIN_SETUP_PASSWORD=pga_secret_pass \
-p 5050:5050 \
--network persisting-network \
registry.ocp4.example.com:8443/crunchydata/crunchy-pgadmin4:ubi8-4.30-1
...output omitted...
cd9g...aa8f
```

**Note**

After executing the previous command, you might see SELinux warnings. Those warnings are safe to ignore.

- 5.5. Log in to pgAdmin by using a web browser and navigating to `http://localhost:5050`. Log in with the credentials that you used to start the container:
 - Email Address/Username: `gls@example.com`
 - Password: `pga_secret_pass`
- 5.6. Connect to the `persisting-pg12` database container by clicking **Add New Server**.



- 5.7. In the General tab, set `rpi-store` as the name.
- 5.8. Switch to the Connection tab. Fill the form with the following data and leave the rest of the fields with their default values.

Field	Value
Hostname/address	<code>persisting-pg12</code>
Username	<code>backend</code>
Password	<code>secret_pass</code>

Click **Save**. The application verifies the connection before exiting the form.



Note

Because the database is listening on the host machine, you might assume that the `localhost` value works for the `Host` name field. This does not work because pgAdmin runs in a container and `localhost` refers to the pgAdmin container itself.

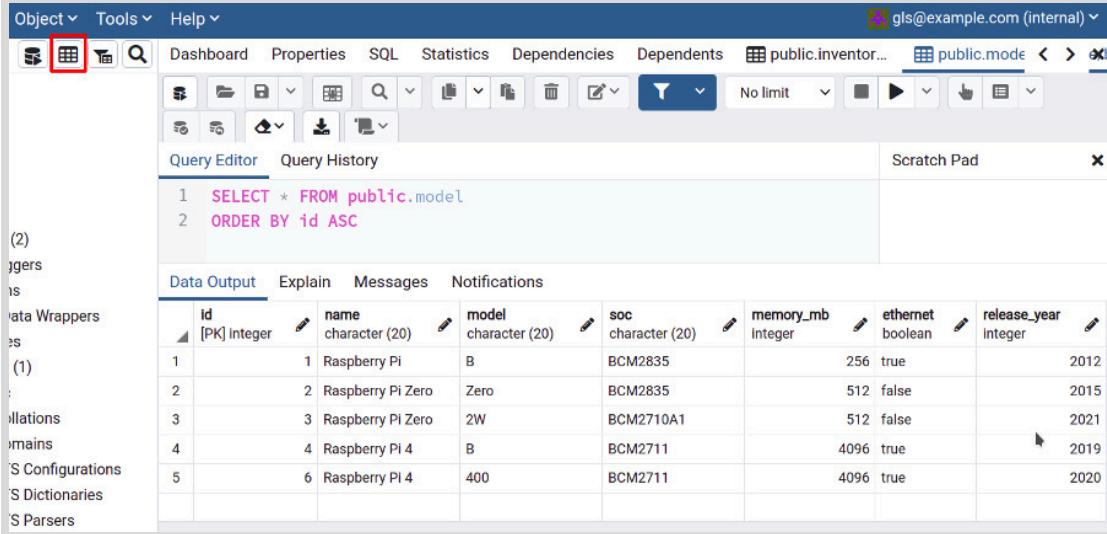
- 5.9. View the data in the `model` table to verify the pgAdmin access to the `rpi-store` database in the `persisting-pg12` container.

Select the `model` table by clicking **Servers** > `rpi-store` > **Databases** > `rpi-store` > **Schemas** > `public` > **Tables** > `model`.

The screenshot shows the pgAdmin interface with the following tree structure:

- Servers (1) > rpi-store
 - Databases (2)
 - postgres
 - rpi-store
 - Casts
 - Catalogs (2)
 - Event Triggers
 - Extensions
 - Foreign Data Wrappers
 - Languages
 - Schemas (1)
 - public
 - Collations
 - Domains
 - FTS Configurations
 - FTS Dictionaries
 - FTS Parsers
 - FTS Templates
 - Foreign Tables
 - Functions
 - Materialized Views
 - Procedures
 - Sequences
- Tables (2)
 - inventory
 - model

Click the **View Data** icon to verify that pgAdmin queries the data in the **persisting-pg12** container.



The screenshot shows the pgAdmin interface with the 'Query Editor' tab selected. The query is:

```
1 SELECT * FROM public.model
2 ORDER BY id ASC
```

The results are displayed in a table:

	id [PK] integer	name character (20)	model character (20)	soc character (20)	memory_mb integer	ethernet boolean	release_year integer
1	1	Raspberry Pi	B	BCM2835	256	true	2012
2	2	Raspberry Pi Zero	Zero	BCM2835	512	false	2015
3	3	Raspberry Pi Zero	2W	BCM2710A1	512	false	2021
4	4	Raspberry Pi 4	B	BCM2711	4096	true	2019
5	6	Raspberry Pi 4	400	BCM2711	4096	true	2020

- 6. Migrate the testing environment data to a container running PostgreSQL 13, a newer version of the PostgreSQL database. Use a new data volume to avoid incompatibilities between the PostgreSQL versions.

- 6.1. Run the `pg_dump` command in the running database container to back up the `rpi-store`. A file called `/tmp/db_dump` stores the backup in the `persisting-pg12` container. Use the `-Fc` option to use the custom format which compresses the backup file.

```
[student@workstation ~]$ podman exec persisting-pg12 \
  pg_dump -Fc rpi-store -f /tmp/db_dump
no output expected
```

- 6.2. Use the `podman cp` command to copy the dump file to the host machine.

```
[student@workstation ~]$ podman cp persisting-pg12:/tmp/db_dump /tmp/db_dump
no output expected
```

- 6.3. Stop the `persisting-pg12` container.

```
[student@workstation ~]$ podman stop persisting-pg12
persisting-pg12
```



Note

You might see a `StopSignal` warning when you stop the container. You can safely ignore this warning.

- 6.4. Create a volume called `rpi-store-data-pg13` to store the database files.

Chapter 5 | Persisting Data

```
[student@workstation ~]$ podman volume create rpi-store-data-pg13  
rpi-store-data-pg13
```

- 6.5. Create a PostgreSQL 13 container that uses the `rpi-store-data-pg13` volume.

```
[student@workstation ~]$ podman run -d \  
--name persisting-pg13 \  
-e POSTGRES_USER=backend \  
-e POSTGRES_PASSWORD=secret_pass \  
-e POSTGRES_DATABASE=rpi-store \  
-v rpi-store-data-pg13:/var/lib/pgsql/data \  
registry.ocp4.example.com:8443/rhel9/postgresql-13:1  
00e9...36a8
```

- 6.6. Copy the dump file to the new container by using the `podman cp` command.

```
[student@workstation ~]$ podman cp /tmp/db_dump persisting-pg13:/tmp/db_dump  
no output expected
```

- 6.7. Run `pg_restore` to restore the `rpi-store` database in the `persisting-pg13` container. Set `rpi-store` as the destination database by using the `-d` option.

```
[student@workstation ~]$ podman exec persisting-pg13 \  
pg_restore -d rpi-store /tmp/db_dump  
no output expected
```

- 6.8. Query the `model` table in the `persisting-pg13` container to validate that the migration worked.

```
[student@workstation ~]$ podman exec -it persisting-pg13 \  
psql -d rpi-store -c "select * from model"  
...output omitted...  
(5 rows)
```

- 6.9. Remove the container with the previous PostgreSQL version and its associated volume to eliminate unused resources.

```
[student@workstation ~]$ podman rm persisting-pg12  
ab8f...ce8c
```

Because named volumes persist after deleting the container, you must delete the `rpi-store-data` volume manually.

```
[student@workstation ~]$ podman volume rm rpi-store-data  
rpi-store-data
```

Finish

On the `workstation` machine, use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish persisting-databases
```

▶ Lab

Persisting Data

Use volumes to provide persistence to an application.

Outcomes

You should be able to create volumes, import data into volumes, and use volumes in an application.

This lab uses a URL shortener application, which consists of three components: a database container, a back-end container, and a front-end container.

The source code for the front end and back end is available at `~/D0188/solutions/persisting-lab` after you execute the lab script.

Note the following:

- The back-end container uses the following information:
 - The back end uses default values for the user, password, and database for simplicity.
 - The back end uses the database container name to resolve the database IP address. Do not change the database container name.
- The front-end container uses the following information:
 - The front end uses an Nginx server to redirect requests from `localhost:8080` to the `persisting-backend:8080` host. Do not change the back-end container name.
 - If the front end exits after start, execute `podman logs persisting-frontend` to check the logs.
- The application can become unresponsive after you stop individual containers. If this happens, stop all containers and start the containers in order of database, back end, and front end.

Before You Begin

As the student user on the workstation machine, use the `lab` command to prepare your system for this exercise.

```
[student@workstation ~]$ lab start persisting-lab
```

The lab script continuously evaluates the objectives of this lab. Keep the script running in a terminal window and complete the objectives of this lab from a new terminal window.

Instructions

1. Create a named volume with the following parameters:
 - The volume is called `postgres-vol`.
 - The volume contains the contents of the `~/D0188/labs/persisting-lab/postgres-vol.tar.gz` file.
2. Start the application database container with the following parameters:
 - Call the container `persisting-db`.

- Start the container in the background.
 - Connect the container to the **persisting-net** network.
 - Use the following environment variables:
 - POSTGRESQL_PASSWORD=pass
 - POSTGRESQL_USER=user
 - POSTGRESQL_DATABASE=db
 - Mount the **postgres-vol** volume to the `/var/lib/pgsql/data` directory.
 - Use the `registry.ocp4.example.com:8443/rhel9/postgresql-13:1` image.
- 3.** Start the back end with the following parameters:
- Call the container **persisting-backend**.
 - Start the container in the background.
 - Use the environment variable `DB_HOST=persisting-db`.
 - Expose the port **8080** on the machine to route requests to port **8080** inside the container.
 - Connect the container to the **persisting-net** network.
 - Use the `registry.ocp4.example.com:8443/redhattraining/podman-urlshortener-backend` image.
- 4.** Start the front end with the following parameters:
- Call the container **persisting-frontend**.
 - Start the container in the background.
 - Connect the container to the **persisting-net** network.
 - Expose the port **3000** on the machine to route requests to port **8080** inside the container.
 - Use the `registry.ocp4.example.com:8443/redhattraining/podman-urlshortener-frontend` image.
- 5.** Test the application.
- 5.1. In a web browser, verify the functionality of the application at `http://localhost:3000`.
 - 5.2. In a web browser, test the database data import by navigating to `http://localhost:8080/api/shorturl/a9yi4rcl5uuzunv`.
The `a9yi4rcl5uuzunv` short URL is a part of the database data that you imported in a previous step into the **postgres-vol** volume.

Finish

As the **student** user on the **workstation** machine, use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

Press **y** when the `lab start` command prompts you to execute the `finish` function. Alternatively, execute the following command:

```
[student@workstation ~]$ lab finish persisting-lab
```

► Solution

Persisting Data

Use volumes to provide persistence to an application.

Outcomes

You should be able to create volumes, import data into volumes, and use volumes in an application.

This lab uses a URL shortener application, which consists of three components: a database container, a back-end container, and a front-end container.

The source code for the front end and back end is available at `~/D0188/solutions/persisting-lab` after you execute the lab script.

Note the following:

- The back-end container uses the following information:
 - The back end uses default values for the user, password, and database for simplicity.
 - The back end uses the database container name to resolve the database IP address. Do not change the database container name.
- The front-end container uses the following information:
 - The front end uses an Nginx server to redirect requests from `localhost:8080` to the `persisting-backend:8080` host. Do not change the back-end container name.
 - If the front end exits after start, execute `podman logs persisting-frontend` to check the logs.
- The application can become unresponsive after you stop individual containers. If this happens, stop all containers and start the containers in order of database, back end, and front end.

Before You Begin

As the `student` user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

```
[student@workstation ~]$ lab start persisting-lab
```

The lab script continuously evaluates the objectives of this lab. Keep the script running in a terminal window and complete the objectives of this lab from a new terminal window.

Instructions

1. Create a named volume with the following parameters:
 - The volume is called `postgres-vol`.
 - The volume contains the contents of the `~/D0188/labs/persisting-lab/postgres-vol.tar.gz` file.
- 1.1. Create the volume.

```
[student@workstation ~]$ podman volume create postgres-vol
postgres-vol
```

- 1.2. Import the ~/D0188/labs/persisting-lab/postgres-vol.tar.gz file to the volume:

```
[student@workstation ~]$ podman volume import postgres-vol \
~/D0188/labs/persisting-lab/postgres-vol.tar.gz
...no output expected...
```

2. Start the application database container with the following parameters:

- Call the container persisting-db.
- Start the container in the background.
- Connect the container to the persisting-net network.
- Use the following environment variables:
 - POSTGRESQL_PASSWORD=pass
 - POSTGRESQL_USER=user
 - POSTGRESQL_DATABASE=db
- Mount the postgres-vol volume to the /var/lib/pgsql/data directory.
- Use the registry.ocp4.example.com:8443/rhel9/postgresql-13:1 image.

- 2.1. Create the persisting-net network.

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

- 2.2. Start the database container.

```
[student@workstation ~]$ podman run --name persisting-db -d \
--net persisting-net -e POSTGRESQL_USER=user -e POSTGRESQL_PASSWORD=pass \
-e POSTGRESQL_DATABASE=db \
--mount='type=volume,src=postgres-vol,dst=/var/lib/pgsql/data' \
registry.ocp4.example.com:8443/rhel9/postgresql-13:1
c97f...4a29
```

3. Start the back end with the following parameters:

- Call the container persisting-backend.
- Start the container in the background.
- Use the environment variable DB_HOST=persisting-db.
- Expose the port 8080 on the machine to route requests to port 8080 inside the container.
- Connect the container to the persisting-net network.
- Use the registry.ocp4.example.com:8443/redhattraining/podman-
urlshortener-backend image.

```
[student@workstation ~]$ podman run --name persisting-backend -d \
-e DB_HOST=persisting-db -p 8080:8080 --net persisting-net \
registry.ocp4.example.com:8443/redhattraining/podman-urlshortener-backend
3a46...4e60
```

4. Start the front end with the following parameters:

- Call the container `persisting-frontend`.
- Start the container in the background.
- Connect the container to the `persisting-net` network.
- Expose the port `3000` on the machine to route requests to port `8080` inside the container.
- Use the `registry.ocp4.example.com:8443/redhattraining/podman-urlshortener-frontend` image.

```
[student@workstation ~]$ podman run --name persisting-frontend -d \
--net persisting-net -p 3000:8080 \
registry.ocp4.example.com:8443/redhattraining/podman-urlshortener-frontend
b10e...940f
```

5. Test the application.

- 5.1. In a web browser, verify the functionality of the application at `http://localhost:3000`.
- 5.2. In a web browser, test the database data import by navigating to `http://localhost:8080/api/shorturl/a9yi4rc15uuzunv`.

The `a9yi4rc15uuzunv` short URL is a part of the database data that you imported in a previous step into the `postgres-vol` volume.

Finish

As the `student` user on the `workstation` machine, use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

Press `y` when the `lab start` command prompts you to execute the `finish` function. Alternatively, execute the following command:

```
[student@workstation ~]$ lab finish persisting-lab
```

Summary

- Containers use a copy-on-write (COW) file system.
- Containers use a thin runtime layer, which is ephemeral by default.
- You can mount volumes and bind mounts to persist container data.
- Some workloads, such as databases, might experience performance problems with the COW file system.
- You can use Podman volumes to avoid using the COW file system.

Chapter 6

Troubleshooting Containers

Goal

Analyze container logs and configure a remote debugger.

Objectives

- Read container logs and troubleshoot common container problems.
- Configure a remote debugger during application development.

Sections

- Container Logging and Troubleshooting (and Guided Exercise)
- Remote Debugging Containers (and Guided Exercise)

Lab

- Troubleshooting Containers

Container Logging and Troubleshooting

Objectives

- Read container logs and troubleshoot common container problems.

Troubleshoot Container Startup

A container might not be able to start successfully for different reasons, for example, due to missing configuration or a file access issue.

Depending on the containerized application, the process executed by the container can either exit with an error status or keep running in an inconsistent state. You can list running and stopped containers by using the `podman ps -a` command.

```
[user@host ~]$ podman ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
b72652504844 ... 28 sec... Up... ok_service
867f6f559629 ... /bin... 21 sec... Exited... 0... failing_service
```

If the container is in the `Exited` status, then the problem might be in the start-up process. Many applications output error information when they encounter an issue during startup. To access this information, you can use the `podman logs` command.

```
[user@host ~]$ podman logs CONTAINER
...output omitted...
```

If you want to follow the logs in real time, then you can add the `-f` option to the preceding command.

Troubleshoot Container Networking

Common container networking issues include the following situations:

- Incorrect port mapping.
- No network access between containers.
- Hostname resolution problems (DNS).

Port Mapping Issues

Developers must distinguish between the following port configuration:

- Ports that the application inside the container listens on.
- Podman port mapping configuration, which maps ports on the host to the application ports inside the container.

If the port that you assign to the container in the port mapping configuration does not match the application port, then the containerized application fails to communicate with the host.

You can use the `podman port CONTAINER` command to list the current container port mapping.

```
[user@host ~]$ podman port CONTAINER
8000/tcp -> 0.0.0.0:8080
```

In the preceding example, a container exposes the 8000 container port to the host machine on the 8080 host port at all the host network interfaces.

Applications might listen on a different port than you expect. Because Podman commands do not interact with the application, you cannot use Podman commands to verify the port that the application uses.

To verify the application ports in use, list the open network ports in the running container. Use Linux commands such as the socket statistics (`ss`) command to list open ports. A socket is the combination of a port and an IP address. The `ss` command lists the open sockets in a system. You can provide the `ss` command with options to filter and produce the desired output:

- `-p`: display the process using the socket
- `-a`: display listening and established connections
- `-n`: display numeric ports instead of mapped service names
- `-t`: display TCP sockets

```
[user@host ~]$ podman exec -it CONTAINER ss -pant
Netid State ... Local Address:Port Peer Address:Port Process
tcp LISTEN ... 0.0.0.0:9091 0.0.0.0:* users:(python",pid=...)
```

Use the output of the `ss` command to verify that your port mapping matches the application ports in use. You can use values such as the local address and port, the peer address and port, and the process. In the preceding example, a Python service is listening on port 9091. Therefore, to expose this service to the host machine, make sure that a port mapping exists for this port.

To reduce the container attack surface, containers usually lack many commands. You can run the host system commands within the container *network namespace* by using the `nsenter` command. A container namespace is how the Linux kernel isolates a system resource view from the container perspective. For example, a container network namespace shows the system's networking stack to the container as if the container was the only process using the stack. There are other types of namespaces, but this material is out of scope for the course.

You can target a container namespace by using the `nsenter -n -t` command with the container process ID (PID).

To get the container PID you can use the following `podman inspect` command:

```
[user@host ~]$ podman inspect CONTAINER --format '{{.State.Pid}}'
CONTAINER_PID
```

After getting the container PID, you can run the `nsenter` command. Run the command with elevated privileges by using `sudo`, as follows:

```
[user@host ~]$ sudo nsenter -n -t CONTAINER_PID ss -pant
Netid State ... Local Address:Port Peer Address:Port Process
tcp LISTEN ... 0.0.0.0:9091 0.0.0.0:* ...
...output omitted...
```

**Note**

Containerized applications should listen on the 0.0.0.0 address, which refers to any network interface within the container. Using the 127.0.0.1 loopback interface isolates the application from communicating outside of the container.

Container Network Connectivity Issues

Developers commonly use Podman networks to isolate container network traffic from the host. If a container does not attach to a Podman network, then it cannot communicate with other containers on that network.

You can use the `podman inspect` command to verify that every container is using a specific network.

```
[user@host ~]$ podman inspect CONTAINER --format='{{.NetworkSettings.Networks}}'
map[network_name:0xc000a825a0]
```

When containers communicate by using Podman networks, there is no port mapping involved.

Name Resolution Issues

Podman networks provide connectivity for container-to-container network traffic by using IP addresses. However, because IP addresses might change, developers use hostnames to address these containers in a predictable way. The Domain Name System (DNS) service translates container names to IP addresses for network communication. If the network in use does not have DNS enabled, then containers cannot resolve the container hostnames and cannot communicate.

To ensure that DNS is enabled for a Podman network use the `podman network inspect` command.

```
[user@host ~]$ podman network inspect NETWORK
...output omitted...
  "dns_enabled": true,
...output omitted...
```

Podman Events

When troubleshooting container issues, the initial step is to gather information. You typically start by examining the container logs. However, this approach assumes that the container was operational for sufficient time to produce logs. If the container had never started, there would have been no logs. In such a situation, you can use events to obtain more data. Events provide supplementary information in addition to the container logs.

Podman includes an events system that records activities. Podman monitors objects such as containers, images, pods, and volumes. Podman also tracks event types such as create, start, stop, pull, or remove. For instance, with events, you can follow when you create a container or pull an image. Monitor and view events in Podman by using the `podman events` command.

Podman requires a backend logging mechanism for recording events. By default, the logging mechanism used is `journald`. The `events_logger` field in the `containers.conf` file controls this behavior. The available logging methods are `file`, `journald`, and `none`.

Use the `podman info` command to view the current value.

```
[user@host ~]$ podman info --format {{.Host.EventLogger}}
journald
```

To monitor events in Podman, use the `podman events` command. By default, the `podman events` command follows new events as they occur continuously. You can use the `--stream=false` option to force the command to exit after reading the last known event. The following example prints the image pull and container create events.

```
[user@host ~]$ podman events --stream=false
2024-01-16 13:51:46.074958359 -0500 EST system refresh
2024-01-16 ... -0500 EST image pull 699...47d registry.access.redhat.com/rhscl/
httpd-24-rhel7 ①
2024-01-16 ... -0500 EST container create fb6...ce4
  (image=registry.access.redhat.com/rhscl/httpd-24-rhel7:latest ②
...output omitted...
```

① The image pull event.

② The container create event.

The `--filter` or `-f` option enables you to filter events by object and event type multiple times.

```
[user@host ~]$ podman events --filter event=create --filter type=container
--stream=false
2024-01-16 ... -0500 EST container create fb6...ce4
  (image=registry.access.redhat.com/rhscl/httpd-24-rhel7:latest
...output omitted...
```

You can use the `--since` and `--until` options to view past events. These options enable you to specify a time range for the events you are interested in. The `--since` and `--until` option values can be RFC3339Nano time stamps or a Go duration string such as `10m` or `5h`.

```
[user@host ~]$ podman events --since 5m --stream=false
2024-01-28 ... -0500 EST image pull bc1...011 registry.access.redhat.com/ubi8/ubi
2024-01-28 ... -0500 EST container remove f9f...63e
  (image=registry.access.redhat.com/ubi8/ubi:latest
```



References

Namespaces and Nsenter

<https://www.redhat.com/sysadmin/container-namespaces-nsenter>

`podman-events(1)` man page

► Guided Exercise

Container Logging and Troubleshooting

Learn how to troubleshoot and solve some common problems when running containers.

Outcomes

You should be able to:

- Search container logs.
- Verify a container definition.
- Use commands inside a container for troubleshooting.
- Use host commands that are not present in the container for container troubleshooting.
- Troubleshoot networking issues.
- Troubleshoot file permission issues.

Before You Begin

As the student user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

The `lab start` command performs the following actions:

- Creates a PostgreSQL database container called `smart-home-db` that is attached to the `troubleshooting-lab` network.
- Creates the `smart-home-api` container, which fails to start.
- Copies the `automations.yaml` configuration file to the project directory.

The smart home API uses a server called `uvicorn` to run a Python back end.

```
[student@workstation ~]$ lab start troubleshooting-logging
```

Instructions

- 1. Read the `smart-home-api` container logs to see the cause of the failure.

```
[student@workstation ~]$ podman logs smart-home-api
...output omitted...
File "/opt/app-root/lib64/python3.9/site-packages/psycopg2/__init__.py", line
122, in connect
    conn = _connect(dsn, connection_factory=connection_factory, **kwasync)
sqlalchemy.exc.OperationalError: (psycopg2.OperationalError) could not translate
host name "smart-home-db" to address: Name or service not known

(Background on this error at: https://sqlalche.me/e/14/e3q8)
```

The container fails with an error that indicates that there is a DNS problem. The application in the `smart-home-api` container cannot resolve the `smart-home-db` database hostname.

- 2. Verify that you can resolve the database container name by using DNS.

- 2.1. Inspect the smart-home-db container to verify the networks that it is using.

```
[student@workstation ~]$ podman inspect \
smart-home-db --format='{{.NetworkSettings.Networks}}'
map[troubleshooting-lab:0xc000f92900]
```

- 2.2. Verify that the troubleshooting-lab network has DNS enabled.

```
[student@workstation ~]$ podman network inspect troubleshooting-lab
[
{
  "name": "troubleshooting-lab",
  ...output omitted...
  "dns_enabled": true,
  ...output omitted...
}
]
```

- 2.3. Inspect the smart-home-api container to verify the networks that it is using.

```
[student@workstation ~]$ podman inspect \
smart-home-api --format='{{.NetworkSettings.Networks}}'
map[]
```

The smart-home-api container is not using any network. The container requires access to the troubleshooting-lab network to reach the smart-home-db container.

- 2.4. Remove the smart-home-api container.

```
[student@workstation ~]$ podman rm smart-home-api
8eee...ef70
```

- 2.5. Run the smart-home-api container and attach it to the troubleshooting-lab network with the following configuration.

- Use the `--rm` and `-d` options.
- Call the container `smart-home-api`.
- Attach the container to the troubleshooting-lab network.
- Use the following environment variables:
 - `DB_HOST=smart-home-db`
 - `DB_USER=backend`
 - `DB_PASSWORD=secret_pass`
- Map the host port 8080 to container port 8080.
- Use the `registry.ocp4.example.com:8443/redhattraining/smart-home-api:1.0` image.

```
[student@workstation ~]$ podman run -d --rm \
--name smart-home-api \
-e DB_HOST=smart-home-db -e DB_USER=backend -e DB_PASSWORD=secret_pass \
-p 8080:8080 \
--network troubleshooting-lab \
registry.ocp4.example.com:8443/redhattraining/smart-home-api:1.0
a361...241a
```

- 2.6. Verify that the `smart-home-api` container starts successfully by reading the container logs.

```
[student@workstation ~]$ podman logs smart-home-api
INFO: Started server process [1]
INFO: Waiting for application startup.
INFO: Application startup complete.
INFO: Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
```

- ▶ 3. Test the API endpoint of the `smart-home-api` container and troubleshoot the connection problems.

- 3.1. Test the API from inside the `smart-home-api` container by querying the `http://localhost:8080/device/1` address. Verify that the request fails due to a connection error on port 8080.

```
[student@workstation ~]$ podman exec \
smart-home-api curl http://localhost:8080/device/1
...output omitted...
curl: (7) Failed to connect to localhost port 8080: Connection refused
```

- 3.2. Look for the server active connections. Use the `socket statistics (ss)` command with the following options.

- `-p`: show the process using the socket
- `-a`: show listening and established connections
- `-n`: display numeric ports instead of mapped service names
- `-t`: display TCP sockets

```
[student@workstation ~]$ podman exec smart-home-api ss -pant
Error: crun: executable file ss not found in $PATH: No such file or directory: OCI
runtime attempted to invoke a command that was not found
```

The `ss` command is not available in the container. Images usually lack many commands. This is a good security practice because it reduces the container attack surface.

- 3.3. Use the `nsenter` host command to run the host `ss` command in the container.

Copy the container process ID (PID) for later use with the `nsenter` command. The PID number might differ for your container.

```
[student@workstation ~]$ podman inspect smart-home-api --format '{{.State.Pid}}'
2809
```

Chapter 6 | Troubleshooting Containers

Run the `nsenter` command. Use the `-t` option to set the container PID obtained previously as the target. Use the `-n` option to enter the `smart-home-pid` network namespace.

```
[student@workstation ~]$ sudo nsenter -t 2809 -n ss -pant
Netid State Recv-Q Send-Q Local Address:Port Peer Address:Port Process
tcp    LISTEN 0        2048      0.0.0.0:8000      0.0.0.0:*   users:
"uvicorn",pid=76641,fd=23
...output omitted...
```

The application server `uvicorn` is listening on the `0.0.0.0` address on port 8000. In the preceding steps, you attempted to test the server on port 8080, which is incorrect.

- 3.4. Confirm that the application responds on port 8000 from within the container. Send a request to the `http://localhost:8000/device/1` endpoint.

```
[student@workstation ~]$ podman exec -it \
smart-home-api curl http://localhost:8000/device/1
{"id":1,"name":"Bedroom Light","state":"on"}%
```

- 3.5. Test whether the application is reachable from the host by sending a request to the `http://localhost:8080/device/1` endpoint.

```
[student@workstation ~]$ curl http://localhost:8080/device/1
curl: (56) Recv failure: Connection reset by peer
```

The request fails because the port mapping is still using the incorrect container port.

- 3.6. Recreate the container with the correct port mapping. Map port 8080 from the host to port 8000 in the container.

Stop the `smart-home-api` container.

```
[student@workstation ~]$ podman stop smart-home-api
smart-home-api
```

Rerun the container. Map the 8000 container port to the 8080 host port.

```
[student@workstation ~]$ podman run -d --rm \
--name smart-home-api \
-e DB_HOST=smart-home-db -e DB_USER=backend -e DB_PASSWORD=secret_pass \
-p 8080:8000 \
--network troubleshooting-lab \
registry.ocp4.example.com:8443/redhattraining/smart-home-api:1.0
4a57...4f1g
```

- 3.7. Confirm that the application responds from the host by sending a request to the `http://localhost:8080/device/1` endpoint.

```
[student@workstation ~]$ curl http://localhost:8080/device/1
{"id":1,"name":"Bedroom Light","state":"on"}%
```

- 4. Customize the automation scripts of the smart-home-api container. To do this, bind mount the automations.yaml configuration file to the /config/automations.yaml container path.

The API provides an /automations endpoint to validate the current container configuration.

- 4.1. Change into the ~/D0188/labs/troubleshooting-logging/smart-home directory.

```
[student@workstation ~]$ cd ~/D0188/labs/troubleshooting-logging/smart-home
no output expected
```

- 4.2. Stop the smart-home-api container.

```
[student@workstation smart-home]$ podman stop smart-home-api
smart-home-api
```

- 4.3. Recreate the container with the ./automations.yaml file mounted to the / config/automations.yaml file within the container.

```
[student@workstation smart-home]$ podman run -d --rm \
--name smart-home-api \
-e DB_HOST=smart-home-db -e DB_USER=backend -e DB_PASSWORD=secret_pass \
-p 8080:8000 \
--network troubleshooting-lab \
-v ./automations.yaml:/config/automations.yaml \
registry.ocp4.example.com:8443/redhattraining/smart-home-api:1.0
8n57...4f1g
```

- 4.4. Confirm that the container has access to the configuration.

```
[student@workstation smart-home]$ curl http://localhost:8080/automations
{"detail": "No automations defined"}%
```

- 4.5. Verify that the /config/automations.yaml file exists in the container.

```
[student@workstation smart-home]$ podman exec \
smart-home-api ls -l /config/
ls: cannot access '/config/automations.yaml': Permission denied
total 0
?????????? ? ? ? ? ? automations.yaml
```

The file exists but the container cannot read the file permissions.

- 4.6. Troubleshoot SELinux permissions.

Run ls with the -Z option to get the SELinux context for the automations.yaml file on the host.

```
[student@workstation smart-home]$ ls -Z automations.yaml
system_u:object_r:user_home_t:s0 automations.yaml
```

Chapter 6 | Troubleshooting Containers

The `user_home_t` SELinux context indicates that Podman must change the SELinux permissions to access the file.

- 4.7. Stop the `smart-home-api` container.

```
[student@workstation smart-home]$ podman stop smart-home-api  
smart-home-api
```

- 4.8. To fix the SELinux permissions, recreate the container and add the `Z` option to the volume mount.

```
[student@workstation smart-home]$ podman run -d --rm \  
--name smart-home-api \  
-e DB_HOST=smart-home-db -e DB_USER=backend -e DB_PASSWORD=secret_pass \  
-p 8080:8000 \  
--network troubleshooting-lab \  
-v ./automations.yaml:/config/automations.yaml:Z \  
registry.ocp4.example.com:8443/redhattraining/smart-home-api:1.0  
1n57...4f1g
```

Verify that the SELinux context changed for the `automations.yaml` file.

```
[student@workstation smart-home]$ ls -Z automations.yaml  
system_u:object_r:container_file_t:s0:c539,c793 automations.yaml
```

The `container_file_t` SELinux context permits Podman to access the host file from the container.

- 4.9. Validate that the `/automations` endpoint returns the custom configuration from the host `automations.yaml` file.

```
[student@workstation smart-home]$ curl http://localhost:8080/automations  
[{"automation": "Turn garage lights ON when presence detected", "wait_for_trigger":  
 [{"platform": "event", "event_type": "PRESENCE_DETECTED"},  
 {"platform": "state", "device_id": 1, "to": "on", "for": 60}]]%
```

- 5. Return to the home directory:

```
[student@workstation smart-home]$ cd  
no output expected
```

Finish

On the `workstation` machine, use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation smart-home]$ lab finish troubleshooting-logging
```

Remote Debugging Containers

Objectives

- Configure a remote debugger during application development.

Remote Debugging Containers

Debugging applications is an important technique to find and fix bugs. Although a developer can use print statements to observe runtime variables and behavior, using a debugger is more flexible. Debuggers quicken the introspection of complex interactions, which are more likely to have bugs in the first place.

Debugging Without Containers

Many programming language runtimes provide a way to attach a debugger to a running program. When attached, a debugger provides the following features.

- Attach breakpoints that stop program execution.
- Step through individual application instructions.
- Inspect and modify variables.
- Evaluate custom expressions.

For web-based applications, a debugger usually connects via a debug port. If both the application and the debugger are on the same machine, then this connection requires no additional configuration, such as port forwarding.

Debugging Within Containers

Because containers are isolated from their host, debugging containerized applications requires additional steps for connecting a debugger.

Containers do not expose any ports by default, so any debug ports must be forwarded. Additionally, the debug server must listen for remote connections.

For example, consider the Node.js debug mode, which you can enable with the `--inspect` option on the `node` command. By default, the Node.js debug server binds to `127.0.0.1` on port `9229`. To connect to a containerized Node.js application, expose the container port `9229` on the host machine. The debug server must also bind to a non-local interface, for example by using the `0.0.0.0` address.



Warning

Debug protocols permit program tampering, which poses a security risk. Do not expose processes in debug mode outside of a controlled environment.

Live Updating Code

When debugging, developers perform experiments via small changes to the code. However, because container images often include the application code, local changes are not reflected in the image until developers recreate it, for example by using the `podman build` command.

Recreating container images on every code change can be time consuming. One solution is to mount the application code from the host by using a bind mount during debugging. By overlaying the code with a bind mount, developers can test code changes without recreating the image.

This solution depends on the ability of the application runtime to perform live code changes. The application runtime must have the ability to change the application code without restarting the application.

Debugging With VSCodium

VSCodium is an open source version of the Visual Studio Code text editor. Both versions of the editor include a built-in debugger that can connect to various runtimes' debug protocols.

Developers define a *launch configuration* to attach the VSCodium debugger to the application. The following launch configuration example attaches a debugger to a Node.js application via port 9229.

```
{  
  "version": "0.2.0",  
  "configurations": [  
    {  
      "type": "node",  
      "request": "attach",  
      "name": "My Config",  
      "port": 9229,  
      "address": "localhost",  
      "localRoot": "${workspaceFolder}",  
      "remoteRoot": "/"  
    }  
  ]  
}
```



References

Node.js Debugging Getting Started Guide

<https://nodejs.org/en/docs/guides/debugging-getting-started/>

Debugging in Visual Studio Code

<https://code.visualstudio.com/Docs/editor/debugging>

► Guided Exercise

Remote Debugging Containers

Use VSCodium to debug a Node.js application running inside a container.

Outcomes

You should be able to:

- Start a container that exposes the Node.js debug port.
- Attach the VSCodium debugger to an application.
- Use the debugger to step through code execution and find a bug.
- Mount the application code into the container.

Before You Begin

As the student user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command ensures that the exercise materials exist in the classroom.

```
[student@workstation ~]$ lab start troubleshooting-debugging
```

Instructions

- 1. Build and run a container image for the application.

- 1.1. Navigate to the application directory.

```
[student@workstation ~]$ cd ~/DO188/labs/troubleshooting-debugging/nodebug
no output expected
```

- 1.2. Examine the NPM scripts in the `package.json` file.

```
[student@workstation nodebug]$ cat package.json
...output omitted...
"scripts": {
  "start": "node index.js",
  "debug": "nodemon --inspect=0.0.0.0 index.js"
},
...output omitted...
```

The `start` script starts the application. The `debug` script starts the application with the `--inspect` option to enable debugging.

The Node.js debug mode binds to `127.0.0.1` by default, which means it is available only inside the container. To connect from outside of the container, it must bind to `0.0.0.0`.

- 1.3. Build a container image for the application.

```
[student@workstation nodebug]$ podman build -t nodebug .
...output omitted...
Successfully tagged localhost/nodebug:latest
```

- 1.4. Create a container that forwards traffic on the 8080 and 9229 ports to the same ports inside of the container. Override the container start command so that it runs the debug npm script. Run the container in the background by using the -d option and remove the container when it stops by using the --rm option.

```
[student@workstation nodebug]$ podman run -d --rm \
--name nodebug -p 8080:8080 -p 9229:9229 \
nodebug npm run debug
...output omitted...
```

- 1.5. Confirm that the application started in debug mode.

```
[student@workstation nodebug]$ podman logs nodebug
...output omitted...
Debugger listening on ws://0.0.0.0:9229/16a2...4bf6c
...output omitted...
app started on port 8080
```

The application runs on port 8080 and the debugger listens on port 9229. Podman forwards both of these ports from the host to the container.

- 2. Attach VSCodium to the running container and make a request to the /echo endpoint.

- 2.1. Launch VSCodium with the application project.

```
[student@workstation nodebug]$ codium .
```

If prompted, click Yes, I trust the authors.

- 2.2. Click Run > Add configuration to open a selection menu, and then click Node.js.

- 2.3. Replace the default contents with the following launch configuration and save the file.

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "node",
      "request": "attach",
      "name": "Debug Nodebug",
      "port": 9229,
      "address": "localhost",
      "localRoot": "${workspaceFolder}",
      "remoteRoot": "/"
    }
  ]
}
```

The preceding configuration is available in the `~/D0188/labs/troubleshooting-debugging/nodebug/launch-config.json` file.

- 2.4. Click **Run > Start Debugging** to launch the configuration and attach the VSCode debugger to the running application.
- 2.5. In the terminal window, make a request to the running application.

```
[student@workstation nodebug]$ curl localhost:8080/echo?message=hello
no output expected
```

The request does not respond as the hard-coded breakpoint was triggered.



Note

In Node.js, the `debugger` statement is a hard-coded breakpoint when running in debug mode. Outside of the debug mode, this statement has no effect.

- 2.6. Notice that the debugger reached the breakpoint in the `index.js` file and the execution context is displayed in the debug panel.

Observe that the local `message` variable uses the `hello` value. The application sets the variable from the `message` query parameter that you supplied to the request.

- 2.7. Continue the execution by clicking the **Continue** icon.

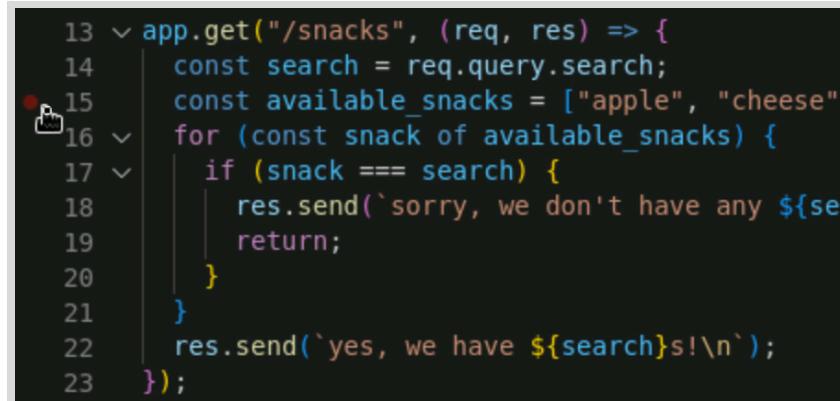
```
.. JS index.js :|> ⌂ ⌂ ⌂ ⌂ ⌂ ⌂ x.js x { } launch
1 const express = require("express");
2
3 const PORT_NUMBER = 8080;
4
5 const app = express();
6
7 app.get("/echo", (req, res) => [
8   const message = req.query.message;
9   debugger;
```

Figure 6.1: Continue program execution

In the terminal window, the `curl` request completes and echoes our message: `hello`.

- 3. Use the debugger to find and fix a bug in the `/snacks` endpoint.

- 3.1. Attach a breakpoint to the endpoint. Click the red circle next to the line in the file that assigns the `available_snacks` variable.



```

13  app.get("/snacks", (req, res) => {
14    const search = req.query.search;
15    const available_snacks = ["apple", "cheese"]
16    for (const snack of available_snacks) {
17      if (snack === search) {
18        res.send(`sorry, we don't have any ${search}`);
19        return;
20      }
21    }
22    res.send(`yes, we have ${search}s!\n`);
23  );

```

Figure 6.2: Attach a breakpoint

**Note**

The red circle does not appear until you hover over the space to the left of the line number.

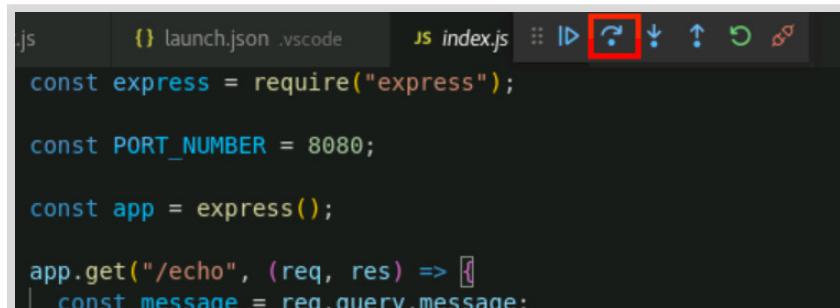
A solid circle persists if you attached the breakpoint successfully.

- 3.2. In the terminal window, make a request to the /snacks endpoint.

```
[student@workstation nodebug]$ curl localhost:8080/snacks?search=apple
```

Execution pauses because the breakpoint in the snacks endpoint is triggered.

- 3.3. Click the **Step Over** icon to step through the function line-by-line.



```

js    {} launch.json .vscode    js index.js :: ID ⏪ ⏴ ⏵ ⏶ ⏷ ⏸ ⏹
const express = require("express");

const PORT_NUMBER = 8080;

const app = express();

app.get("/echo", (req, res) => [
  const message = req.query.message;
]

```

Figure 6.3: Step Over icon

Each time you click the button, the highlighted line of code is executed.

- 3.4. Continue to click the button until the request responds that the search has failed and the function returns.

```
[student@workstation nodebug]$ curl localhost:8080/snacks?search=apple
sorry, we don't have any apples :(
```

This is a bug because the queried snack, apples, is in the available_snacks list.

- 3.5. Try to swap the lines containing `res.send` statements. VSCode does not allow this because the file is read-only.

This editor buffer does not contain the `index.js` file, but instead an in-memory copy of what VSCode is debugging.

- 3.6. Open the `index.js` file by clicking **View > Explorer** and double-clicking the `index.js` file in the explorer panel.

Change the application response when a snack is found as follows:

```
app.get("/snacks", (req, res) => {
  const search == req.query.search;
  const available_snacks == ["apple", "cheese", "cracker", "lunchmeat", "olive"];
  for (const snack of available_snacks) {
    if (snack === search) {
      res.send(`yes, we have ${search}s!\n`);
      return;
    }
  }
  res.send(`sorry, we don't have any ${search}s :(\n`);
});
```

Save the changes to the file.

- 3.7. In the terminal window, make the same request to the snack endpoint as before.

```
[student@workstation nodebug]$ curl localhost:8080/snacks?search=apple
```

Step through the execution again and notice that the bug is not fixed. This is because only the application code on the host was updated. The container contents are unchanged.

▶ 4. Recreate and debug the container with the application code mounted into the container.

- 4.1. In the terminal window, stop the container.

```
[student@workstation nodebug]$ podman stop nodebug
nodebug
```

This also stops the VSCode debug connection.

- 4.2. Install the Node.js dependencies on the host machine.

```
[student@workstation nodebug]$ npm install
...output omitted...
```

- 4.3. Create a container with the application code mounted into the container.

```
[student@workstation nodebug]$ podman run -d --rm \
--name nodebug -p 8080:8080 -p 9229:9229 \
-v ./opt/app-root/src:z \
nodebug npm run debug
9eba...9c81
```

**Note**

Installing the Node.js dependencies in the host machine is necessary because the preceding `-v` option overrides the entire application directory from the container image, which includes runtime dependencies.

- 4.4. Connect VSCode to the debug socket by clicking the **Start Debugging** icon.
- 4.5. In the terminal window, make the same request to the snack endpoint as before.

```
[student@workstation ~]$ curl localhost:8080/snacks?search=apple
```

Step through the execution.

Notice that the bug is fixed because the updated code on the host was mounted inside the container:

```
[student@workstation nodebug]$ curl localhost:8080/snacks?search=apple
yes, we have apples!
```

This set up provides faster debugging by not requiring a container rebuild on every code change.

- 4.6. Disconnect the debugger by clicking **Disconnect** or by closing the VSCode window.

```
JS index.js x ... | x.js | {} launch-d
JS index.js > ...
1 const express = require("express");
2
3 const PORT_NUMBER = 8080;
4
5 const app = express();
6
```

Figure 6.4: Disconnect the debugger

- 5. Create a container image with the updated code and verify that the bug is fixed.

- 5.1. Stop and remove the running debug container.

```
[student@workstation nodebug]$ podman rm -f nodebug
```

- 5.2. Remove the `node_modules` directory from the host.

```
[student@workstation nodebug]$ rm -r node_modules
no output expected
```

- 5.3. Build a container image with the fixed application code.

```
[student@workstation nodebug]$ podman build -t nodebug .
...output omitted...
Successfully tagged localhost/nodebug:latest
...output omitted...
```

- 5.4. Create a container with the new image. Run the application without debug mode and without exposing the debug port.

```
[student@workstation nodebug]$ podman run -d --rm \
--name nodebug -p 8080:8080 nodebug
8ef...eb7
```

- 5.5. Make the same request to the application as before and verify that the response is correct.

```
[student@workstation nodebug]$ curl localhost:8080/snacks?search=apple
yes, we have apples!
```

Finish

On the **workstation** machine, use the **lab** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish troubleshooting-debugging
```

▶ Lab

Troubleshooting Containers

Troubleshoot the quotes-api application.

The application consists of the following components:

- The quotes-api-v1 service with the v1 quotes API version.
- The quotes-api-v2 service with the v2 quotes API version.
- The quotes-ui service, which accepts the quotes API version as an environment variable.

The quotes-ui container starts an NGINX proxy that performs the following tasks:

- Serves the web application.
- Acts as a reverse proxy to route the UI requests to the version specified in the QUOTES_API_VERSION environment variable.

Outcomes

You should be able to:

- Check container logs.
- Run container commands.
- Troubleshoot container networking problems.
- Configure containers by using environment variables.
- Configure containers by overriding container configuration files with host files.

Before You Begin

As the student user on the workstation machine, use the lab command to prepare your system for this exercise.

This command deploys the quotes application in a non-working state. You must troubleshoot and fix the application.

```
[student@workstation ~]$ lab start troubleshooting-lab
```

The lab script continuously evaluates the objectives of this lab. Keep the script running in a terminal window and complete the objectives of this lab from a new terminal window.

Instructions

1. Verify that the following containers are running with the following configuration:

quotes-api-v1 **and** quotes-api-v2

- Container image: registry.ocp4.example.com:8443/redhattraining/wiremock

quotes-ui

- Port mapping: 3000:8080

- Container image: `registry.ocp4.example.com:8443/redhattraining/quotes-ui-versioning:1.0`
- Environment variable: `QUOTES_API_VERSION=v2`

If any of the containers are missing, then read the container logs to determine what is preventing the container from starting. Then, fix the issue and start the container.

Use the following troubleshooting guide to identify and fix the problems:

Problem: unable to retrieve auth token: invalid username/password: unauthorized
Solution: You must log into the image registry.

Problem: requires at least 1 arg(s), only received 0
Solution: Correct your Podman command.

Problem: nginx: [emerg] host not found in upstream
Solution: Nginx reverse proxy cannot resolve hostname to IP address, probably due to a missing Podman network.

- Ensure that the NGINX reverse proxy can route requests to both versions of the quotes API.

The NGINX reverse proxy in the `quotes-ui` container should route requests as indicated in the following table:

NGINX mappings

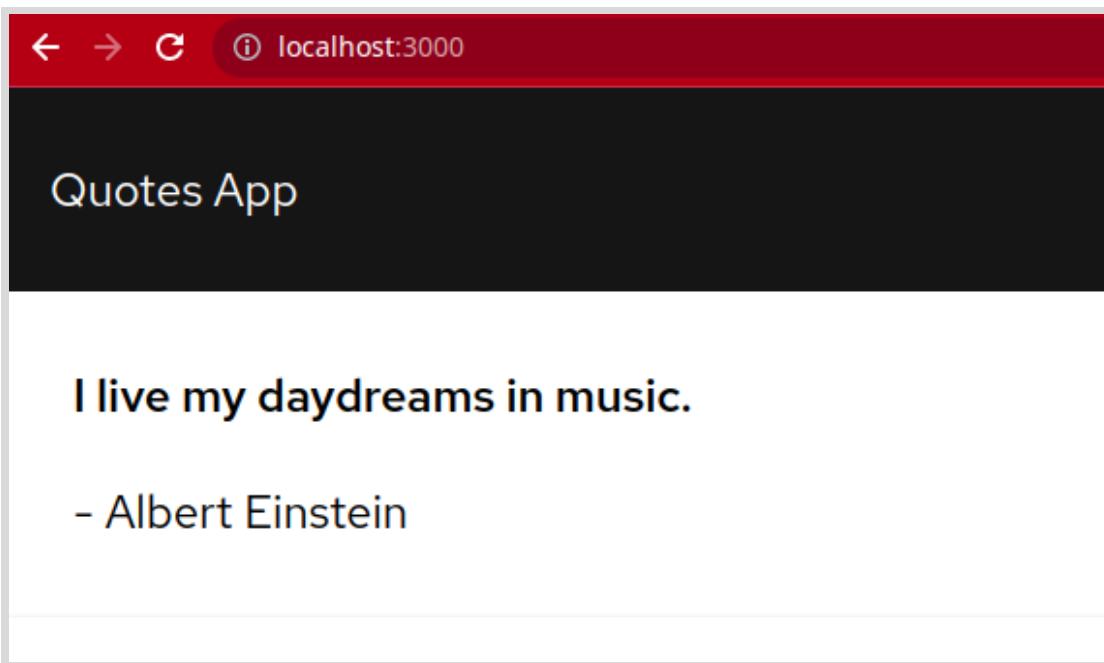
From	To
<code>http://localhost:8080/api/v1/quotes</code>	<code>http://quotes-api-v1:8080/api/v1/quotes</code>
<code>http://localhost:8080/api/v2/quotes</code>	<code>http://quotes-api-v2:8081/api/v2/quotes</code>

The `quotes-ui` logs provide some network information that you might need.

Perform any configuration modifications in the `quotes-ui` container.

Use a bind mount to replace the `/etc/nginx/nginx.conf` file in the `quotes-ui` container with a modified `~/D0188/labs/troubleshooting-lab/nginx.conf` file, if necessary.

3. Confirm that the application is accessible by using a web browser to navigate to `http://localhost:3000`.



Finish

As the student user on the `workstation` machine, use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

Press `y` when the `lab start` command prompts you to execute the `finish` function. Alternatively, execute the following command:

```
[student@workstation ~]$ lab finish troubleshooting-lab
```

► Solution

Troubleshooting Containers

Troubleshoot the quotes-api application.

The application consists of the following components:

- The quotes-api-v1 service with the v1 quotes API version.
- The quotes-api-v2 service with the v2 quotes API version.
- The quotes-ui service, which accepts the quotes API version as an environment variable.

The quotes-ui container starts an NGINX proxy that performs the following tasks:

- Serves the web application.
- Acts as a reverse proxy to route the UI requests to the version specified in the QUOTES_API_VERSION environment variable.

Outcomes

You should be able to:

- Check container logs.
- Run container commands.
- Troubleshoot container networking problems.
- Configure containers by using environment variables.
- Configure containers by overriding container configuration files with host files.

Before You Begin

As the student user on the workstation machine, use the lab command to prepare your system for this exercise.

This command deploys the quotes application in a non-working state. You must troubleshoot and fix the application.

```
[student@workstation ~]$ lab start troubleshooting-lab
```

The lab script continuously evaluates the objectives of this lab. Keep the script running in a terminal window and complete the objectives of this lab from a new terminal window.

Instructions

1. Verify that the following containers are running with the following configuration:

quotes-api-v1 **and** quotes-api-v2

- Container image: `registry.ocp4.example.com:8443/redhattraining/wiremock`

quotes-ui

- Port mapping: `3000:8080`

Chapter 6 | Troubleshooting Containers

- Container image: `registry.ocp4.example.com:8443/redhattraining/quotes-ui-versioning:1.0`
- Environment variable: `QUOTES_API_VERSION=v2`

If any of the containers are missing, then read the container logs to determine what is preventing the container from starting. Then, fix the issue and start the container.

Use the following troubleshooting guide to identify and fix the problems:

Problem: unable to retrieve auth token: invalid username/password: unauthorized
Solution: You must log into the image registry.

Problem: requires at least 1 arg(s), only received 0
Solution: Correct your Podman command.

Problem: nginx: [emerg] host not found in upstream
Solution: Nginx reverse proxy cannot resolve hostname to IP address, probably due to a missing Podman network.

- 1.1. Run the `podman ps` command with the `-a` option to show stopped containers.

```
[student@workstation ~]$ podman ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
b72652504844 ...wiremock:latest 28 sec... Up... quotes-api-v1
a3a2f6809446 ...wiremock:latest 27 sec... Up... quotes-api-v2
867f6f559629 ...quotes-ui-ve... /bin... 21 sec... Exited... 0... quotes-ui
```

The `quotes-ui` container failed to start.

- 1.2. Read the `quotes-ui` logs with the `podman logs` command to look for container start-up errors.

```
[student@workstation ~]$ podman logs quotes-ui
nginx: [emerg] host not found in upstream "quotes-api-v1" in /etc/nginx/
nginx.conf:45
```

The container fails to start because the NGINX proxy in the `quotes-ui` container cannot resolve the `quotes-api-v1` hostname to an IP address.

- 1.3. Use the `podman inspect` command to determine the Podman networks that the containers use to communicate.

Inspect the `quotes-api-v1` networks.

```
[student@workstation ~]$ podman inspect \
quotes-api-v1 --format='{{.NetworkSettings.Networks}}'
map[ttroubleshooting-lab:0xc000a825a0]
```

Inspect the `quotes-api-v2` networks.

```
[student@workstation ~]$ podman inspect \
quotes-api-v2 --format='{{.NetworkSettings.Networks}}'
map[ttroubleshooting-lab:0xc000a825a0]
```

Repeat the command for the quotes-ui container.

```
[student@workstation ~]$ podman inspect \
  quotes-ui --format='{{.NetworkSettings.Networks}}'
map[]
```

The quotes-ui container is missing the troubleshooting-lab network and thus it cannot resolve the quotes-api-v1 and quotes-api-v2 host names.

- 1.4. Remove the current quotes-ui container.

```
[student@workstation ~]$ podman rm quotes-ui
093a...cf4a
```

- 1.5. Recreate the quotes-ui container and attach it to the troubleshooting-lab network.

```
[student@workstation ~]$ podman run -d \
  --name quotes-ui -p 3000:8080 \
  -e QUOTES_API_VERSION=v2 \
  --net troubleshooting-lab \
  registry.ocp4.example.com:8443/redhattraining/quotes-ui-versioning:1.0
d838...7432
```

- 1.6. Verify that the quotes-ui container starts.

```
[student@workstation ~]$ podman ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
b72652504844 ...wiremock:latest 28 sec... Up... quotes-api-v1
a3a2f6809446 ...wiremock:latest 27 sec... Up... quotes-api-v2
6e8581a9d73b ...quotes-ui-versioning:1.0 1 sec... Up... quotes-ui
```

2. Ensure that the NGINX reverse proxy can route requests to both versions of the quotes API.

The NGINX reverse proxy in the quotes-ui container should route requests as indicated in the following table:

NGINX mappings

From	To
<code>http://localhost:8080/api/v1/quotes</code>	<code>http://quotes-api-v1:8080/api/v1/quotes</code>
<code>http://localhost:8080/api/v2/quotes</code>	<code>http://quotes-api-v2:8081/api/v2/quotes</code>

The quotes-ui logs provide some network information that you might need.

Perform any configuration modifications in the quotes-ui container.

Use a bind mount to replace the /etc/nginx/nginx.conf file in the quotes-ui container with a modified ~/D0188/labs/troubleshooting-lab/nginx.conf file, if necessary.

- 2.1. Use the `podman exec` command to run the `curl` command from the `quotes-ui` container to test the v1 and v2 mappings. Use the `localhost` host name to test the NGINX mappings. Use the `silent` option (`-s`) with the `curl` command to print the output only.

Test that the v1 mapping works.

```
[student@workstation ~]$ podman exec quotes-ui \
  curl -s http://localhost:8080/api/v1/quotes
[
  {
    "id": 1,
    "quote": "I live my daydreams in music.",
    "author": "Albert Einstein"
  },
  ...output omitted...
```

Test that the v2 mapping does not work.

```
[student@workstation ~]$ podman exec quotes-ui \
  curl -s http://localhost:8080/api/v2/quotes
<html>
<head><title>502 Bad Gateway</title></head>
<body>
<center><h1>502 Bad Gateway</h1></center>
<hr><center>nginx/1.20.1</center>
</body>
</html>
```

This error means that NGINX was able to map the request but did not get a response from the `quotes-api-v2` service.

- 2.2. Investigate if the `quotes-api-v2` service is using a different port. Run the `podman logs` command on the `quotes-api-v2` container to find the port that the service is using.

```
[student@workstation ~]$ podman logs quotes-api-v2
...output omitted...
port:          8081
...output omitted...
```

The `quotes-api-v2` service is using the 8081 port.

- 2.3. Read the `quotes-ui` container NGINX configuration file at the `/etc/nginx/nginx.conf` path.

```
[student@workstation ~]$ podman exec quotes-ui \
  cat /etc/nginx/nginx.conf
...output omitted...
location /api/v2 {
  rewrite /api/v2/(.*) /$1 break;
  proxy_pass http://quotes-api-v2:8080;
}
...output omitted...
```

The mapping for v2 is redirecting requests to the `http://quotes-api-v2:8080` address, which points to the wrong port.

- 2.4. Read the updated copy of the NGINX configuration at `~/D0188/labs/troubleshooting-lab/nginx.conf`.

```
[student@workstation ~]$ cat ~/D0188/labs/troubleshooting-lab/nginx.conf
...output omitted...
location /api/v2 {
    rewrite /api/v2/(.*) /$1 break;
    proxy_pass http://quotes-api-v2:8081;
}
...output omitted...
```

This version points to the right 8081 port.

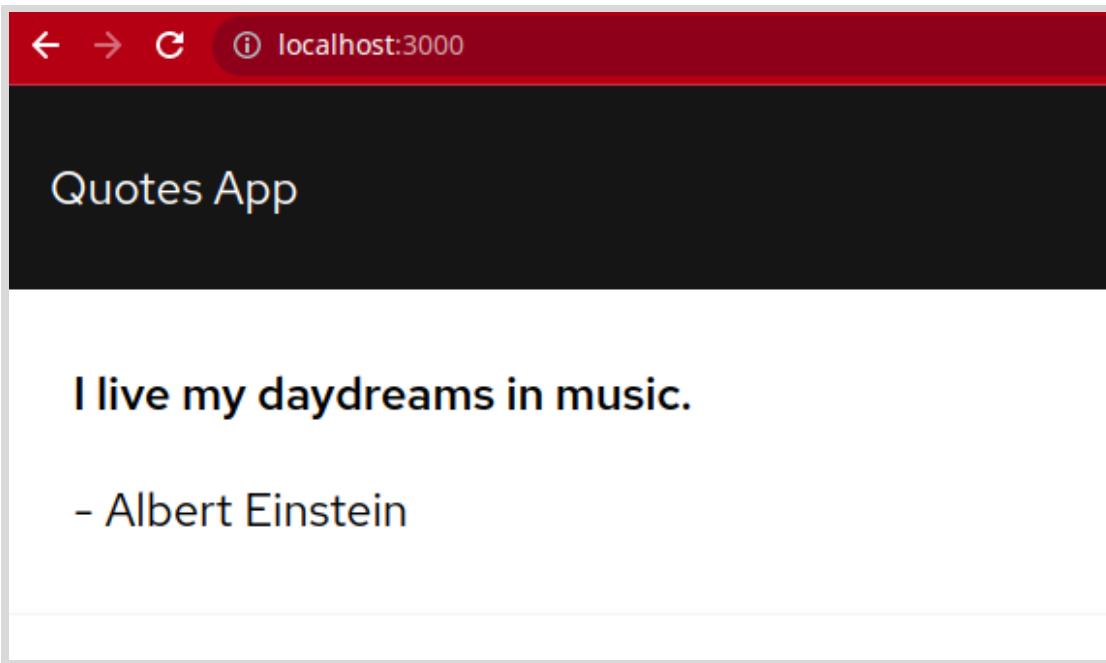
- 2.5. Remove the quotes-ui container by running `podman rm` using the `-f` option.

```
[student@workstation ~]$ podman rm -f quotes-ui
d838...7432
```

- 2.6. Recreate the quotes-ui container. Override the container's `/etc/nginx/nginx.conf` file with the `~/D0188/labs/troubleshooting-lab/nginx.conf` file by using a bind mount with the `:Z` option.

```
[student@workstation ~]$ podman run -d \
--name quotes-ui \
-p 3000:8080 \
-e QUOTES_API_VERSION=v2 \
--net troubleshooting-lab \
-v ~/D0188/labs/troubleshooting-lab/nginx.conf:/etc/nginx/nginx.conf:Z \
registry.ocp4.example.com:8443/redhattraining/quotes-ui-versioning:1.0
a0d1...a73f
```

3. Confirm that the application is accessible by using a web browser to navigate to `http://localhost:3000`.



Finish

As the **student** user on the **workstation** machine, use the **lab** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

Press **y** when the **lab start** command prompts you to execute the **finish** function. Alternatively, execute the following command:

```
[student@workstation ~]$ lab finish troubleshooting-lab
```

Summary

- You can view logs for an application running in a container.
- Using a debugger speeds up application troubleshooting.
- You can attach a debugger to an application running in a container.

Chapter 7

Multi-container Applications with Compose

Goal

Run multi-container applications with Podman Compose.

Objectives

- Describe compose and its common use cases.
- Configure a repeatable developer environment with Compose.

Sections

- Compose Overview and Use Cases (and Quiz)
- Build Developer Environments with Compose (and Guided Exercise)

Lab

- Multi-container Applications with Compose

Compose Overview and Use Cases

Objectives

- Describe compose and its common use cases.

Orchestrate Containers with Podman Compose

In today's computing, many applications are developed as microservices. In the context of containers, every microservice is a container image, which developers manage as an independent container. Because one application is composed of several containers, it might become difficult to manage the containers as a group.

For example, to develop a new microservice, developers commonly implement one of the following strategies:

- Create a *mock* of the rest of the application, such as API endpoints that other microservices provide.
- Start the relevant parts of the application on a local machine.

It is ideal to run the full application on the local machine. When doing so is not possible, developers containerize mock services that approximate production environment.

Podman Compose is an open source tool that you can use to run *Compose files*. A Compose file is a YAML file that specifies the containers to manage, as well as the dependencies among them.

For example, consider the following Compose file:

```
services:
  orders: ①
    image: quay.io/user/python-app ②
    ports:
      - 3030:8080 ③
    environment:
      ACCOUNTS_SERVICE: http://accounts ④
```

- Declare the `orders` container.
- Use the `python-app` container image.
- Bind the `3030` port on your host machine to the `8080` port within the container.
- Pass the `ACCOUNTS_SERVICE` environment variable to the application.

Podman Compose complies with the *Compose Specification*, which defines a YAML-based schema to manage multi-container applications for container runtimes, such as Podman.

Although Compose files should work with any container runtime, some features might depend on specific implementations of the runtime. You might find Compose files called `docker-compose.yaml`. Docker introduced the `docker-compose.yaml` naming convention when it started the Compose project. However, Docker open sourced the Compose Specification, which

also specifies the Compose file naming convention. According to the Compose Specification, the preferred name is `compose.yaml`.

Podman Compose became popular for the following uses:

Development environments

You can approximate a production environment on your local machine by providing applications, databases, or cache systems configuration in a single file.

Consequently, developers can automate complex multi-container deployments with one file.

Automated testing

You can define several test suites, along with their configurations and environment variables, within the same file. In addition, you might approximate an automated pipeline environment on a single machine.

Consequently, developers can execute their applications with multiple isolated database environments, and clients. Developers can also debug complex issues in their local environment, which increases development speed.

Using Podman Compose in a production environment is not recommended. Podman Compose does not support advanced functions that you might need in a production environment, such as load balancing, distributing containers to multiple nodes, managing containers on different nodes, and others.

If you need a production container orchestration solution, then Kubernetes or Red Hat OpenShift is a better option. Red Hat OpenShift can run multi-container applications on different host machines, or *nodes*.

Podman Pods

Podman introduced an alternative to orchestrate containers on a single host with the usage of Podman pods.

Podman pods can use and produce Kubernetes YAML files. These Kubernetes YAML files are a declarative way to define and manage multiple containers and their resources.

The `podman generate kube` command generates a Kubernetes YAML file from existing Podman containers, pods, and volumes. The `podman play kube` command reads the Kubernetes YAML file and then recreates the defined resources on a local machine. These commands help to ensure that a container that is developed with Podman can migrate and work in a Kubernetes ecosystem.

The Compose File

The Compose file is a YAML file that contains the following sections:

- `version` (deprecated): Specifies the Compose version used.
- `services`: Defines the containers used.
- `networks`: Defines the networks used by the containers.
- `volumes`: Specifies the volumes used by the containers.
- `configs`: Specifies the configurations used by the containers.
- `secrets`: Defines the secrets used by the containers.

The `secrets` and `configs` objects are mounted as a file in the containers.

**Warning**

In YAML files, the number of spaces carries meaning.

For example, consider the following YAML snippet:

```
version: "3.9"
services:
backend: {}
```

The previous YAML snippet carries a different meaning from the following snippet:

```
version: "3.9"
services:
  backend: {}
```

The `backend` keyword is preceded by two spaces; therefore the `backend` object is an attribute of the `services` object.

The following Compose file defines the `backend` and `db` services. It overrides the default command for the `backend` image by specifying the `command` property. Finally, the Compose file configures the environment variables that are required to start the database container.

**Note**

This section uses the terms `service` and `container` interchangeably.

```
services:
  backend:
    image: quay.io/example/backend
    ports:
      - "8081:8080"
    command: sh -c "COMMAND"
  db:
    image: registry.redhat.io/rhel8/postgresql-13
    environment:
      POSTGRESQL_ADMIN_PASSWORD: redhat
```

Start and Stop Containers with Podman Compose

You can execute a Compose file by using the `podman-compose up` command, which creates the defined objects, such as volumes or networks, and starts the defined containers.

```
[user@host ~]$ podman-compose up
['podman', '--version', '']
using podman version: 4.2.0
** excluding: set()
['podman', 'network', 'exists', 'user_default'] ①
['podman', 'network', 'create', '--label', 'io.podman.compose.project=user', '--label', 'com.docker.compose.project=user', 'user_default'] ②
...output omitted...
```

- ① Check whether the user_default network exists.
- ② Create the user_default Podman network.

You can list objects that are defined in the Compose file by using the podman command, such as podman network ls to list Podman networks.

```
[user@host ~]$ podman network ls
NETWORK ID      NAME          VERSION      PLUGINS
cd32c76f42f9    user_default   0.4.0        bridge, portmap, firewall, tuning, dnsname
```

Podman generates predictable names for objects that are not explicitly named. In the preceding example, Podman creates a default network for the application.

The default network naming convention uses the current directory name and the _default suffix. The preceding example contains the network user_default because it shows a Compose file executed in the /home/user home directory.

The default container naming convention uses the *DIRECTORY_SERVICE_NUMBER* format. Because the preceding example does not configure the container name, it creates the user_db_1 container.

You can override the default naming conventions by explicitly configuring object names.

The following list shows common podman-compose up command options in their short and long versions:

- -d, --detach: Start containers in the background.
- --force-recreate: Re-create containers on start.
- -V, --renew-anon-volumes: Re-create anonymous volumes.
- --remove-orphans: Remove containers that do not correspond to services that are defined in the current Compose file.

The podman-compose stop command stops running containers that are defined as services. Execute podman-compose down to stop and remove containers that are defined as services.

```
[user@host ~]$ podman-compose down
['podman', '--version', '']
using podman version: 4.2.0
** excluding: set()
podman stop -t 10 compose_db_1
compose_db_1
exit code: 0
podman rm compose_db_1
```

```
aab3...b412
exit code: 0
...output omitted...
```

Podman preserves objects other than containers, such as networks and volumes, so that containers do not lose their local state when restarted.

Networking

Use the `nets` keyword at the same indentation level as the `services` keyword to create and use Podman networks. If the `nets` keyword is not defined in the Podman Compose file, then Podman Compose creates a default DNS-enabled network.

Consider the following Compose file that declares three containers: a front-end application, a back-end application, and a database.

```
services:
  frontend:
    image: quay.io/example/frontend
    networks: ①
      - app-net
    ports:
      - "8082:8080"
  backend:
    image: quay.io/example/backend
    networks: ②
      - app-net
      - db-net
  db:
    image: registry.redhat.io/rhel8/postgresql-13
    environment:
      POSTGRESQL_ADMIN_PASSWORD: redhat
    networks: ③
      - db-net

  networks: ④
    app-net: {}
    db-net: {}
```

- ① The `frontend` service is part of the `app-net` network.
- ② The `backend` service is part of the `app-net` and `db-net` networks.
- ③ The `db` service is part of the `db-net`.
- ④ Definition of the networks.



Note

The open and closed curly braces `{}` mark an empty object. For example, the `app-net: {}` and `app-net: []` definitions are identical. Both definitions signal the creation of the `app-net` network that uses the default configuration.

Services can interact only if they share at least one network. Consequently, the `frontend` service cannot communicate directly with the `db` service. Isolating network traffic provides security advantages, because you can prevent specific containers from accessing protected data.

Volumes

You can declare volumes by using the `volumes` keyword. Mount the volumes in containers by using the `VOLUME_NAME:CONTAINER_DIRECTORY:OPTIONS` syntax.

```
services:
  db:
    image: registry.redhat.io/rhel8/postgresql-13
    environment:
      POSTGRESQL_ADMIN_PASSWORD: redhat
    ports:
      - "5432:5432"
    volumes: ❶
      - db-vol:/var/lib/postgresql/data

  volumes: ❷
    db-vol: {}
```

- ❶ The `db` service maps the `db-vol` volume to the `/var/lib/postgresql/data` directory within the container.
- ❷ Declaration of volumes.

If you created a volume that is not managed by Podman Compose, for example, because you used the `podman volume create` command, then you can specify it in the `volumes` definition.

```
services:
  db:
    image: registry.redhat.io/rhel8/postgresql-13
    environment:
      POSTGRESQL_ADMIN_PASSWORD: redhat
    ports:
      - "5432:5432"
    volumes:
      - my-volume:/var/lib/postgresql/data

  volumes:
    my-volume:
      external: true
```

You can also define bind mounts by providing a relative or absolute path on your host machine. In the following example, Podman mounts the `./local/redhat` directory on the host machine as the `/var/lib/postgresql/data` directory in the container.

```
services:
  db:
    image: registry.redhat.io/rhel8/postgresql-13
    environment:
      POSTGRESQL_ADMIN_PASSWORD: redhat
    ports:
```

```
- "5432:5432"  
volumes:  
- ./local/redhat:/var/lib/postgresql/data:z
```



References

GitHub - Podman Compose

<https://github.com/containers/podman-compose>

GitHub - Compose Spec

<https://github.com/compose-spec/compose-spec>

Podman Compose Vs Docker Compose

<https://www.redhat.com/sysadmin/podman-compose-docker-compose>

► Quiz

Compose Overview and Use Cases

Choose the correct answers to the following questions:

► 1. Based on your understanding of Podman Compose, which statement presents a suitable use for Podman Compose?

- a. Automate deployment of multi-container applications for a multi-node production environment.
- b. Automate deployment of multi-container applications for local development.
- c. Stop using Podman to gain security benefits of Podman Compose.
- d. Stop using Podman to gain network isolation benefits of Podman Compose.

► 2. Based on the provided output of Podman Compose, which statement is correct?

```
['podman', 'network', 'exists', 'compose_app-net']
podman run --name=compose_frontend_1 -d --net compose_app-net --network-alias
frontend -p 8082:8080 quay.io/redhattraining/ubi-htt
64029...ffe1
exit code: 0
['podman', 'network', 'exists', 'compose_app-net']
['podman', 'network', 'exists', 'compose_db-net']
podman run --name=compose_backend_1 -d --net compose_db-net,compose_app-net --
network-alias backend -p 8081:8080 quay.io/redhattraining/podman-quotes:quarkus
2023...13a1
exit code: 0
['podman', 'network', 'exists', 'compose_db-net']
podman run --name=compose_db_1 -d -e POSTGRESQL_ADMIN_PASSWORD=redhat --net
compose_db-net --network-alias db -p 8001:5432 registry.redhat.io/rhel8/
postgresql-13
e15b...c0da
exit code: 0
```

- a. The application failed to start because some objects defined in the Compose file already exist, such as the network.
- b. The application consists of two containers.
- c. The front-end container can reach the database container.
- d. The front-end container uses two networks.
- e. The back-end container can reach the database and front-end containers.
- f. The Compose file exposes two containers on the port 8080.

► **3. Which statement about networks in Podman Compose is correct?**

- a. You cannot limit network traffic among containers in a single Compose file. To limit such traffic, you must create multiple Compose files.
- b. You must use at most one network per container.
- c. Containers cannot communicate over the network by default. Create a default network to allow network traffic.
- d. Containers can communicate over the network by default. Define additional networks to further limit the network traffic.

► 4. Based on the provided Compose file, which three statements are correct? (Choose three.)

```
services:  
  db-admin:  
    image: quay.io/pgadmin4  
    environment:  
      PGADMIN_DEFAULT_EMAIL: user@redhat.com  
      PGADMIN_DEFAULT_PASSWORD: redhat  
    ports:  
      - "9876:80"  
  db:  
    image: registry.redhat.io/rhel8/postgresql-13  
    environment:  
      POSTGRESQL_USER: redhat  
      POSTGRESQL_DATABASE: db  
      POSTGRESQL_PASSWORD: redhat  
    ports:  
      - "5432:5423"  
    volumes:  
      - db-vol:/var/lib/postgresql/data  
  
volumes:  
  db-vol: {}
```

- a. The db-admin container can send traffic to the db container by using the db hostname.
- b. The db-admin container cannot communicate with the db container. To fix the issue, you must define a new, common network.
- c. The db-admin container is accessible from the host machine on port 80.
- d. The db-admin container is accessible from the host machine on port 9876.
- e. The db container must expose the port 5432. Otherwise, the db-admin container cannot send traffic to that port.
- f. The traffic from the db-admin container to the db container uses a default network. Therefore, exposing the 5432 port is not necessary unless you intend to access the DB from the host network.
- g. The Compose file is syntactically wrong because the db-vol: {} line is incorrect. You must change it to db-vol: to fix the issue.

► **5. Which three statements about Podman Compose are correct? (Choose three.)**

- a. Execute `podman-compose up` to create and start containers that are defined in the Compose file.
- b. Execute `podman compose start` to create and start containers that are defined in the Compose file.
- c. Execute `podman-compose down` to stop and delete containers that are defined in the Compose file.
- d. Execute `podman-compose stop` to stop and delete containers that are defined in the Compose file.
- e. Execute `podman-compose stop` to stop containers that are defined in the Compose file.

► Solution

Compose Overview and Use Cases

Choose the correct answers to the following questions:

► 1. Based on your understanding of Podman Compose, which statement presents a suitable use for Podman Compose?

- a. Automate deployment of multi-container applications for a multi-node production environment.
- b. Automate deployment of multi-container applications for local development.
- c. Stop using Podman to gain security benefits of Podman Compose.
- d. Stop using Podman to gain network isolation benefits of Podman Compose.

► 2. Based on the provided output of Podman Compose, which statement is correct?

```
[ 'podman', 'network', 'exists', 'compose_app-net' ]
podman run --name=compose_frontend_1 -d --net compose_app-net --network-alias
frontend -p 8082:8080 quay.io/redhattraining/ubi-httdp
64029...ffe1
exit code: 0
[ 'podman', 'network', 'exists', 'compose_app-net' ]
[ 'podman', 'network', 'exists', 'compose_db-net' ]
podman run --name=compose_backend_1 -d --net compose_db-net,compose_app-net --
network-alias backend -p 8081:8080 quay.io/redhattraining/podman-quotes:quarkus
2023...13a1
exit code: 0
[ 'podman', 'network', 'exists', 'compose_db-net' ]
podman run --name=compose_db_1 -d -e POSTGRESQL_ADMIN_PASSWORD=redhat --net
compose_db-net --network-alias db -p 8001:5432 registry.redhat.io/rhel8/
postgresql-13
e15b...c0da
exit code: 0
```

- a. The application failed to start because some objects defined in the Compose file already exist, such as the network.
- b. The application consists of two containers.
- c. The front-end container can reach the database container.
- d. The front-end container uses two networks.
- e. The back-end container can reach the database and front-end containers.
- f. The Compose file exposes two containers on the port 8080.

► **3. Which statement about networks in Podman Compose is correct?**

- a. You cannot limit network traffic among containers in a single Compose file. To limit such traffic, you must create multiple Compose files.
- b. You must use at most one network per container.
- c. Containers cannot communicate over the network by default. Create a default network to allow network traffic.
- d. Containers can communicate over the network by default. Define additional networks to further limit the network traffic.

► 4. Based on the provided Compose file, which three statements are correct? (Choose three.)

```

services:
  db-admin:
    image: quay.io/pgadmin4
    environment:
      PGADMIN_DEFAULT_EMAIL: user@redhat.com
      PGADMIN_DEFAULT_PASSWORD: redhat
    ports:
      - "9876:80"
  db:
    image: registry.redhat.io/rhel8/postgresql-13
    environment:
      POSTGRESQL_USER: redhat
      POSTGRESQL_DATABASE: db
      POSTGRESQL_PASSWORD: redhat
    ports:
      - "5432:5423"
  volumes:
    - db-vol:/var/lib/postgresql/data

volumes:
  db-vol: {}

```

- The db-admin container can send traffic to the db container by using the db hostname.
- The db-admin container cannot communicate with the db container. To fix the issue, you must define a new, common network.
- The db-admin container is accessible from the host machine on port 80.
- The db-admin container is accessible from the host machine on port 9876.
- The db container must expose the port 5432. Otherwise, the db-admin container cannot send traffic to that port.
- The traffic from the db-admin container to the db container uses a default network. Therefore, exposing the 5432 port is not necessary unless you intend to access the DB from the host network.
- The Compose file is syntactically wrong because the db-vol: {} line is incorrect. You must change it to db-vol: to fix the issue.

► **5. Which three statements about Podman Compose are correct? (Choose three.)**

- a. Execute `podman-compose up` to create and start containers that are defined in the Compose file.
- b. Execute `podman compose start` to create and start containers that are defined in the Compose file.
- c. Execute `podman-compose down` to stop and delete containers that are defined in the Compose file.
- d. Execute `podman-compose stop` to stop and delete containers that are defined in the Compose file.
- e. Execute `podman-compose stop` to stop containers that are defined in the Compose file.

Build Developer Environments with Compose

Objectives

- Configure a repeatable developer environment with Compose.

Podman Compose and Podman

Podman Compose translates Compose files into Podman CLI commands. Through translated commands, Podman Compose interacts directly with Podman and does not communicate with Podman's API socket. In contrast, Docker Compose communicates directly with the Docker daemon by using its REST API.

By not interacting with the Podman's API socket, Podman Compose removes the need to run the Podman service that provides the API socket, saving resource consumption and providing a more native and lightweight solution for Podman users. Additionally, because it interacts with Podman directly, Podman Compose makes better use of rootless containers and pods than Docker Compose.

Before you use Podman Compose, install a compatible Podman version on your system. Podman Compose recommends using a modern version of Podman that is version 3.4 or later. This course uses Podman 4.2.0.

You can confirm the Podman version with the `podman --version` command.

```
[user@host ~]$ podman --version
podman version 4.2.0
```

Podman Compose is also available as a Python package. You can install it with the `pip3` command, as follows:

```
[user@host ~]$ pip3 install podman-compose
```

You can also install the latest development version from the Podman Compose GitHub repository:

```
[user@host ~]$ pip3 install https://github.com/containers/podman-compose/archive/
devel.tar.gz
```

Multi-container Developer Environments with Compose

With Podman Compose, you can declaratively configure the services that you need to develop your application. You can place the required services into the Compose file inside your application repository so that developers can start these services by issuing a single `podman-compose up` command. Running your development environment dependencies like this isolates the developer from shared developer environments. Also, this predefined configuration saves developers from having to locally configure services such as databases or external dependencies.

For example, to develop a back-end application, you can use Podman Compose and a single Compose file to deploy a development environment that includes a PostgreSQL database container and a pgAdmin interface container to manage the database. You can mount data loading scripts for the database to provide developers with the required development data. You can specify the service dependencies by using the `depends_on` property followed by a list of services that need to start first.

```
services:  
  database:  
    image: "registry.redhat.io/rhel9/postgresql-13"  
    container_name: "appdev-postgresql"  
    volumes:  
      - ./scripts/database/initial-data:/opt/app-root/src/postgresql-start:Z  
...output omitted...  
  database-admin:  
    image: "registry.connect.redhat.com/crunchydata/crunchy-pgadmin4:ubi8-4.30-1"  
    container_name: "appdev-pgadmin"  
    depends_on:  
      - database  
...output omitted...
```

The previous example demonstrates that the `database-admin` container must start after the `database` container.

If your application depends on external services, then you can add a mock server to your Compose file. You can provide the mock server endpoints and fixed responses as configuration files in your repository. Then you can bind mount these files by using relative paths to your project directory within your Compose file.

```
services:  
...output omitted...  
  mock_service:  
    image: "MOCK_SERVICE_IMAGE"  
    volumes:  
      - ./mocks/SERVICE/ENDPOINTS:TARGET_DIRECTORY:Z  
      - ./mocks/SERVICE/FIXED_RESPONSES:TARGET_DIRECTORY:Z  
...output omitted...
```



References

GitHub - Podman Compose

<https://github.com/containers/podman-compose>

► Guided Exercise

Build Developer Environments with Compose

Configure a repeatable developer environment with Podman Compose.

Outcomes

You should be able to:

- Create a compose file that contains the definition of a PostgreSQL server and a pgAdmin interface.
- Create a compose file that contains the definition of a pgAdmin interface.
- Start and run the developer environment.
- Access the pgAdmin interface from a web browser to retrieve the data from the tables.

Before You Begin

As the `student` user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

This command copies the necessary files for the development environment.

```
[student@workstation ~]$ lab start compose-environments
```

Instructions

- 1. Create a compose file that contains the definition of a PostgreSQL server.
- 1.1. Change to the `~/D0188/labs/compose-environments` directory and open the `compose.yml` file.
- ```
[student@workstation ~]$ cd ~/D0188/labs/compose-environments
[student@workstation compose-environments]$ gedit compose.yml
```
- 1.2. Define a database container that uses the `registry.ocp4.example.com:8443/rhel9/postgresql-13:1` image. Forward port 5432 from the localhost to the same port inside the container.

```
services:
 db:
 image: "registry.ocp4.example.com:8443/rhel9/postgresql-13:1"
 ports:
 - "5432:5432"
```

- 1.3. Define the following environment variables:

| Field               | Value     |
|---------------------|-----------|
| POSTGRESQL_USER     | backend   |
| POSTGRESQL_DATABASE | rpi-store |
| POSTGRESQL_PASSWORD | redhat    |

```
services:
 db:
 image: "registry.ocp4.example.com:8443/rhel9/postgresql-13:1"
 environment:
 POSTGRESQL_USER: backend
 POSTGRESQL_DATABASE: rpi-store
 POSTGRESQL_PASSWORD: redhat
 ports:
 - "5432:5432"
```

- 1.4. Bind mount the `~/D0188/labs/compose-environments/database_scripts` directory to the `/opt/app-root/src/postgresql-start` directory with the `Z` option for SELinux. You can use the relative path to the `compose.yml` file for the `database_scripts` directory as the bind mount.

```
services:
 db:
 image: "registry.ocp4.example.com:8443/rhel9/postgresql-13:1"
 environment:
 POSTGRESQL_USER: backend
 POSTGRESQL_DATABASE: rpi-store
 POSTGRESQL_PASSWORD: redhat
 ports:
 - "5432:5432"
 volumes:
 - ./database_scripts:/opt/app-root/src/postgresql-start:Z
```

- 1.5. Define a persistent volume called `rpi` for the container. Bind mount the `rpi` volume to the `/var/lib/pgsql/data` directory in the container.

```
services:
 db:
 image: "registry.ocp4.example.com:8443/rhel9/postgresql-13:1"
 environment:
 POSTGRESQL_USER: backend
 POSTGRESQL_DATABASE: rpi-store
 POSTGRESQL_PASSWORD: redhat
 ports:
 - "5432:5432"
 volumes:
 - ./database_scripts:/opt/app-root/src/postgresql-start:Z
 - rpi:/var/lib/pgsql/data
```

```
volumes:
 rpi: {}
```

1.6. Call the container `compose_environments_postgresql`, and save the file.

```
services:
 db:
 image: "registry.ocp4.example.com:8443/rhel9/postgresql-13:1"
 container_name: "compose_environments_postgresql"
 environment:
 POSTGRES_USER: backend
 POSTGRES_DATABASE: rpi-store
 POSTGRES_PASSWORD: redhat
 ports:
 - "5432:5432"
 volumes:
 - ./database_scripts:/opt/app-root/src/postgresql-start:Z
 - rpi:/var/lib/pgsql/data
 volumes:
 rpi: {}
```

► 2. Define a pgAdmin server in the `compose.yml` file.

2.1. Define a database admin interface container that uses the `registry.ocp4.example.com:8443/crunchydata/crunchy-pgadmin4:ubi8-4.30-1` image. Map port 5050 from the container to port 5050 on the host.

```
services:
 db-admin:
 image:
 "registry.ocp4.example.com:8443/crunchydata/crunchy-pgadmin4:ubi8-4.30-1"
 ports:
 - "5050:5050"
 db:
 image: "registry.ocp4.example.com:8443/rhel9/postgresql-13:1"
 container_name: "compose_environments_postgresql"
 environment:
 POSTGRES_USER: backend
 POSTGRES_DATABASE: rpi-store
 POSTGRES_PASSWORD: redhat
 ports:
 - "5432:5432"
 volumes:
 - ./database_scripts:/opt/app-root/src/postgresql-start:Z
 - rpi:/var/lib/pgsql/data

 volumes:
 rpi: {}
```

2.2. Define the following environment variables:

| Field                  | Value            |
|------------------------|------------------|
| PGADMIN_SETUP_EMAIL    | user@example.com |
| PGADMIN_SETUP_PASSWORD | redhat           |

```

services:
 db-admin:
 image: "registry.ocp4.example.com:8443/crunchydata/crunchy-
pgadmin4:ubi8-4.30-1"
 environment:
 PGADMIN_SETUP_EMAIL: user@example.com
 PGADMIN_SETUP_PASSWORD: redhat
 ports:
 - "5050:5050"
 db:
 image: "registry.ocp4.example.com:8443/rhel9/postgresql-13:1"
 container_name: "compose_environments_postgresql"
 environment:
 POSTGRESQL_USER: backend
 POSTGRESQL_DATABASE: rpi-store
 POSTGRESQL_PASSWORD: redhat
 ports:
 - "5432:5432"
 volumes:
 - ./database_scripts:/opt/app-root/src/postgresql-start:Z
 - rpi:/var/lib/pgsql/data

volumes:
 rpi: {}

```

2.3. Name the container `compose_environments_pgadmin`. Save and close the file.

```

services:
 db-admin:
 image: "registry.ocp4.example.com:8443/crunchydata/crunchy-
pgadmin4:ubi8-4.30-1"
 container_name: "compose_environments_pgadmin"
 environment:
 PGADMIN_SETUP_EMAIL: user@example.com
 PGADMIN_SETUP_PASSWORD: redhat
 ports:
 - "5050:5050"
 db:
 image: "registry.ocp4.example.com:8443/rhel9/postgresql-13:1"
 container_name: "compose_environments_postgresql"
 environment:
 POSTGRESQL_USER: backend
 POSTGRESQL_DATABASE: rpi-store
 POSTGRESQL_PASSWORD: redhat
 ports:
 - "5432:5432"

```

```
volumes:
 - ./database_scripts:/opt/app-root/src/postgresql-start:Z
 - rpi:/var/lib/pgsql/data

volumes:
 rpi: {}
```

**Note**

You can refer to the completed `compose.yml` file in the `~/D0188/solutions/compose-environments` directory.

► 3. Run the developer environment.

- 3.1. From the `~/D0188/labs/compose-environments` directory, use the `compose.yml` file to start the containerized development environment. Use the `-d` option to run the containers in the background.

```
[student@workstation compose-environments]$ podman-compose up -d
['podman', '--version', '']
using podman version: ...
** excluding: set()
['podman', 'network', 'exists', 'compose-environments_default']
...output omitted...
exit code: 0
```

3.2. Confirm that the two containers are running.

```
[student@workstation compose-environments]$ podman-compose ps
using podman version: ...
podman ps -a --filter label=io.podman.compose.project=compose-environments
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
d64b...5c6f registry... /opt/crunchy... 23 sec... Up... ...5050... ...pgadmin
91ae...474e registry... run-postgresql 23 sec... Up... ...5432... ...postg...
exit code: 0
```

3.3. Confirm that the persistent volumes exist.

```
[student@workstation compose-environments]$ podman volume list
DRIVER VOLUME NAME
local 91a6...f45d
local bd15...a5e2
local compose-environments_rpi
local f056...7eb0
```

**Note**

The command might display additional volumes from previous exercises.

- 3.4. Retrieve the logs from both containers and confirm that errors are not reported in the logs. Press `Ctrl+C` to exit the logs.

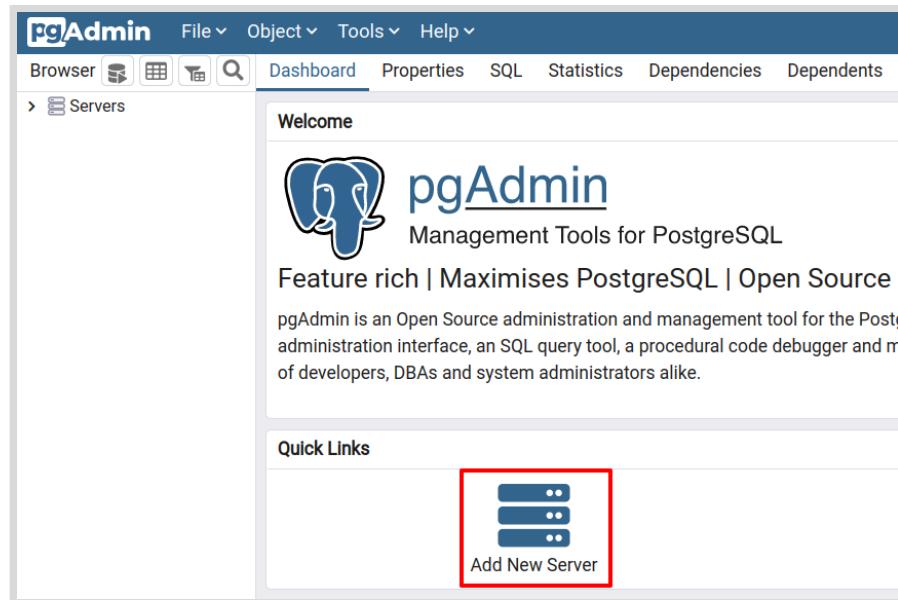
```
[student@workstation compose-environments]$ podman-compose logs -n -f
['podman', '--version', '']
using podman version: ...
podman logs -f -n compose_environments_pgadmin compose_environments_postgresql
...output omitted...
compose_environments_postgresql Starting server...
compose_environments_postgresql 2022-08-11 14:42:02.237 UTC [1] LOG: redirecting
log output to logging collector process
compose_environments_postgresql 2022-08-11 14:42:02.237 UTC [1] HINT: Future log
output will appear in directory "log".
...output omitted...
compose_environments_pgadmin Thu Aug 11 14:41:57 UTC 2022 INFO: Setting up
pgAdmin4 database..
compose_environments_pgadmin Thu Aug 11 14:42:05 UTC 2022 INFO: Starting Apache
web server..
```

- ▶ 4. Access the pgAdmin interface from a web browser. Retrieve and modify data from the database.

- 4.1. Open a web browser and navigate to <http://localhost:5050>. Access the pgAdmin interface as the `user@example.com` user with the `redhat` password.



- 4.2. Click **Add New Server** to connect to the `compose_environments_postgresql` database container.



- 4.3. In the **General** tab, set **rpi-store** as the name.
- 4.4. Switch to the **Connection** tab. Complete the form with the following data and leave the rest of the fields with their default values.

| Field             | Value   |
|-------------------|---------|
| Host name/address | db      |
| Username          | backend |
| Password          | redhat  |

Click **Save**. The application verifies the connection before exiting the form.

- 4.5. Navigate to **Servers > rpi-store > Databases > rpi-store**, and then select **Tools > Query Tool** from the menu. In the **Query Editor**, enter the following query.

```
select * from inventory
```

- 4.6. Press F5 to execute the query and retrieve data from the **inventory** table.

The screenshot shows the pgAdmin interface connected to the 'rpi-store/backend' database. The left sidebar shows the server structure under 'rpi-store'. The main area displays the results of the query 'select \* from inventory'. The data output is as follows:

|   | <b>id</b><br>[PK] integer | <b>model_id</b><br>integer | <b>quantity</b><br>integer |
|---|---------------------------|----------------------------|----------------------------|
| 1 | 1                         | 1                          | 0                          |
| 2 | 2                         | 2                          | 20                         |
| 3 | 3                         | 3                          | 300                        |
| 4 | 4                         | 4                          | 40                         |

- 4.7. Modify the data in the `inventory` table. Double-click the 20 value in the `quantity` column. Enter 10 as the value, press `Enter`, and then press `F6` to save the changes.

The screenshot shows the pgAdmin interface connected to the 'rpi-store/backend' database. The left sidebar shows the server structure under 'rpi-store'. The main area displays the results of the query 'select \* from inventory'. The data output is as follows:

|   | <b>id</b><br>[PK] integer | <b>model_id</b><br>integer | <b>quantity</b><br>integer |
|---|---------------------------|----------------------------|----------------------------|
| 1 | 1                         | 1                          | 0                          |
| 2 | 2                         | 2                          | 10                         |
| 3 | 3                         | 3                          | 300                        |
| 4 | 4                         | 4                          | 40                         |
| 5 | 5                         | 5                          | 440                        |

- ▶ 5. From your terminal, stop the development environment.

```
[student@workstation compose-environments]$ podman-compose down
['podman', '--version', '']
using podman version: ...
** excluding: set()
podman stop -t 10 compose_environments_postgresql
compose_environments_postgresql
exit code: 0
podman stop -t 10 compose_environments_pgadmin
compose_environments_pgadmin
exit code: 0
podman rm compose_environments_postgresql
738f...0506
exit code: 0
```

```
podman rm compose_environments_pgadmin
b584...670c
exit code: 0
```

## Finish

On the **workstation** machine, change to the **student** user home directory and use the **lab** command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish compose-environments
```

## ▶ Lab

# Multi-container Applications with Compose

Create a compose file to deploy your application in a testing environment. The application uses three components: a UI, a back end, and an external service that the back end uses. Because the development team does not own the external service, they decided to use a mock server called `Wiremock` to mock the external service interactions with the back end.

## Outcomes

You should be able to:

- Create a multi-container compose file with the following features:
  - Bind mounts
  - Environment variables
  - Networks
  - Published ports
- Reload the compose file after modifying it.

## Before You Begin

As the student user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

```
[student@workstation ~]$ lab start compose-lab
```

The `start` function copies a file called `compose.yaml` that you must complete throughout this exercise. It also copies the configuration for `wiremock` to mock the `quotes-provider` service.

The lab script continuously evaluates the objectives of this lab. Keep the script running in a terminal window and complete the objectives of this lab from a new terminal window.

## Instructions

1. Change into the `~/D0188/labs/compose-lab` directory. This directory contains the compose file called `compose.yaml`, which you must complete. Then, start the containers in the background.

| Container Name  | Image                                                                               |
|-----------------|-------------------------------------------------------------------------------------|
| quotes-provider | <code>registry.ocp4.example.com:8443/redhattraining/wiremock</code>                 |
| quotes-api      | <code>registry.ocp4.example.com:8443/redhattraining/podman-quotesapi-compose</code> |
| quotes-ui       | <code>registry.ocp4.example.com:8443/redhattraining/podman-quotes-ui</code>         |

**Note**

The WireMock container image provides the ability to create a fake API service. This exercise uses this image to run the quotes-provider API. In a real-world scenario, your API image is built from your application source code.

2. Configure the quotes-provider mock service by providing the Wiremock mappings and responses in the ~/D0188/labs/compose-lab/wiremock/stubs directory.  
Wiremock expects two directories to configure the mock server:
  - **mappings**: contains files that define the HTTP endpoints.
  - **\_files**: contains files with fixed responses.
 To provide the Wiremock configuration, mount the ~/D0188/labs/compose-lab/wiremock/stubs directory as the /home/wiremock directory in the container.  
To apply the changes, delete and restart the containers by using the podman-compose command.
3. Expose the quotes-api service to the host machine on port 8080. The quotes-api service listens on port 8080 in the container.  
To apply the changes, delete and restart the containers by using the podman-compose command.
4. Isolate together the quotes-provider and quotes-api services by creating a Podman network called lab-net in the compose.yaml file.  
To apply the changes, delete and restart the containers by using the podman-compose command.
5. The quotes-api and quotes-provider services share a network, but the quotes-api service is missing the hostname configuration of the provider endpoint. Set the QUOTES\_SERVICE environment variable to configure the quotes-provider URL. Use the default name resolution, the quotes-provider configuration, the http protocol, and the port 8080 to configure this value.  
To apply the changes, delete and restart the containers by using the podman-compose command.
6. Expose the quotes-ui service on port 3000 so that the host can access the application on <http://localhost:3000>. The quotes-ui is listening on port 8080 in the UI container.  
To apply the changes, delete and restart the containers by using the podman-compose command.  
Use the previous URL to visit the quotes application by using a web browser.

**Finish**

As the student user on the workstation machine, use the lab command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

Press y when the lab start command prompts you to execute the finish function. Alternatively, execute the following command:

```
[student@workstation ~]$ lab finish compose-lab
```



## ► Solution

# Multi-container Applications with Compose

Create a compose file to deploy your application in a testing environment. The application uses three components: a UI, a back end, and an external service that the back end uses. Because the development team does not own the external service, they decided to use a mock server called `Wiremock` to mock the external service interactions with the back end.

## Outcomes

You should be able to:

- Create a multi-container compose file with the following features:
  - Bind mounts
  - Environment variables
  - Networks
  - Published ports
- Reload the compose file after modifying it.

## Before You Begin

As the student user on the `workstation` machine, use the `lab` command to prepare your system for this exercise.

```
[student@workstation ~]$ lab start compose-lab
```

The `start` function copies a file called `compose.yaml` that you must complete throughout this exercise. It also copies the configuration for `wiremock` to mock the `quotes-provider` service.

The lab script continuously evaluates the objectives of this lab. Keep the script running in a terminal window and complete the objectives of this lab from a new terminal window.

## Instructions

1. Change into the `~/D0188/labs/compose-lab` directory. This directory contains the compose file called `compose.yaml`, which you must complete. Then, start the containers in the background.

| Container Name  | Image                                                                               |
|-----------------|-------------------------------------------------------------------------------------|
| quotes-provider | <code>registry.ocp4.example.com:8443/redhattraining/wiremock</code>                 |
| quotes-api      | <code>registry.ocp4.example.com:8443/redhattraining/podman-quotesapi-compose</code> |
| quotes-ui       | <code>registry.ocp4.example.com:8443/redhattraining/podman-quotes-ui</code>         |

**Note**

The WireMock container image provides the ability to create a fake API service. This exercise uses this image to run the quotes-provider API. In a real-world scenario, your API image is built from your application source code.

- 1.1. Change into the ~/D0188/labs/compose-lab directory.

```
[student@workstation compose-lab]$ cd ~/D0188/labs/compose-lab
no output expected
```

- 1.2. Update the `compose.yaml` file to look like the following code block:

```
services:
 wiremock:
 container_name: "quotes-provider"
 image: "registry.ocp4.example.com:8443/redhattraining/wiremock"
 quotes-api:
 container_name: "quotes-api"
 image: "registry.ocp4.example.com:8443/redhattraining/podman-quotesapi-
compose"
 quotes-ui:
 container_name: "quotes-ui"
 image: "registry.ocp4.example.com:8443/redhattraining/podman-quotes-ui"
```

- 1.3. From the directory that contains the `compose.yaml` file, run `podman-compose up` to create the containers. Use the `-d` option to run the containers in the background.

```
[student@workstation compose-lab]$ podman-compose up -d
['podman', '--version', '']
using podman version: ...
** excluding: set()
...output omitted...
```

2. Configure the quotes-provider mock service by providing the Wiremock mappings and responses in the ~/D0188/labs/compose-lab/wiremock/stubs directory.

Wiremock expects two directories to configure the mock server:

- `mappings`: contains files that define the HTTP endpoints.
- `_files`: contains files with fixed responses.

To provide the Wiremock configuration, mount the ~/D0188/labs/compose-lab/wiremock/stubs directory as the /home/wiremock directory in the container.

To apply the changes, delete and restart the containers by using the `podman-compose` command.

- 2.1. Use a bind mount to map ~/D0188/labs/compose-lab/wiremock/stubs to the /home/wiremock directory. Provide the Z option to add the correct SELinux permissions.

```

services:
 wiremock:
 container_name: "quotes-provider"
 image: "registry.ocp4.example.com:8443/redhattraining/wiremock"
 volumes:
 - ~/D0188/labs/compose-lab/wiremock/stubs:/home/wiremock:Z
 quotes-api:
 container_name: "quotes-api"
 image: "registry.ocp4.example.com:8443/redhattraining/podman-quotesapi-compose"
 quotes-ui:
 container_name: "quotes-ui"
 image: "registry.ocp4.example.com:8443/redhattraining/podman-quotes-ui"

```

- 2.2. Apply the changes by running the `podman-compose down` command followed by `podman-compose up -d`.

Run `podman-compose down` to stop and remove the containers in the `compose.yaml` file.

```

[student@workstation compose-lab]$ podman-compose down
['podman', '--version', '']
using podman version: ...
** excluding: set()
podman stop -t 10 quotes-ui
quotes-ui
exit code: 0
podman stop -t 10 quotes-api
quotes-api
exit code: 0
podman stop -t 10 quotes-provider
quotes-provider
exit code: 0
podman rm quotes-ui
ff3d...093e
exit code: 0
podman rm quotes-api
2a5d...5be3
exit code: 0
podman rm quotes-provider
48c9...c2f8
exit code: 0

```

Run `podman-compose up` to re-create the containers with the new changes.

```

[student@workstation compose-lab]$ podman-compose up -d
['podman', '--version', '']
using podman version: ...
** excluding: set()
...output omitted...

```

3. Expose the `quotes-api` service to the host machine on port 8080. The `quotes-api` service listens on port 8080 in the container.

To apply the changes, delete and restart the containers by using the `podman-compose` command.

- 3.1. Use the `ports` property to expose the port 8080 in the container to port 8080 on the host.

```
services:
 wiremock:
 container_name: "quotes-provider"
 image: "registry.ocp4.example.com:8443/redhattraining/wiremock"
 volumes:
 - ~/D0188/labs/compose-lab/wiremock/stubs:/home/wiremock:Z
 quotes-api:
 container_name: "quotes-api"
 image: "registry.ocp4.example.com:8443/redhattraining/podman-quotesapi-
compose"
 ports:
 - "8080:8080"
 quotes-ui:
 container_name: "quotes-ui"
 image: "registry.ocp4.example.com:8443/redhattraining/podman-quotes-ui"
```

- 3.2. Apply the changes by running the `podman-compose down` command followed by `podman-compose up -d`.

Run `podman-compose down` to stop and remove the containers in the `compose.yaml` file.

```
[student@workstation compose-lab]$ podman-compose down
...output omitted...
```

Run `podman-compose up` to re-create the containers with the new changes.

```
[student@workstation compose-lab]$ podman-compose up -d
...output omitted...
```

4. Isolate together the `quotes-provider` and `quotes-api` services by creating a Podman network called `lab-net` in the `compose.yaml` file.

To apply the changes, delete and restart the containers by using the `podman-compose` command.

- 4.1. Use the `nets` top-level property to create the `lab-net` network from the `compose.yaml` file. Then use the `nets` property under the `wiremock` and `quotes-api` services to connect both containers to the `lab-net` network.

```
services:
 wiremock:
 container_name: "quotes-provider"
 image: "registry.ocp4.example.com:8443/redhattraining/wiremock"
 volumes:
 - ~/D0188/labs/compose-lab/wiremock/stubs:/home/wiremock:Z
 networks:
 - lab-net
 quotes-api:
```

```
container_name: "quotes-api"
image: "registry.ocp4.example.com:8443/redhattraining/podman-quotesapi-
compose"
ports:
- "8080:8080"
networks:
- lab-net
quotes-ui:
container_name: "quotes-ui"
image: "registry.ocp4.example.com:8443/redhattraining/podman-quotes-ui"

networks:
lab-net: {}
```

- 4.2. Apply the changes by running the `podman-compose down` command followed by the `podman-compose up -d` command.

Run the `podman-compose down` command to stop and remove the containers in the `compose.yaml` file.

```
[student@workstation compose-lab]$ podman-compose down
...output omitted...
```

Run the `podman-compose up` command to re-create the containers with the new changes.

```
[student@workstation compose-lab]$ podman-compose up -d
...output omitted...
```

5. The `quotes-api` and `quotes-provider` services share a network, but the `quotes-api` service is missing the hostname configuration of the provider endpoint. Set the `QUOTES_SERVICE` environment variable to configure the `quotes-provider` URL. Use the default name resolution, the `quotes-provider` configuration, the `http` protocol, and the port `8080` to configure this value.

To apply the changes, delete and restart the containers by using the `podman-compose` command.

- 5.1. Add the `QUOTES_SERVICE="http://quotes-provider:8080"` environment variable to the `compose.yaml` file.

```
services:
wiremock:
container_name: "quotes-provider"
image: "registry.ocp4.example.com:8443/redhattraining/wiremock"
volumes:
- ~/D0188/labs/compose-lab/wiremock/stubs:/home/wiremock:Z
networks:
- lab-net
quotes-api:
container_name: "quotes-api"
image: "registry.ocp4.example.com:8443/redhattraining/podman-quotesapi-
compose"
ports:
- "8080:8080"
```

```

networks:
 - lab-net
environment:
 QUOTES_SERVICE: "http://quotes-provider:8080"
quotes-ui:
 container_name: "quotes-ui"
 image: "registry.ocp4.example.com:8443/redhattraining/podman-quotes-ui"

networks:
 lab-net: {}

```

- 5.2. Apply the changes by running the `podman-compose down` command followed by the `podman-compose up -d` command.

Run the `podman-compose down` command to stop and remove the containers in the `compose.yaml` file.

```
[student@workstation compose-lab]$ podman-compose down
...output omitted...
```

Run the `podman-compose up` command to re-create the containers with the new changes.

```
[student@workstation compose-lab]$ podman-compose up -d
...output omitted...
```

6. Expose the `quotes-ui` service on port 3000 so that the host can access the application on `http://localhost:3000`. The `quotes-ui` is listening on port 8080 in the UI container.

To apply the changes, delete and restart the containers by using the `podman-compose` command.

Use the previous URL to visit the quotes application by using a web browser.

- 6.1. Use the `ports` property in the `quotes-ui` service to expose the port 8080 in the container to port 3000 on the host.

```

services:
 wiremock:
 container_name: "quotes-provider"
 image: "registry.ocp4.example.com:8443/redhattraining/wiremock"
 volumes:
 - ~/D0188/labs/compose-lab/wiremock/stubs:/home/wiremock:Z
 networks:
 - lab-net
 quotes-api:
 container_name: "quotes-api"
 image: "registry.ocp4.example.com:8443/redhattraining/podman-quotesapi-
compose"
 ports:
 - "8080:8080"
 networks:
 - lab-net
 environment:
 QUOTES_SERVICE: "http://quotes-provider:8080"
 quotes-ui:

```

```
container_name: "quotes-ui"
image: "registry.ocp4.example.com:8443/redhattraining/podman-quotes-ui"
ports:
 - "3000:8080"

networks:
 lab-net: {}
```

- 6.2. Apply the changes by running the `podman-compose down` command followed by the `podman-compose up -d` command.

Run the `podman-compose down` command to stop and remove the containers in the `compose.yaml` file.

```
[student@workstation compose-lab]$ podman-compose down
...output omitted...
```

Run the `podman-compose up` command to re-create the containers with the new changes.

```
[student@workstation compose-lab]$ podman-compose up -d
...output omitted...
```

- 6.3. Navigate to `http://localhost:3000` to verify that the quotes application shows famous quotes in the browser.

## Finish

As the `student` user on the `workstation` machine, use the `lab` command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

Press `y` when the `lab start` command prompts you to execute the `finish` function. Alternatively, execute the following command:

```
[student@workstation ~]$ lab finish compose-lab
```

# Summary

---

- Podman Compose is an open source tool to run Compose files.
- Podman Compose removes the need to run the Podman service that provides the API socket, to save resources and to provide a more native and lightweight solution.
- Podman Compose translates Compose files into Podman CLI commands.
- The `podman generate kube` command generates a Kubernetes YAML file from existing Podman containers, pods, and volumes.
- The `podman play kube` command reads the Kubernetes YAML file, and then recreates the defined resources in a Kubernetes or an RHOCP cluster.
- You can use Podman Compose to quickly build a stateful, multi-container environment for your applications.
- You can restrict communication between specific containers by defining additional networks for the pod and then defining the network settings.
- You can use Podman Compose to approximate a production environment on your local machine by providing applications, databases, or cache systems configuration in a single file.
- You can use Podman Compose to execute your application and database containers in isolated local environments.
- Using Podman Compose in a production environment is not recommended, because it does not support advanced features that you might need in a production environment, such as load balancing and distributing containers to multiple host machines, or nodes. Instead, use Kubernetes or RHOCP for a production container orchestration solution.

## Chapter 8

# Container Orchestration with OpenShift and Kubernetes

### Goal

Orchestrate containerized applications with Kubernetes and OpenShift.

### Objectives

- Deploy an application in OpenShift.
- Deploy a multi-pod application to OpenShift and make it externally available.

### Sections

- Deploy Applications in OpenShift (and Guided Exercise)
- Multi-pod Applications (and Guided Exercise)

### Lab

- Container Orchestration with Kubernetes and OpenShift

# Deploy Applications in OpenShift

---

## Objectives

- Deploy an application in OpenShift.

## Kubernetes and Red Hat OpenShift

Kubernetes is an orchestration platform that simplifies the deployment, management, and scaling of containerized applications. Kubernetes uses several servers, called *nodes*, to ensure the resiliency and scalability of the deployed applications. Kubernetes forms a cluster of servers that run containers and are centrally managed by a set of control plane servers. A server can act as both a control plane node and a compute node, but those roles are usually separated for increased stability, security, and manageability.

Red Hat OpenShift is a set of modular components and services that use and expand Kubernetes. Red Hat OpenShift adds capabilities such as remote management, increased security, monitoring and auditing, application lifecycle management, and self-service interfaces for developers.

Based on your infrastructure needs, you can choose from various Red Hat OpenShift editions. Red Hat OpenShift Container Platform (RHOCP) is one of the self-managed options, which you can install and configure on your own infrastructure. RHOCP supports many different target platforms, such as Amazon Web Services (AWS), Microsoft Azure, and bare metal.

Red Hat OpenShift is also available as a cloud managed service. A managed platform can reduce your operational effort and increase your productivity, because you do not need to maintain the infrastructure layer. Examples of Red Hat OpenShift managed services are Microsoft Azure Red Hat OpenShift (ARO) and Red Hat OpenShift Service on AWS (ROSA), among others.



### Note

The remainder of this chapter covers RHOCP, but the same concepts apply to the managed Red Hat OpenShift editions.

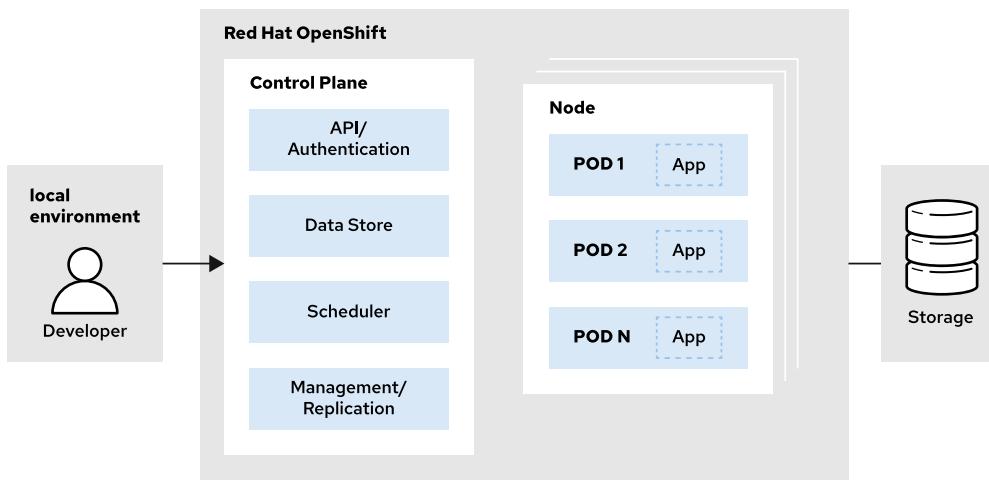


Figure 8.1: RHOCP architecture

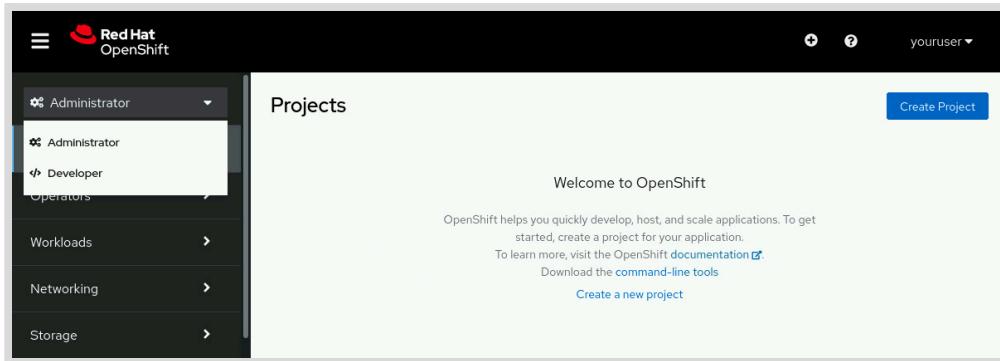
Developers deploy application containers grouped in *pods*. Pods are Kubernetes objects that represent the smallest deployment unit managed by RHOCP.

Developers group related pods and other RHOCP objects in *projects*. A project is an RHOCP resource which is similar to the *namespace* Kubernetes resource. Grouping RHOCP objects in projects is useful, for example, for configuring limits on resource usage for teams or applications.

## The RHOCP Web Console

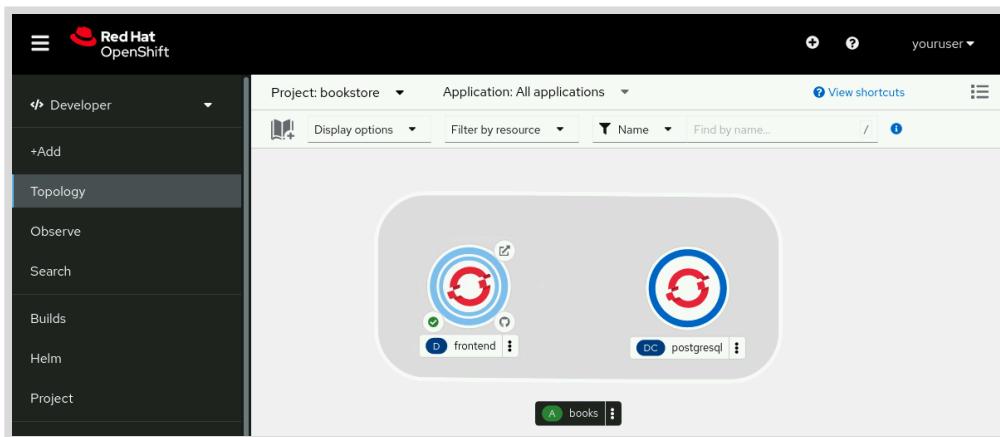
The RHOCP web console is a browser-based user interface that you can use to interact with RHOCP. You can deploy and manage your applications by using the web console.

The web console provides the *Administrator* and *Developer* perspectives. The Developer perspective focuses on the status and management of the deployed applications. The Administrator perspective focuses on the status and management of RHOCP and its resources.



**Figure 8.2: The Administrator and Developer perspectives**

The *Topology* view in the Developer perspective provides a visual representation of deployed applications.



**Figure 8.3: The Topology view**

## The RHOCP CLI

Command-line utilities (CLI) provide an alternative for interacting with your RHOCP cluster. Developers familiar with Kubernetes can use the `kubectl` utility to manage an RHOCP cluster. This course uses the `oc` command-line utility, which is designed to take advantage of additional RHOCP features.

The CLI utilities provide developers with a range of commands that are useful for managing the RHOCP cluster and its applications. Each command is translated into an API call, and the response is displayed in the command line.

The following is a list of the most common oc commands.

### **oc login**

Before you can interact with your RHOCP cluster, you must authenticate your requests. Use the `login` command to authenticate your requests.

For example, in this course, you can use the following command:

```
[user@host ~]$ oc login https://api.ocp4.example.com:6443
Username: developer
Password: developer
Login successful.
```

You don't have any projects. You can try to create a new project, by running

```
$ oc new-project <projectname>
```

```
Welcome to OpenShift! See 'oc help' to get started.
```

### **oc get**

Use the `get` command to retrieve a list of selected resources in the selected project.

You must specify the resource type to list.

For example, the following command returns the list of the pod resources in the current project.

```
[user@host ~]$ oc get pod
NAME READY STATUS RESTARTS AGE
quotes-api-6c9f758574-nk8kd 1/1 Running 0 39m
quotes-ui-d7d457674-rbk17 1/1 Running 0 67s
```

### **oc create**

Use the `create` command to create an RHOCP resource. Developers commonly use the `-f` flag to indicate the file that contains the JSON or YAML representation of an RHOCP resource.

For example, to create resources from the `pod.yaml` file, use the following command:

```
[user@host ~]$ oc create -f pod.yaml
pod/quotes-pod created
```

RHOCP resources in YAML format are discussed later.

### **oc delete**

Use the `delete` command to delete an existing RHOCP resource. You must specify the resource type and the resource name.

For example, to delete the `quotes-ui` pod, use the following command:

```
[user@host ~]$ oc delete pod quotes-ui
pod/quotes-ui deleted
```

### oc logs

Use the `logs` command to print the standard output of a pod. This command requires a pod name as an argument. You can print only logs of a container in a pod, which means the resource type is omitted.

For example, to print the logs from the `react-ui` pod, use the following command:

```
[user@host ~]$ oc logs react-ui
Compiled successfully!

You can now view ts-page in the browser.

Local: http://localhost:3000
On Your Network: http://10.0.1.23:3000
...output omitted...
```

The `oc` utility provides equal functionality to the web console. For a full list of commands, execute `oc --help`. Additionally, you can execute `oc command --help`, for example `oc logs --help` to view the `oc logs` command documentation.

## RHOCP Resources

Developers configure RHOCP by using a set of Kubernetes and RHOCP-specific objects. When you create or modify an object, you make a persistent record of the intended state. RHOCP reads the object and modifies the current state accordingly.



### Note

RHOCP objects generally use the YAML definition format because the YAML format is easy to read and understand.

If you are not familiar with the YAML format, see the references section for more information.

## Shared Resource Fields

All RHOCP and Kubernetes objects can be represented as a JSON or YAML structure with common fields. Consider the following pod object in the YAML format:

```
kind: Pod 1
apiVersion: v1 2
metadata: 3
 name: example-pod
 namespace: example-namespace
spec: 4
 ...definition omitted...
status: 5
 conditions:
 - lastProbeTime: null
 lastTransitionTime: "2022-08-19T12:59:22Z"
```

```

status: "True"
type: PodScheduled
containerStatuses:
- containerID: cri-o://e37c...f5c2
 image: quay.io/example/awesome-container:latest
 lastState: {}
 name: podman-quotes-ui
 ready: true
...object omitted...

```

- ➊ Schema identifier. In this example, the object conforms to the pod schema.
- ➋ Identifier of the object schema version.
- ➌ Metadata for a given resource, such as annotations, labels, name, namespace, and others. The `.metadata.namespace` field refers to an RHOCP project.
- ➍ The desired state of the object.
- ➎ Current state of the object. This field is provided by RHOCP, and lists information such as runtime status, readiness, container image, and others.

Every RHOCP resource contains the `.kind`, `.apiVersion`, `.spec` and `.status` fields. However, when you create an object definition, you do not need to provide the `.status` field. The `.status` field is useful, for example, for troubleshooting the reason for a pod scheduling error.

Use the `oc explain` command to get information about valid fields for an object. For example, execute `oc explain pod` to get information about possible Pod object fields. You can use the YAML path to get information about a particular field, for example:

```

[user@host ~]$ oc explain pod.metadata.name
KIND: Pod
VERSION: v1

FIELD: name <string>

DESCRIPTION:
...output omitted...

```

## Label Kubernetes Objects

Labels are key-value pairs that you define in the `.metadata.labels` object, for example:

```

kind: Pod
apiVersion: v1
metadata:
 name: example-pod
 labels:
 app: example-pod
 group: developers
...object omitted...

```

The preceding example contains the `app=example-pod` and `group=developers` labels. Developers often use labels to target a set of objects by using the `-l` or the equivalent `--`

`selector` option. For example, the following `oc get` command lists pods that contain the `group=developers` label:

```
[user@host ~]$ oc get pod --selector group=developers
NAME READY STATUS RESTARTS AGE
example-pod-6c9f758574-7fhg 1/1 Running 5 11d
```

## Deploy Applications as Pods to RHOCP

A pod represents a group of one or multiple containers that share resources, such as a network interface or file system.

Grouping containers into pods is useful for implementing multi-container design patterns, such as using an initialization container to load application data when a pod starts for the first time. Multi-container design patterns are out of scope for this course.

You can use pods locally or on a cluster. On a local system, Podman can manage pods by using the `podman pod` command. This is useful for developing and testing applications that use a multi-container design pattern. On Kubernetes or RHOCP, pods are the atomic unit for deploying and managing your applications. For example, you can deploy a single-container application by creating a pod that contains a single container.

Pods do not expose advanced functionality, such as application scaling or zero-downtime updates. Kubernetes and RHOCP implement such functionality with objects that control pod sets, such as the `Deployment` object. Controller objects are covered in a later section.

## Create Pods Declaratively

Storing a pod definition in a file is called the *declarative approach* for creating RHOCP objects, because you declare the intended pod state outside of RHOCP. This means you can store the pod definition in a version control system, such as Git, and incrementally change your definition as your application evolves.

The following YAML object demonstrates the important fields of a pod object:

```
kind: Pod 1
apiVersion: v1
metadata:
 name: example-pod 2
 namespace: example-project 3
spec: 4
 containers: 5
 - name: example-container 6
 image: quay.io/example/awesome-container 7
 ports: 8
 - containerPort: 8080
 env: 9
 - name: GREETING
 value: "Hello from the awesome container"
```

- 1** This YAML object defines a pod.
- 2** The `.metadata.name` field defines the name of the pod.

- ③ The project in which you create the pod. If this project does not exist, then the pod creation fails. If you do not specify a project, then RHOCP uses the currently configured project.
- ④ The `.spec` field contains the pod object configuration.
- ⑤ The `.spec.containers` field defines containers in a pod. In this example, the `example-pod` contains one container.
- ⑥ The name of the container inside of a pod. Container names are important for `oc` commands when a pod contains multiple containers.
- ⑦ The image for the container.
- ⑧ The port metadata specifies what ports the container uses. This property is similar to the `EXPOSE` Containerfile property.
- ⑨ The `env` property defines environment variables.

You can create the pod object by saving the YAML definition into a file and then using the `oc` command, for example:

```
[user@host ~]$ oc create -f pod.yaml
pod/example-pod created
```

## Create Pods Imperatively

RHOCP also provides the *imperative approach* to create RHOCP objects. The imperative approach uses the `oc run` command to create a pod without a definition.

The following command creates the `example-pod` pod:

```
[user@host ~]$ oc run example-pod \
--image=quay.io/example/awesome-container \
--env GREETING='Hello from the awesome container' \
--port=8080
pod/example-pod created
```

- ① The pod `.metadata.name` definition.
- ② The image used for the single container in this pod.
- ③ The `--env` option injects an environment variable in the container of this pod.
- ④ The port metadata definition.

The imperative commands are a faster way of creating pods, because such commands do not require a pod object definition. However, developers lose the ability to version and incrementally change the pod definition.

Generally, developers test the deployment by using imperative commands, and then use the imperative commands to generate the pod object definition. Use the `--dry-run=client` option to avoid creating the object in RHOCP. Additionally, use the `-o yaml` or `-o json` option to configure the definition format.

The following command is an example of generating the YAML definition for the `example-pod` pod.

```
[user@host ~]$ oc run example-pod \
--image=quay.io/example/awesome-container \
--env GREETING='Hello from the awesome container' \
--port=8080 \
--dry-run=client -o yaml
apiVersion: v1
kind: Pod
metadata:
 creationTimestamp: null
 labels:
 run: example-pod
 name: example-pod
spec:
 containers:
 ...output omitted...
```

## Application Networking in RHOCP

Developers configure internal pod-to-pod network communication in RHOCP by using the Service object. Applications send requests to the service name and port. RHOCP provides a virtual network which reroutes such requests to the pods that the service targets by using labels.

### Create Services Declaratively

The following YAML object demonstrates a service object:

```
apiVersion: v1
kind: Service
metadata:
 name: backend
spec:
 ports:
 - port: 8080 ①
 protocol: TCP
 targetPort: 8080 ②
 selector: ③
 app: backend-app
```

- ①** Service port. This is the port on which the service listens.
- ②** Target port. This is the pod port to which the service routes requests. This port corresponds to the `containerPort` value in the pod definition.
- ③** The selector configures which pods to target. In this case, the service routes to any pods that contain the `app=backend-app` label.

The preceding example defines the `backend` service. This means that applications can send requests to the `http://backend:8080` URL. Such requests are routed to the pods with the `app=backend-app` label on port 8080.

In RHOCP, the default service type is `ClusterIP`, which means the service is used for pod-to-pod routing within the RHOCP cluster. Other service types, such as the `LoadBalancer` service, are outside of the scope of this course.

## Create Services Imperatively

Similarly to pods, you can create services imperatively by using the `oc expose` command. The following example creates a service for the `backend-app` pod:

```
[user@host ~]$ oc expose pod backend-app \
--port=8080 \
--targetPort=8080 \
--name=backend-app
service/backend-app exposed
```

- ❶ Port on which the service listens.
- ❷ Target container port.
- ❸ Service name.

You can also use the `--dry-run=client` and `-o` options to generate a service definition, for example:

```
[user@host ~]$ oc expose pod backend-app \
--port=8080 \
--targetPort=8080 \
--name=backend-app \
--dry-run=client -o yaml
apiVersion: v1
kind: Service
metadata:
 name: backend-app
spec:
...output omitted...
```



## References

For more information about the YAML format, refer to the *YAML in a Nutshell* chapter in the Red Hat Enterprise Linux Atomic Host 7 *Getting Started with Kubernetes* documentation at

[https://access.redhat.com/documentation/en-us/red\\_hat\\_enterprise\\_linux\\_atomic\\_host/7/html-single/getting\\_started\\_with\\_kubernetes/index#yaml\\_in\\_a\\_nutshell](https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_atomic_host/7/html-single/getting_started_with_kubernetes/index#yaml_in_a_nutshell)

For more information about the architecture, refer to the *Architecture overview* chapter in the Red Hat OpenShift Container Platform 4.14 *Architecture* documentation at

[https://access.redhat.com/documentation/en-us/openshift\\_container\\_platform/4.14/html-single/architecture/index#architecture-overview](https://access.redhat.com/documentation/en-us/openshift_container_platform/4.14/html-single/architecture/index#architecture-overview)

For more information about the web console, refer to the Red Hat OpenShift Container Platform 4.14 *Web console* documentation at

[https://access.redhat.com/documentation/en-us/openshift\\_container\\_platform/4.14/html-single/web\\_console/index](https://access.redhat.com/documentation/en-us/openshift_container_platform/4.14/html-single/web_console/index)

For more information about the oc CLI, refer to the *OpenShift CLI* chapter in the Red Hat OpenShift Container Platform 4.14 *CLI tools* documentation at

[https://access.redhat.com/documentation/en-us/openshift\\_container\\_platform/4.14/html-single/cli\\_tools/index#openshift-cli-oc](https://access.redhat.com/documentation/en-us/openshift_container_platform/4.14/html-single/cli_tools/index#openshift-cli-oc)

For more information about pods, refer to the *Using Pods* section in the *Overview of Nodes* chapter in the Red Hat Openshift Container Platform 4.14 *Nodes* documentation at

[https://access.redhat.com/documentation/en-us/openshift\\_container\\_platform/4.14/html-single/nodes/index#working-with-pods](https://access.redhat.com/documentation/en-us/openshift_container_platform/4.14/html-single/nodes/index#working-with-pods)

For more information about Red Hat OpenShift Service on AWS, refer to the Red Hat OpenShift Service on AWS 4 *Introduction to ROSA* documentation at

[https://access.redhat.com/documentation/en-us/red\\_hat\\_openshift\\_service\\_on\\_aws/4/html/introduction\\_to\\_rosa/index](https://access.redhat.com/documentation/en-us/red_hat_openshift_service_on_aws/4/html/introduction_to_rosa/index)

### Microsoft Azure Red Hat OpenShift

<https://www.redhat.com/en/technologies/cloud-computing/openshift/azure>

### Kubernetes API Conventions

<https://github.com/kubernetes/community/blob/master/contributors/devel/sig-architecture/api-conventions.md>

### Kubernetes - Service

<https://kubernetes.io/docs/concepts/services-networking/service/>

### Service API Reference

[https://docs.openshift.com/container-platform/4.14/rest\\_api/network\\_apis/service-v1.html](https://docs.openshift.com/container-platform/4.14/rest_api/network_apis/service-v1.html)

### Pod API Reference

[https://docs.openshift.com/container-platform/4.14/rest\\_api/workloads\\_apis/pod-v1.html](https://docs.openshift.com/container-platform/4.14/rest_api/workloads_apis/pod-v1.html)

## ► Guided Exercise

# Deploy Applications in OpenShift

Create and modify Red Hat OpenShift Container Platform (RHOCP) objects.

## Outcomes

You should be able to:

- Use the RHOCP Web Console to verify the status of applications.
- Use the RHOCP Web Console to modify the Service objects.
- Use the `oc` command-line interface to verify the status of applications.
- Use the `oc` command-line interface to deploy new applications.

## Before You Begin

As the student user on the workstation machine, use the `lab` command to prepare your system for this exercise.

This command deploys the `podman-hello` application into the `ocp-applications` project.

```
[student@workstation ~]$ lab start openshift-applications
```

## Instructions

- 1. Log in to the cluster as the `developer` user, and ensure that you use the `ocp-applications` project.

- 1.1. Log in to the cluster as the `developer` user.

```
[student@workstation ~]$ oc login -u developer -p developer \
https://api.ocp4.example.com:6443
Login successful.

...output omitted...
```

- 1.2. Ensure that you use the `ocp-applications` project.

```
[student@workstation ~]$ oc project ocp-applications
Already on project "ocp-applications" on server "https://
api.ocp4.example.com:6443".
```

- 2. Explore the `podman-hello-client` application.

- 2.1. Open the `~/D0188/labs/openshift-applications/podman-hello-client/Containerfile` file in a text editor, such as `gedit`.

Note the default environment variable values and the `CMD` instruction:

```

FROM registry.access.redhat.com/ubi8/ubi-minimal:8.6

ARG PROTO="http" \
 URL="hello-server-svc" \
 PORT="3000" \
 ENDPOINT="greet"

ENV PROTO=${PROTO} \
 URL=${URL} \
 PORT=${PORT} \
 ENDPOINT=${ENDPOINT}

...Containerfile omitted...

CMD ["./client.sh"]

```

- 2.2. Open the ~/D0188/labs/openshift-applications/podman-hello-client/client.sh file in a text editor, such as gedit.

Note that the client uses the environment variables in the following format:

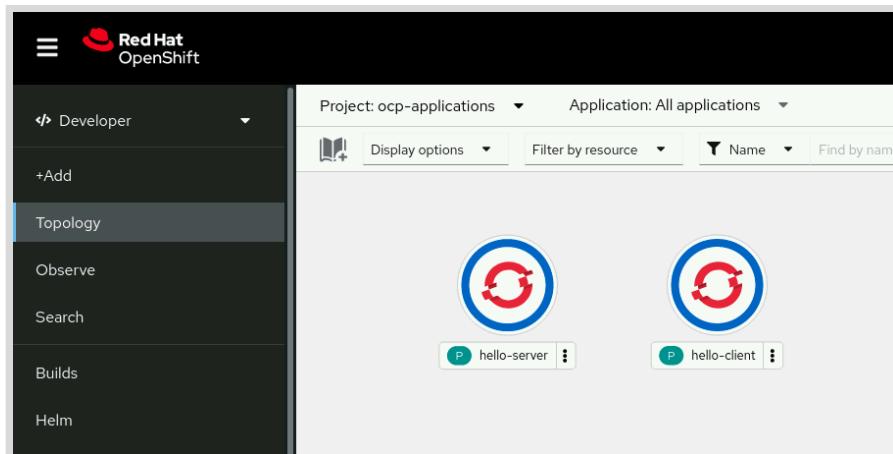
```

...script omitted...
curl ${OPTS} "${PROTO}://${URL}:${PORT}/${ENDPOINT}"
...script omitted...

```

This means that the client sends requests to the `http://hello-server-svc:3000/greet` URL by default.

- ▶ 3. Explore the deployed application in the ocp-applications project by using the RHOCP Web Console.
  - 3.1. In a web browser, navigate to `https://console-openshift-console.apps.ocp4.example.com`. Select the `htpasswd_provider` user provider, and log in with the following credentials:
    - **Username:** developer
    - **Password:** developer
 Click **Skip tour** if prompted.
  - 3.2. Click **ocp-applications**, then click the overflow menu and select **Topology**. Note the **hello-server** and **hello-client** application pods.



- 3.3. Click the **hello-server** pod. Notice the **hello-server-svc** service.

The service serves on port 3000 and routes requests to the port 3000 inside the **hello-server** pod.

- 3.4. Click the **hello-client** pod. Then, view the application logs by clicking **View logs**.

The **hello-client** application sends requests to the `http://hello-server-svc:3000/greet` URL and prints the `{"hello": "world"}` response to the console.

- 3.5. Return to the **Topology** view by clicking the **Red Hat OpenShift** logo.

► 4. Modify the **hello-server-svc** service to serve on the 8080 port.

- 4.1. Click the **hello-server** pod. Then, click the **hello-server-svc** service.

- 4.2. Click the **YAML** tab. In the text editor, scroll to the `spec.ports` object and change the `port` property to the 8080 value:

```
...YAML omitted...
spec:
 ports:
 - protocol: TCP
 port: 8080
 targetPort: 3000
...YAML omitted...
```

Click **Save**.

- 4.3. Verify the service port mapping.

Click the **Details** tab and verify that the **Port Mapping** section contains the **8080 Port** and the **3000 Pod port or name** values.

► 5. Modify the **hello-client** pod to use the 8080 port.

- 5.1. Return to the terminal window and verify that the `oc` command is logged in with the RHOCP API.

```
[student@workstation ~]$ oc whoami --show-context
ocp-applications/api-ocp4-example-com:6443/developer
```

- 5.2. Display the last line of the `hello-client` pod logs.

```
[student@workstation ~]$ oc logs hello-client | tail -n 1
empty line expected
```

In the preceding instruction, the output of the `oc logs hello-client` command is used as the input for the `tail` command. The `tail -n 1` command displays the last line of the output.

The result is empty because `hello-client` cannot reach the `hello-server` pod on port 3000.

- 5.3. Delete the `hello-client` pod.

```
[student@workstation ~]$ oc delete pod hello-client
pod "hello-client" deleted
```

- 5.4. Recreate the `hello-client` pod. Set the `PORT` environment variable to the `8080` value and use the `registry.ocp4.example.com:8443/redhat/training/podman-hello-client:latest` image.

```
[student@workstation ~]$ oc run hello-client --env PORT=8080 \
--image registry.ocp4.example.com:8443/redhat/training/podman-hello-client:latest
pod "hello-client" created
```

- 5.5. Verify that the `hello-client` pod is in the `Running` status.

```
[student@workstation ~]$ oc get pod
NAME READY STATUS RESTARTS AGE
hello-client 1/1 Running 0 32s
hello-server 1/1 Running 0 83m
```

- 5.6. Display the last line of the `hello-client` pod logs.

```
[student@workstation ~]$ oc logs hello-client | tail -n 1
{"hello":"world"}
```

This means that the `hello-client` pod can reach the `hello-server` pod.

## Finish

On the `workstation` machine, use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish openshift-applications
```

# Multi-pod Applications

## Objectives

- Deploy a multi-pod application to OpenShift and make it externally available.

## Manage Pods with Controllers

Kubernetes encourages developers to use a declarative deployment style, which means that the developer declares the desired application state and Kubernetes reaches the declared state. Kubernetes uses the *controller pattern* to implement the declarative deployment style.

In the controller pattern, a controller continuously watches the current state of a system. If the system state does not match the desired state, then the controller sends signals to the system until the system state matches the desired state. This is called the *control loop*.

Kubernetes uses a number of controllers to deal with constant change. Consider a single-container *bare pod* application, which means an application that is deployed by using the Pod object. When a bare pod application fails, for example due to a memory leak, Kubernetes can handle the failure by restarting the containers according to the pod's `restartPolicy`. However, in case of a node failure, Kubernetes does not reschedule a bare pod, which is why developers rarely use bare pods for application deployment.

You can use a number of controller objects provided by Kubernetes, such as `Deployment`, `ReplicaSet`, `StatefulSet`, and others.

## Deploy Pods with Deployments

The `Deployment` object is a Kubernetes controller that manages pods. Developers use deployments for the following use cases:

### Managing Pod Scaling

You can configure the number of pod replicas for a given deployment. If the actual replica count decreases, for example due to node failure or a network partition, then the deployment schedules new pods until the declared replica count is reached.

### Managing Application Changes

Most of the Pod object fields are immutable, such as the name or environment variable configuration. To change those fields with bare pods, developers must delete the pod and recreate it with the new configuration.

When you manage pods with a deployment, you declare the new configuration in the deployment object. The deployment controller then deletes the original pods and recreates the new pods that contain the new configuration.

### Managing Application Updates

Deployments implement the `RollingUpdate` strategy for gradually updating application pods when you declare a new application image. This ensures zero-downtime application updates.

Additionally, you can *roll back* to the previous application version in case the update does not work correctly.

## Create Deployments Declaratively

The following YAML object demonstrates a Kubernetes deployment:

```
apiVersion: apps/v1
kind: Deployment
metadata: ①
 labels:
 app: deployment-label
 name: example-deployment
spec:
 replicas: 3 ②
 selector: ③
 matchLabels:
 app: example-deployment
 strategy: RollingUpdate ④
 template: ⑤
 metadata:
 labels:
 app: example-deployment
 spec: ⑥
 containers:
 - image: quay.io/example/awesome-container
 name: awesome-pod
```

- ① Deployment object metadata. This deployment uses the `app=deployment-label` label.
- ② Number of pod replicas. This deployment maintains 3 identical containers in 3 pods.
- ③ Selector for pods that the deployment manages. This deployment manages pods that use the `app=example-deployment` label.
- ④ Strategy for updating pods. The default `RollingUpdate` strategy ensures gradual, zero-downtime pod rollout when you modify the container image.
- ⑤ The template configuration for pods that the deployment creates and manages.
- ⑥ The `.spec.template.spec` field corresponds to the `.spec` field of the Pod object.

## Create Deployments Imperatively

You can use the `oc create deployment` command to create a deployment:

```
[user@host ~]$ oc create deployment example-deployment \ ①
 --image=quay.io/example/awesome-container \ ②
 --replicas=3 ③
deployment/example-deployment created
```

- ① Set the name to `example-deployment`.
- ② Set the image.
- ③ Create and maintain 3 replica pods.

You can also use the `--dry-run=client` and `-o` options to generate a deployment definition, for example:

```
[user@host ~]$ oc create deployment example-deployment \
--image=quay.io/example/awesome-container \
--replicas=3 \
--dry-run=client -o yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 creationTimestamp: null
...output omitted...
```

## Deployment Pod Selection

Controllers create and manage the lifecycle of pods that are specified in the controller configuration. Consequently, a controller must have an ownership relationship with pods, and one pod can be owned by at most one controller.

The deployment controller uses labels to target dependent pods. For example, consider the following deployment configuration:

```
apiVersion: apps/v1
kind: Deployment
...configuration omitted...
spec:
 selector:
 matchLabels: ①
 app: example-deployment
 template:
 metadata:
 labels: ②
 app: example-deployment
 spec:
...configuration omitted...
```

- ① The `.spec.selector.matchLabels` field.
- ② The `.spec.template.metadata.labels` field.

The `.spec.template.metadata.labels` field determines the set of labels applied to the pods created or managed by this deployment. Consequently, the `.spec.selector.matchLabels` field must be a subset of labels of the `.spec.template.metadata.labels` field.

A deployment that targets pod labels missing from the pod template is considered invalid.



### Note

The deployment controller targets a ReplicaSet object, which in turn targets pods. Additionally, pod ownership uses object references so that multiple pods can use the same labels and be managed by different controllers.

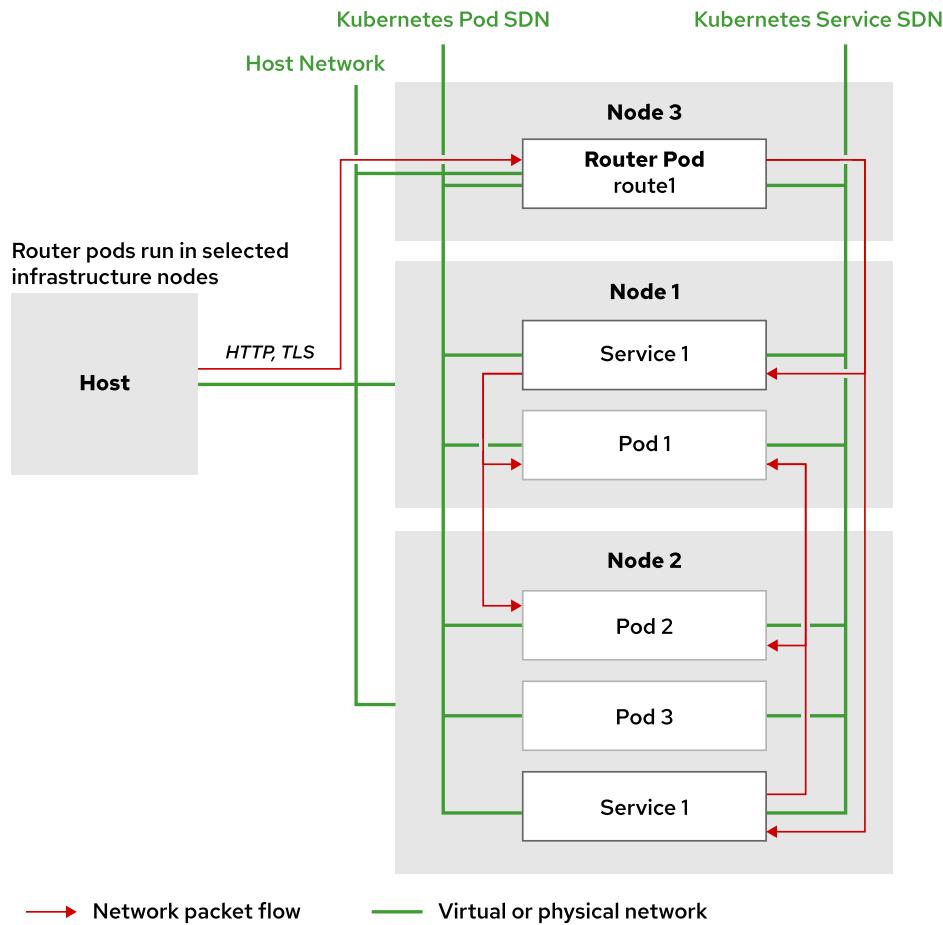
See the references section for complete information about object ownership and the deployment controller.

## Expose Applications for External Access

Developers commonly expose applications for access from outside of the Kubernetes cluster. The default Service object provides a stable, internal IP address and a DNS record for pods. However, applications outside of the Kubernetes cluster cannot connect to the IP address or resolve the DNS record.

You can use the Route object to expose applications. Route is a Red Hat OpenShift Container Platform (RHOCP) object that connects a public-facing IP address and DNS hostname to an internal-facing IP address. Routes use information from the service object to route requests to pods.

RHOCP exposes an HAProxy router pod that listens on a RHOCP node public IP address. The router serves as an ingress entrypoint to the internal RHOCP traffic.



**Figure 8.5: RHOCP network architecture**

Routes are RHOCP-specific objects. If you require full compatibility with other Kubernetes distributions, you can use the Ingress object to create and manage routes, which is out of scope of this course.

## Create Routes Declaratively

The following YAML object demonstrates a RHOCP route:

```

apiVersion: route.openshift.io/v1
kind: Route
metadata:
 labels:
 app: app-ui
 name: app-ui
 namespace: awesome-app
spec:
 port:
 targetPort: 8080 ①
 host: ""
 to: ②
 kind: "Service"
 name: "app-ui"

```

- ① The target port on pods selected by the service this route points to. If you use a string, then the route uses a named port in the target endpoints port list.
- ② The route target. Currently, only the Service target is allowed.

The preceding route routes requests to the `app-ui` service endpoints on port 8080. Because the `app-ui` route does not specify the hostname in the `.spec.host` field, the hostname is generated in the following format:

```
route-name-project-name.default-domain
```

In the preceding example, the hostname in the classroom RHOC cluster is `app-ui-awesome-app.apps.ocp4.example.com`. Consequently, RHOC routes external requests from the `http://app-ui-awesome-app.apps.ocp4.example.com` domain to the `app-ui:8080` internal RHOC service.

Administrators configure the default domain during RHOC deployment.

## Create Routes Imperatively

You can use the `oc expose service` command to create a route:

```
[user@host ~]$ oc expose service app-ui
route.route.openshift.io/app-ui exposed
```

You can also use the `--dry-run=client` and `-o` options to generate a route definition, for example:

```
[user@host ~]$ oc expose service app-ui \
 --dry-run=client -o yaml
apiVersion: apps/v1
kind: Route
metadata:
 creationTimestamp: null
...output omitted...
```

Note that you can use the `oc expose imperative` command in the following forms:

- `oc expose pod POD_NAME`: create a service for a specific pod.
- `oc expose deployment DEPLOYMENT_NAME`: create a service for all pods managed by a controller, in this case a deployment controller.
- `oc expose service SERVICE_NAME`: create a route that targets the specified service.



## References

### Controllers | Kubernetes

<https://kubernetes.io/docs/concepts/architecture/controller/>

### Deployments | Kubernetes

<https://kubernetes.io/docs/concepts/workloads/controllers/deployment/>

### Garbage Collection | Kubernetes

<https://kubernetes.io/docs/concepts/architecture/garbage-collection/#owners-dependents>

### Workloads | Kubernetes

<https://kubernetes.io/docs/concepts/workloads/>

For more information about RHOCN networking, refer to the Red Hat OpenShift Container Platform 4.14 Networking documentation at

[https://access.redhat.com/documentation/en-us/openshift\\_container\\_platform/4.14/html-single/networking/index](https://access.redhat.com/documentation/en-us/openshift_container_platform/4.14/html-single/networking/index)

## ► Guided Exercise

# Multi-pod Applications

Deploy a multi-pod application into Red Hat OpenShift Container Platform (RHOCP).

### Outcomes

You should be able to use the `oc` command-line utility to:

- Create RHOCP deployments.
- Configure networking.
- Expose applications for external access.

### Before You Begin

As the student user on the workstation machine, use the `lab` command to prepare your system for this exercise.

```
[student@workstation ~]$ lab start openshift-multipod
```

### Instructions

- 1. Log in to the cluster as the `developer` user, and ensure that you use the `ocp-multipod` project.
- 1.1. Log in to the cluster as the `developer` user.

```
[student@workstation ~]$ oc login -u developer -p developer \
https://api.ocp4.example.com:6443
Login successful.

...output omitted...
```

- 1.2. Ensure that you use the `ocp-multipod` project.

```
[student@workstation ~]$ oc project ocp-multipod
Already on project "ocp-multipod" on server "https://api.ocp4.example.com:6443".
```

- 2. Create a Deployment resource for the Gitea application.

- 2.1. Use the `oc create deployment` command to create the Gitea deployment. Configure the deployment port 3030 and use the `registry.ocp4.example.com:8443/redhattraining/podman-gitea:latest` container image.

```
[student@workstation ~]$ oc create deployment gitea --port 3030 \
--image=registry.ocp4.example.com:8443/redhattraining/podman-gitea:latest
Warning: would violate PodSecurity "restricted:v1.24" ...
...output omitted...
deployment.apps/gitea created
```

**Note**

You can ignore pod security warnings for exercises in this course. Red Hat OpenShift uses the Security Context Constraints controller to provide safe defaults for pod security.

2.2. Verify the pod status.

```
[student@workstation ~]$ oc get po
NAME READY STATUS RESTARTS AGE
gitea-6d446c7b48-b89hz 1/1 Running 0 75s
```

If the pod is in the **ContainerCreating** status, repeat the preceding command after a few seconds.

The preceding command uses the **po** short name to refer to the **pod** resource. In the preceding command, the names **po**, **pod**, and **pods** are equivalent.

- ▶ 3. Create a **Deployment** resource for a PostgreSQL database called **gitea-postgres**. The Gitea application uses the database to store data.

You can use the completed file in the `~/D0188/solutions/openshift-multipod` directory.

- 3.1. Use the `oc create deployment` command to create the **gitea-postgres** database.

Configure the deployment port 5432 and use the `registry.ocp4.example.com:8443/rhel9/postgresql-13:1` container image.

Use the `--dry-run=client` and `-o yaml` options to generate a YAML file, and redirect the output to the `postgres.yaml` file.

```
[student@workstation ~]$ oc create deployment gitea-postgres --port 5432 -o yaml \
--image=registry.ocp4.example.com:8443/rhel9/postgresql-13:1 \
--dry-run=client > postgres.yaml
no output expected
```

The preceding command uses output redirection (`>`) to create the `postgres.yaml` file.

- 3.2. Open the `postgres.yaml` file in an editor, such as `gedit`, and add the following environment variables:

```
...file omitted...
 containers:
 - image: registry.ocp4.example.com:8443/rhel9/postgresql-13:1
 name: postgresql-13
 ports:
 - containerPort: 5432
 env:
 - name: POSTGRESQL_USER
 value: gitea
 - name: POSTGRESQL_PASSWORD
 value: gitea
 - name: POSTGRESQL_DATABASE
 value: gitea
...file omitted...
```

To get more information about the `env` property, execute the `oc explain deployment.spec.template.spec.containers.env` command.



### Warning

The YAML format is white-space sensitive. You must use correct indentation.

In the preceding example, the `env:` object indentation is 8 spaces.

### 3.3. Use the `oc create` command to create the deployment.

```
[student@workstation ~]$ oc create -f postgres.yaml
Warning: would violate PodSecurity "restricted:v1.24"...
...output omitted...
deployment.apps/gitea-postgres created
```

If you receive an error, compare your `postgres.yaml` file to the complete `~/DO188/solutions/openshift-multipod/postgres.yaml` file.

### 3.4. Verify the pod status.

```
[student@workstation ~]$ oc get po
NAME READY STATUS RESTARTS AGE
gitea-6d446c7b48-rf578 1/1 Running 0 6m18s
gitea-postgres-86d47f494d-7rl5g 1/1 Running 0 18s
```

## ► 4. Configure networking for the database.

### 4.1. Expose the PostgreSQL database on the `gitea-postgres` hostname.

```
[student@workstation ~]$ oc expose deployment gitea-postgres
service/gitea-postgres exposed
```

### 4.2. Verify that the PostgreSQL service uses the `gitea-postgres` name and the 5432 port.

```
[student@workstation ~]$ oc get svc
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
gitea-postgres ClusterIP 172.30.124.174 <none> 5432/TCP 50s
```

► 5. Configure networking for the Gitea application.

- 5.1. Expose the application within the RHOCP cluster.

```
[student@workstation ~]$ oc expose deployment gitea
service/gitea exposed
```

- 5.2. Expose the gitea service for external access.

```
[student@workstation ~]$ oc expose service gitea
route.route.openshift.io/gitea exposed
```

► 6. Test your application functionality.

- 6.1. Verify the external URL for your application.

```
[student@workstation ~]$ oc get route
NAME HOST/PORT SERVICES PORT WILDCARD
gitea gitea-ocp-multipod.apps.ocp4.example.com gitea 3030 None
```

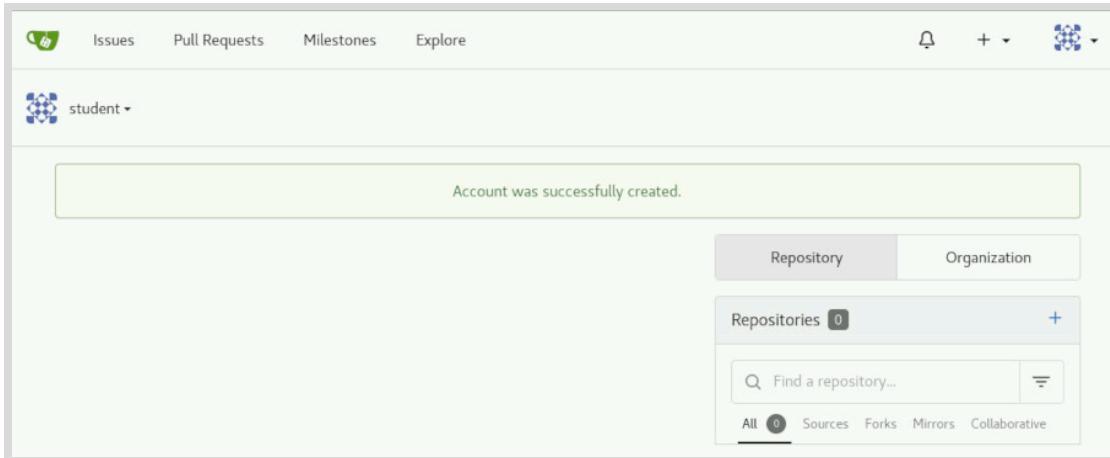
- 6.2. In a web browser, open the `gitea-ocp-multipod.apps.ocp4.example.com` URL and use the following configuration:

- Database type: PostgreSQL
- Host: gitea-postgres:5432
- Username: gitea
- Password: gitea
- Database name: gitea
- Server Domain: gitea-ocp-multipod.apps.ocp4.example.com
- Gitea Base URL: <http://gitea-ocp-multipod.apps.ocp4.example.com>

Then, click **Install Gitea**.

This step is successful if you see the login page.

- 6.3. Optionally, click **Register** to create a user and log in.



**Figure 8.6: A running Gitea instance when logged in**

## Finish

On the **workstation** machine, use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish openshift-multipod
```

## ▶ Lab

# Container Orchestration with Kubernetes and OpenShift

Debug and deploy a multi-container application to the Red Hat OpenShift Container Platform (RHOCP).

## Outcomes

You should be able to:

- Verify and correct the configuration of the Service and Deployment RHOCP objects.
- Deploy RHOCP objects.

## Before You Begin

In this exercise, your task is to deploy the quotes application to RHOCP.

The quotes application uses the quotes-api and quotes-ui containerized microservices.

Your colleague managed to deploy the first tier of the application, the quotes-ui container, to RHOCP. However, the pod crashes and does not respond to requests. Additionally, the colleague faces difficulties when trying to deploy the quotes-api container to RHOCP. You, the RHOCP expert in the company, are tasked with helping your colleague.

As the student user on the workstation machine, use the lab command to:

- Create the ocp-lab project.
- Deploy the quotes-ui microservice.

```
[student@workstation ~]$ lab start openshift-lab
```

The lab script continuously evaluates the objectives of this lab. Keep the script running in a terminal window and complete the objectives of this lab from a new terminal window.

## Instructions

1. Log in to the cluster as the developer user, and ensure that you use the ocp-lab project.

- 1.1. Log in to the cluster as the developer user.

```
[student@workstation ~]$ oc login -u developer -p developer \
https://api.ocp4.example.com:6443
Login successful.

...output omitted...
```

- 1.2. Ensure that you use the ocp-lab project.

```
[student@workstation ~]$ oc project ocp-lab
Already on project "ocp-lab" on server "https://api.ocp4.example.com:6443".
```

- Change into the ~/D0188/labs/openshift-lab/ directory.

This directory contains the quotes-api YAML files that your colleague created. Be aware that the YAML files might contain mistakes.

```
[student@workstation ~]$ cd ~/D0188/labs/openshift-lab/
```

- Use the deployment.yaml file to deploy the quotes-api container in the ocp-lab RHOCP project.
- Use the service.yaml file to configure the quotes-ui container networking in the ocp-lab project.

Configure the service.yaml file to conform to the following requirements:

- The quotes-ui container must reach the quotes-api container at the http://quotes-api:8080 URL.
- The quotes-api container listens on port 8080 by default.
- Deploy the quotes-ui container after the quotes-api container becomes available on the quotes-api host. The application architect advised you to restart the quotes-ui application if it is deployed in the incorrect order.

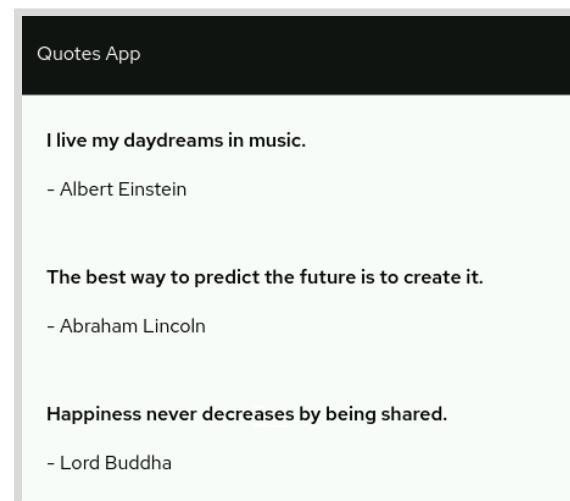


### Note

If you make a mistake, delete and recreate the Service object.

For example, you can use the `oc delete -f service.yaml` command to delete the Service object.

- In a web browser, navigate to `http://quotes-ui-ocp-lab.apps.ocp4.example.com` and verify that the application works.



## Finish

As the **student** user on the **workstation** machine, use the **lab** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish openshift-lab
```

## ► Solution

# Container Orchestration with Kubernetes and OpenShift

Debug and deploy a multi-container application to the Red Hat OpenShift Container Platform (RHOCP).

### Outcomes

You should be able to:

- Verify and correct the configuration of the Service and Deployment RHOCP objects.
- Deploy RHOCP objects.

### Before You Begin

In this exercise, your task is to deploy the quotes application to RHOCP.

The quotes application uses the quotes-api and quotes-ui containerized microservices.

Your colleague managed to deploy the first tier of the application, the quotes-ui container, to RHOCP. However, the pod crashes and does not respond to requests. Additionally, the colleague faces difficulties when trying to deploy the quotes-api container to RHOCP. You, the RHOCP expert in the company, are tasked with helping your colleague.

As the student user on the workstation machine, use the lab command to:

- Create the ocp-lab project.
- Deploy the quotes-ui microservice.

```
[student@workstation ~]$ lab start openshift-lab
```

The lab script continuously evaluates the objectives of this lab. Keep the script running in a terminal window and complete the objectives of this lab from a new terminal window.

## Instructions

1. Log in to the cluster as the developer user, and ensure that you use the ocp-lab project.

- 1.1. Log in to the cluster as the developer user.

```
[student@workstation ~]$ oc login -u developer -p developer \
https://api.ocp4.example.com:6443
Login successful.

...output omitted...
```

- 1.2. Ensure that you use the ocp-lab project.

```
[student@workstation ~]$ oc project ocp-lab
Already on project "ocp-lab" on server "https://api.ocp4.example.com:6443".
```

2. Change into the ~/D0188/labs/openshift-lab/ directory.

This directory contains the quotes-api YAML files that your colleague created. Be aware that the YAML files might contain mistakes.

```
[student@workstation ~]$ cd ~/D0188/labs/openshift-lab/
```

3. Use the deployment.yaml file to deploy the quotes-api container in the ocp-lab RHOCP project.

- 3.1. Try to create the deployment by using the deployment.yaml file.

```
[student@workstation openshift-lab]$ oc create -f deployment.yaml
The Deployment "quotes-api" is invalid: spec.template.metadata.labels: Invalid
value: map[string]string{"app":"quotes-api"}: `selector` does not match template
`labels`
```

The deployment defines an application pod with the app=quotes-api label. However, the spec.selector.matchLabels field uses a different label.

- 3.2. Open the deployment.yaml file in a text editor, such as gedit, and modify the spec.selector.matchLabels field to use the same label as the spec.template.metadata.labels field.

```
apiVersion: apps/v1
kind: Deployment
metadata:
 labels:
 app: quotes-api
 name: quotes-api
spec:
 replicas: 1
 selector:
 matchLabels:
 app: quotes-api
 template:
 metadata:
 labels:
 app: quotes-api
 spec:
 containers:
 - image: registry.ocp4.example.com:8443/redhattraining/podman-quotes-
api:openshift
 name: podman-quotes-api
```

- 3.3. Create the deployment by using the deployment.yaml file.

```
[student@workstation openshift-lab]$ oc create -f deployment.yaml
Warning: would violate PodSecurity ...output omitted...
deployment.apps/quotes-api created
```

**Note**

You can ignore pod security warnings for exercises in this course. Red Hat OpenShift uses the Security Context Constraints controller to provide safe defaults for pod security.

- 3.4. Verify that the quotes-api application pod is in the RUNNING state.

```
[student@workstation openshift-lab]$ oc get po
NAME READY STATUS RESTARTS AGE
quotes-api-6c9f758574-nk8kd 1/1 Running 0 5s
quotes-ui-d7d457674-mljrb 0/1 CrashLoopBackOff 15 (3m9s ago) 55m
```

If the application pod is in the ContainerCreating state, then execute the previous command again after a few seconds.

4. Use the service.yaml file to configure the quotes-ui container networking in the ocp-lab project.

Configure the service.yaml file to conform to the following requirements:

- The quotes-ui container must reach the quotes-api container at the `http://quotes-api:8080` URL.
- The quotes-api container listens on port 8080 by default.
- Deploy the quotes-ui container after the quotes-api container becomes available on the quotes-api host. The application architect advised you to restart the quotes-ui application if it is deployed in the incorrect order.

**Note**

If you make a mistake, delete and recreate the Service object.

For example, you can use the `oc delete -f service.yaml` command to delete the Service object.

- 4.1. Open the service.yaml file in a text editor, such as gedit. Then, configure the service to serve on port 8080.

```
...file omitted...
spec:
 ports:
 - port: 8080
 protocol: TCP
 targetPort: 3000
 selector:
 app: quotes
```

- 4.2. Configure the service to send requests to port 8080.

```
...file omitted...
spec:
 ports:
 - port: 8080
 protocol: TCP
 targetPort: 8080
 selector:
 app: quotes
```

4.3. Configure the service to send requests to pods with the quotes-api label.

```
...file omitted...
spec:
 ports:
 - port: 8080
 protocol: TCP
 targetPort: 8080
 selector:
 app: quotes-api
```

4.4. Configure the service to be available on the quotes-api hostname.

```
apiVersion: v1
kind: Service
metadata:
 labels:
 app: quotes
 name: quotes-api
...file omitted...
```

4.5. Create the service by using the `service.yaml` file.

```
[student@workstation openshift-lab]$ oc create -f service.yaml
service/quotes-api created
```

4.6. Verify the service configuration.

The endpoint IP address might differ in your output.

```
[student@workstation openshift-lab]$ oc describe service quotes-api
Name: quotes-api
Namespace: ocp-lab
Labels: app=quotes
Annotations: <none>
Selector: app=quotes-api
...output omitted...
Port: <unset> 8080/TCP
TargetPort: 8080/TCP
Endpoints: 10.8.0.102:8080
...output omitted...
```

If your output differs from the highlighted output of the previous command, return to the previous steps and ensure you configured your service correctly.

- 4.7. Verify that the quotes-ui container is still failing.

```
[student@workstation openshift-lab]$ oc get po
NAME READY STATUS RESTARTS AGE
quotes-api-6c9f758574-nk8kd 1/1 Running 0 20m
quotes-ui-d7d457674-mljrb 0/1 CrashLoopBackoff 15 (3m9s ago) 55m
```

- 4.8. Restart the quotes-ui container.

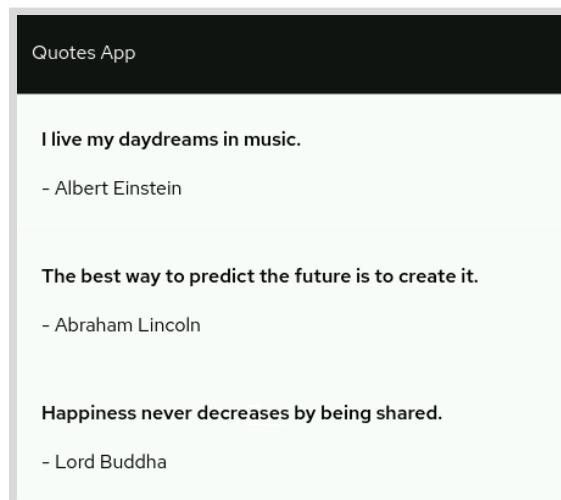
You can delete containers that contain the app=quotes-ui label, and let the quotes-ui deployment recreate the container.

```
[student@workstation openshift-lab]$ oc delete pod -l app=quotes-ui
pod "quotes-ui-d7d457674-9cw7l" deleted
```

Then, verify that the quotes-ui deployment created a new container.

```
[student@workstation openshift-lab]$ oc get po
NAME READY STATUS RESTARTS AGE
quotes-api-6c9f758574-nk8kd 1/1 Running 0 39m
quotes-ui-d7d457674-rbkl7 1/1 Running 0 67s
```

5. In a web browser, navigate to <http://quotes-ui-ocp-lab.apps.ocp4.example.com> and verify that the application works.



## Finish

As the student user on the workstation machine, use the lab command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab finish openshift-lab
```

# Summary

---

- Deploy applications by using the Red Hat OpenShift Container Platform (RHOCP) web console, or command-line utilities, such as `kubectl` or `oc`.
- The smallest unit of deployment for RHOCP is a pod, which represents a group of one or multiple containers that share resources.
- Use controller objects, such as `Deployment`, to create managed pods.
- Services provide a stable IP address and a resolvable DNS name for a pod or a set of pods.
- Routes expose an application outside of an RHOCP cluster.



## Chapter 9

# Comprehensive Review

### Goal

Review tasks from *Red Hat OpenShift Development I: Introduction to Containers with Podman*.

### Sections

- Comprehensive Review

### Lab

- Comprehensive Review

# Comprehensive Review

---

## Objectives

After completing this section, you should have reviewed and refreshed the knowledge and skills that you learned in *Red Hat OpenShift Development I: Introduction to Containers with Podman*.

### Reviewing Red Hat OpenShift Development I: Introduction to Containers with Podman

Before beginning the comprehensive review for this course, you should be comfortable with the topics covered in each chapter. Do not hesitate to ask the instructor for extra guidance or clarification on these topics.

#### **Chapter 1, Introduction and Overview of Containers**

Describe how containers facilitate application development.

- Describe the basics of containers and how containers differ from Virtual Machines.
- Describe container orchestration and the features of Red Hat OpenShift.

#### **Chapter 2, Podman Basics**

Manage and run containers with Podman.

- Run a containerized service with Podman.
- Describe how containers communicate with each other.
- Expose ports to access containerized services.
- Explore running containers.
- List, stop, and delete containers with Podman.

#### **Chapter 3, Container Images**

Navigate container registries to find and manage container images.

- Navigate container registries.
- Pull and manage container images.

#### **Chapter 4, Custom Container Images**

Build custom container images to containerize applications.

- Create a Containerfile by using basic commands.
- Create a Containerfile that uses best practices.
- Run rootless containers with Podman.

## **Chapter 5, Persisting Data**

Run database containers with persistence.

- Describe the process for mounting volumes and common use cases.
- Build containerized databases.

## **Chapter 6, Troubleshooting Containers**

Analyze container logs and configure a remote debugger.

- Read container logs and troubleshoot common container problems.
- Configure a remote debugger during application development.

## **Chapter 7, Multi-container Applications with Compose**

Run multi-container applications with Podman Compose.

- Describe compose and its common use cases.
- Configure a repeatable developer environment with Compose.

## **Chapter 8, Container Orchestration with OpenShift and Kubernetes**

Orchestrate containerized applications with Kubernetes and OpenShift.

- Deploy an application in OpenShift.
- Deploy a multi-pod application to OpenShift and make it externally available.

## ▶ Lab

# Comprehensive Review

Deploy a social media application consisting of multiple containerized components:

- A PostgreSQL database
- A Spring Boot API
- A React UI hosted with NGINX

Use what you have learned throughout this course to create and start a set of containers that host the application and its resources. The Beeper application consists of a UI and an API. The API uses a PostgreSQL database to persist user messages.

## Outcomes

You should be able to:

- Run a container from a public image.
- Manage a container's lifecycle.
- Build efficient container images.
- Create and publish container images.
- Manage container communication and Podman networking.
- Troubleshoot containers.
- Use container storage.

## Before You Begin

As the student user on the `workstation` machine, use the `lab start` command to prepare your system for this exercise.

This command copies the application source code to the workstation and checks that Podman has access to the required images.

The lab script continuously evaluates the objectives of this lab. Keep the script running in a terminal window, and complete the objectives of this lab from a new terminal window.

After each objective, return to the lab script evaluation to see if you have finished the objective successfully. When you finish the objectives, the `lab` command prompts you to execute the `finish` function.

```
[student@workstation ~]$ lab start comprehensive-review
```

The script copies applications and related files into the `~/D0188/labs/comprehensive-review/` directory.

## Specifications

- Create two Podman networks called `beeper-backend` and `beeper-frontend` that have DNS enabled.

- Create a PostgreSQL database container that matches the following criteria:
  - Use the `registry.ocp4.example.com:8443/rhel9/postgresql-13:1` container image.
  - Use `beeper-db` for the name of the container.
  - Connect the container to the `beeper-backend` network.
  - Attach a new volume called `beeper-data` to the `/var/lib/pgsql/data` directory in the container.
  - Pass the following environment variables to the container:
    - Name: `POSTGRESQL_USER`, value: `beeper`
    - Name: `POSTGRESQL_PASSWORD`, value: `beeper123`
    - Name: `POSTGRESQL_DATABASE`, value: `beeper`
- Create a container image for the Beeper API that matches the following criteria:
  - Use a multi-stage build with a builder image to compile the Java application.
  - The build stage should perform the following actions:
    - Use the `registry.ocp4.example.com:8443/ubi8/openjdk-17:1.12` container image. This image uses `/home/jboss` as the working directory.
    - Copy the contents of the `~/D0188/labs/comprehensive-review/beeper-backend` host directory to the image's working directory. Change the owner to the `jboss` user.
    - Use the `mvn -s settings.xml package` command to create the `/home/jboss/target/beeper-1.0.0.jar` binary file.
  - The execution stage should perform the following actions:
    - Use the container image `registry.ocp4.example.com:8443/ubi8/openjdk-17-runtime:1.12`.
    - Copy the `beeper-1.0.0.jar` binary file from the builder image.
    - Run the `java -jar beeper-1.0.0.jar` command to start the Beeper API.
    - Tag the image with `beeper-api:v1`.
- Create a container for the Beeper API that matches the following criteria:
  - Use the `beeper-api:v1` image that you created.
  - Use `beeper-api` for the name of the container.
  - Pass an environment variable to the container called `DB_HOST` with `beeper-db` as the value.
  - Connect the container to the `beeper-backend` and `beeper-frontend` networks.
- Create a container image for the Beeper UI that matches the following criteria:
  - Use a multi-stage build with a builder image to compile the TypeScript React application.
  - The build stage should perform the following actions:

- Use the `registry.ocp4.example.com:8443/ubi9/nodejs-18:1` container image. This image uses `/opt/app-root/src` as the working directory.
- Set `root` as the user.
- Copy the contents of the `~/D0188/labs/comprehensive-review/beeper-ui` host directory into the image's working directory.
- Run the `npm install` command in the application directory to install build dependencies within the image.
- Run the `npm run build` command in the application directory to build the application. This command saves the built artifacts in the `/opt/app-root/src/build` directory.
- The execution stage should perform the following actions:
  - Use the `registry.ocp4.example.com:8443/ubi8/nginx-118:1` container image.
  - Copy the `~/D0188/labs/comprehensive-review/beeper-ui/nginx.conf` host file into the `/etc/nginx/` directory of the execution stage.
  - Copy the built application artifacts from the builder stage into the `/usr/share/nginx/html` directory of the execution stage.
  - Use the `nginx -g "daemon off;"` command to start the application.
  - Tag the image with `beeper-ui:v1`.
- Create a container for the Beeper UI that matches the following criteria:
  - Use the `beeper-ui:v1` image that you created.
  - Use `beeper-ui` for the container name.
  - Map container port `8080` to host port `8080`.
  - Connect the container to the `beeper-frontend` network.
  - Use the `--ulimit nofile=65535:65535` as a build parameter to avoid a build error.
- With the three containers running and configured correctly, the UI is available at `http://localhost:8080`.
  - Click **New Beep** to create a message that is persisted after restarting or recreating the containers.

## Finish

As the `student` user on the `workstation` machine, use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

Press `y` when the `lab start` command prompts you to execute the `finish` function. Alternatively, execute the following command:

```
[student@workstation ~]$ lab finish comprehensive-review
```

## ► Solution

# Comprehensive Review

Deploy a social media application consisting of multiple containerized components:

- A PostgreSQL database
- A Spring Boot API
- A React UI hosted with NGINX

Use what you have learned throughout this course to create and start a set of containers that host the application and its resources. The Beeper application consists of a UI and an API. The API uses a PostgreSQL database to persist user messages.

## Outcomes

You should be able to:

- Run a container from a public image.
- Manage a container's lifecycle.
- Build efficient container images.
- Create and publish container images.
- Manage container communication and Podman networking.
- Troubleshoot containers.
- Use container storage.

## Before You Begin

As the student user on the `workstation` machine, use the `lab start` command to prepare your system for this exercise.

This command copies the application source code to the workstation and checks that Podman has access to the required images.

The lab script continuously evaluates the objectives of this lab. Keep the script running in a terminal window, and complete the objectives of this lab from a new terminal window.

After each objective, return to the lab script evaluation to see if you have finished the objective successfully. When you finish the objectives, the `lab` command prompts you to execute the `finish` function.

```
[student@workstation ~]$ lab start comprehensive-review
```

The script copies applications and related files into the `~/D0188/labs/comprehensive-review/` directory.

1. Create the `beeper-backend` and `beeper-frontend` Podman networks.
  - 1.1. Create the `beeper-backend` Podman network.

```
[student@workstation ~]$ podman network create beeper-backend
beeper-backend
```

- 1.2. Create the beeper-frontend Podman network.

```
[student@workstation ~]$ podman network create beeper-frontend
beeper-frontend
```

2. Create the database container with an attached volume for persistence.

- 2.1. Create the beeper-data volume.

```
[student@workstation ~]$ podman volume create beeper-data
beeper-data
```

- 2.2. Create the PostgreSQL container with beeper-db as the name. Mount the beeper-data volume to the /var/lib/pgsql/data directory in the container.

```
[student@workstation ~]$ podman run -d --name beeper-db \
-e POSTGRESQL_USER=beeper \
-e POSTGRESQL_PASSWORD=beeper123 \
-e POSTGRESQL_DATABASE=beeper \
-v beeper-data:/var/lib/pgsql/data --net beeper-backend \
registry.ocp4.example.com:8443/rhel9/postgresql-13:1
...output omitted...
611...f27
```

3. Create a container for the Beeper API by using a custom Containerfile. The application is available at the ~/D0188/labs/comprehensive-review/beeper-backend directory.

- 3.1. Navigate to the application directory.

```
[student@workstation ~]$ cd ~/D0188/labs/comprehensive-review/beeper-backend
```

- 3.2. Create a Containerfile file with the following contents:

```
FROM registry.ocp4.example.com:8443/ubi8/openjdk-17:1.12 as builder
COPY --chown=jboss . .
RUN mvn -s settings.xml package

FROM registry.ocp4.example.com:8443/ubi8/openjdk-17-runtime:1.12
COPY --from=builder /home/jboss/target/beeper-1.0.0.jar .
ENTRYPOINT ["java", "-jar", "beeper-1.0.0.jar"]
```

- 3.3. Build a container image for the API and use beeper-api:v1 as the tag.

```
[student@workstation beeper-backend]$ podman build -t beeper-api:v1 .
...output omitted...
Successfully tagged localhost/beeper-api:v1
28d...d95
```

- 3.4. Create a container that matches the specifications for the Beeper API by using the tag built in the previous step.

```
[student@workstation beeper-backend]$ podman run -d \
--name beeper-api --net beeper-backend,beeper-frontend \
-e DB_HOST=beeper-db beeper-api:v1
ab4...2ef
```

4. Create a container for the Beeper UI by using a custom Containerfile. The application is available at the ~/D0188/labs/comprehensive-review/beeper-ui directory.

- 4.1. Navigate to the application directory.

```
[student@workstation beeper-backend]$ cd \
~/D0188/labs/comprehensive-review/beeper-ui
```

- 4.2. Create a Containerfile file with the following contents:

```
FROM registry.ocp4.example.com:8443/ubi9/nodejs-18:1 AS builder
USER root
COPY .
RUN npm install && \
 npm run build

FROM registry.ocp4.example.com:8443/ubi8/nginx-118:1
COPY nginx.conf /etc/nginx/
COPY --from=builder /opt/app-root/src/build /usr/share/nginx/html
CMD nginx -g "daemon off;"
```

- 4.3. Build a container image for the UI and use beeper-ui:v1 as the tag.

```
[student@workstation beeper-ui]$ podman build --ulimit nofile=65535:65535 \
-t beeper-ui:v1 .
...output omitted...
Successfully tagged localhost/beeper-ui:v1
7fd...ee1
```

- 4.4. Create a container that matches the specifications for the UI by using the tag built in the previous step.

```
[student@workstation beeper-ui]$ podman run -d \
--name beeper-ui --net beeper-frontend \
-p 8080:8080 beeper-ui:v1
620...a5c
```

- 4.5. Optionally, access the UI by using a web browser to navigate to `http://localhost:8080`. The UI shows the list of user-created messages. To create a new message, click **New Beep**.

## Finish

As the **student** user on the **workstation** machine, use the **lab** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

Press **y** when the **lab start** command prompts you to execute the **finish** function.  
Alternatively, execute the following command:

```
[student@workstation ~]$ lab finish comprehensive-review
```