

Join the explorers, builders, and individuals who boldly offer new solutions to old problems. For open source, innovation is only possible because of the people behind it.

RED HAT® TRAINING+ CERTIFICATION

STUDENT WORKBOOK (ROLE)

OCP 4.0 DO180

INTRODUCTION TO CONTAINERS, KUBERNETES, AND RED HAT OPENSHIFT

Edition 1



INTRODUCTION TO CONTAINERS, KUBERNETES, AND RED HAT OPENSHIFT



OCP 4.0 D0180
Introduction to Containers, Kubernetes, and Red Hat
OpenShift
Edition 1 20190613
Publication date 20190613

Authors: Zach Guterman, Dan Kolepp, Eduardo Ramirez Ronco, Jordi Sola Alaball
Editor: Seth Kenlon, Dave Sacco, Connie Petlitzer

Copyright © 2018 Red Hat, Inc.

The contents of this course and all its modules and related materials, including handouts to audience members, are
Copyright © 2018 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 e-mail training@redhat.com or phone toll-free (USA) +1 (866) 626-2994 or +1 (919) 754-3700.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, Hibernate, Fedora, the Infinity Logo, and RHCE are trademarks of Red Hat, Inc., registered 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 and/or its affiliates.

XFS® is a registered trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

The OpenStack® Word Mark and OpenStack Logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Contributors: Michael Jarrett and Forrest Taylor

Document Conventions	vii
Introduction	ix
DO180: Introduction to Containers, Kubernetes, and Red Hat OpenShift	ix
Orientation to the Classroom Environment	x
Internationalization	xiii
1. Introducing Container Technology	1
Overview of Container Technology	2
Quiz: Overview of Container Technology	5
Overview of Container Architecture	9
Quiz: Overview of Container Architecture	12
Overview of Kubernetes and OpenShift	14
Quiz: Describing Kubernetes and OpenShift	17
Summary	19
2. Creating Containerized Services	21
Provisioning Containerized Services	22
Guided Exercise: Creating a MySQL Database Instance	28
Lab: Creating Containerized Services	31
Summary	36
3. Managing Containers	37
Managing the Life Cycle of Containers	38
Guided Exercise: Managing a MySQL Container	46
Attaching Persistent Storage to Containers	50
Guided Exercise: Persisting a MySQL Database	53
Accessing Containers	56
Guided Exercise: Loading the Database	60
Lab: Managing Containers	63
Summary	73
4. Managing Container Images	75
Accessing Registries	76
Quiz: Working With Registries	82
Manipulating Container Images	86
Guided Exercise: Creating a Custom Apache Container Image	92
Lab: Managing Images	97
Summary	106
5. Creating Custom Container Images	107
Designing Custom Container Images	108
Quiz: Approaches to Container Image Design	112
Building Custom Container Images with Dockerfiles	114
Guided Exercise: Creating a Basic Apache Container Image	119
Lab: Creating Custom Container Images	123
Summary	130
6. Deploying Containerized Applications on OpenShift	131
Describing Kubernetes and OpenShift Architecture	132
Quiz: Describing Kubernetes and OpenShift	138
Creating Kubernetes Resources	142
Guided Exercise: Deploying a Database Server on OpenShift	152
Creating Routes	157
Guided Exercise: Exposing a Service as a Route	161
Creating Applications with Source-to-Image	166
Guided Exercise: Creating a Containerized Application with Source-to-Image	176
Creating Applications with the OpenShift Web Console	183

Guided Exercise: Creating an Application with the Web Console	188
Lab: Deploying Containerized Applications on OpenShift	200
Summary	204
7. Deploying Multi-Container Applications	205
Considerations for Multi-Container Applications	206
Guided Exercise: Deploying the Web Application and MySQL Containers	211
Deploying a Multi-Container Application on OpenShift	218
Guided Exercise: Creating an Application with a Template	228
Lab: Deploying Multi-Container Applications	235
Summary	242
8. Troubleshooting Containerized Applications	243
Troubleshooting S2I Builds and Deployments	244
Guided Exercise: Troubleshooting an OpenShift Build	249
Troubleshooting Containerized Applications	257
Guided Exercise: Configuring Apache Container Logs for Debugging	263
Lab: Troubleshooting Containerized Applications	266
Summary	277
9. Comprehensive Review	279
Comprehensive Review	280
Lab: Containerizing and Deploying a Software Application	283
A. Implementing Microservices Architecture	293
Implementing Microservices Architectures	294
Guided Exercise: Refactoring the To Do List Application	298
Summary	302

DOCUMENT CONVENTIONS



REFERENCES

"References" describe where to find external documentation 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 a trick that makes your life easier.



IMPORTANT

"Important" boxes detail things that are easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring a box labeled "Important" will not cause data loss, but may cause irritation and frustration.



WARNING

"Warnings" should not be ignored. Ignoring warnings will most likely cause data loss.

INTRODUCTION

DO180: INTRODUCTION TO CONTAINERS, KUBERNETES, AND RED HAT OPENSHIFT

DO180: Introduction to Containers, Kubernetes, and Red Hat OpenShift is a hands-on course that teaches students how to create, deploy, and manage containers using Podman, Kubernetes, and the Red Hat OpenShift Container Platform.

One of the key tenants of the DevOps movement is continuous integration and continuous deployment. Containers have become a key technology for the configuration and deployment of applications and microservices. Red Hat OpenShift Container Platform is an implementation of Kubernetes, a container orchestration system.

COURSE OBJECTIVES

- Demonstrate knowledge of the container ecosystem.
- Manage Linux containers using Podman.
- Deploy containers on a Kubernetes cluster using the OpenShift Container Platform.
- Demonstrate basic container design and the ability to build container images.
- Implement a container-based architecture using knowledge of containers, Kubernetes, and OpenShift.

AUDIENCE

- System Administrators
- Developers
- IT Leaders and Infrastructure Architects

PREREQUISITES

Students should meet one or more of the following prerequisites:

- Be able to use a Linux terminal session and issue operating system commands. An RHCSA certification is recommended but not required.
- Have experience with web application architectures and their corresponding technologies.

ORIENTATION TO THE CLASSROOM ENVIRONMENT

In this course, students will do most hands-on practice exercises and lab work with a computer system referred to as **workstation**. This is a virtual machine (VM), which has the host name `workstation.lab.example.com`.

A second VM, **services**, with the host name `services.lab.example.com`, hosts supporting services that would be provided by a typical corporation for its developers:

- A private container registry containing the images needed for the course.
- A Git server that stores the source code for the applications developed during the course.
- A Nexus server with a repository of modules for Node.js development.

A third VM, **master0**, with the host name `master0.lab.example.com`, hosts the OpenShift Container Platform cluster.

The fourth and fifth VMs, with the host names `worker0.lab.example.com` and `worker1.lab.example.com` respectively, are worker nodes in the OpenShift Container Platform cluster.

A sixth VM, **lb**, with the host name `lb.lab.example.com`, is the load balancer for the OpenShift Container Platform cluster.

All student machines have a standard user account, **student**, with the password **student**. Access to the **root** account is available from the **student** account, using the **sudo** command.

The following table lists the virtual machines that are available in the classroom environment:

Classroom Machines

MACHINE NAME	IP ADDRESSES	ROLE
<code>content.example.com</code> , <code>materials.example.com</code> , <code>classroom.example.com</code>	172.25.254.254, 172.25.252.254	Classroom utility server
<code>workstation.lab.example.com</code> , <code>workstationX.example.com</code>	172.25.250.254, 172.25.252.X	Student graphical workstation
<code>lb.lab.example.com</code>	172.25.250.42	Load balancer for the OpenShift Container Platform cluster
<code>master0.lab.example.com</code>	172.25.250.11	OpenShift Container Platform cluster server
<code>worker0.lab.example.com</code>	172.25.250.51	OpenShift Container Platform cluster node

MACHINE NAME	IP ADDRESSES	ROLE
worker1.lab.example.com	172.25.250.52	OpenShift Container Platform cluster node
services.lab.example.com, registry.lab.example.com	172.25.250.13	Classroom private registry

The environment runs a central utility server, `classroom.example.com`, which acts as a NAT router for the classroom network to the outside world. It provides DNS, DHCP, HTTP, and other content services to students. It uses two names, `content.example.com` and `materials.example.com`, to provide course content used in the practice and lab exercises.

The `workstation.lab.example.com` student virtual machine acts as a NAT router between the student network (172.25.250.0/24) and the classroom physical network (172.25.252.0/24). `workstation.lab.example.com` is also known as `workstationX.example.com`, where X in the host name will be a number that varies from student to student.

LAB EXERCISE SETUP AND GRADING

Most activities use the **lab** command, executed on `workstation`, to prepare and evaluate the exercise. The **lab** command takes two arguments: the activity's name and a subcommand of **start**, **grade**, or **finish**.

- The **start** subcommand is used at the beginning of an exercise. It verifies that the systems are ready for the activity, possibly making any necessary configuration changes.
 - The **grade** subcommand is executed at the end of a lab. It provides external confirmation that the activity's requested steps were performed correctly.
- Guided exercises do not support a **grade** subcommand.
- The **finish** subcommand is executed as the last step of every exercise and lab. It executes an necessary administrative tasks to complete the activity, such as selectively cleaning up host changes that result from the activity.

In a Red Hat Online Learning classroom, students are assigned remote computers that are accessed through a web application hosted at `rol.redhat.com` [<http://rol.redhat.com>]. Students should log in to this machine using the user credentials they provided when registering for the class.

Controlling the stations

The state of each virtual machine in the classroom is displayed on the page found under the Online Lab tab.

Machine States

MACHINE STATE	DESCRIPTION
STARTING	The machine is in the process of booting.
STARTED	The machine is running and available (or, when booting, soon will be).
STOPPING	The machine is in the process of shutting down.

MACHINE STATE	DESCRIPTION
STOPPED	The machine is completely shut down. Upon starting, the machine will boot into the same state as when it was shut down (the disk will have been preserved).
PUBLISHING	The initial creation of the virtual machine is being performed.
WAITING_TO_START	The virtual machine is waiting for other virtual machines to start.

Depending on the state of a machine, a selection of the following actions will be available.

Classroom/Machine Actions

BUTTON OR ACTION	DESCRIPTION
PROVISION LAB	Create the ROL classroom. This creates all of the virtual machines needed for the classroom and starts them. This will take several minutes to complete.
DELETE LAB	Caution: Any work generated on the disks is lost.
START LAB	Start all machines in the classroom.
SHUTDOWN LAB	Stop all machines in the classroom.
OPEN CONSOLE	<code>workstation.lab.example.com ssh</code>
ACTION → Start	“power on”
ACTION → Shutdown	Gracefully shut down the machine, preserving the contents of its disk.
ACTION → Power Off	Forcefully shut down the machine, preserving the contents of its disk. This is equivalent to removing the power from a physical machine.
ACTION → Reset	Caution: Any work generated on the disk is lost.

At the start of a lab exercise, if an instruction to reset `workstation` appears, click ACTION → Reset for the `workstation` virtual machine. Likewise, if an instruction to reset `infrastructure` appears, click ACTION → Reset for the `infrastructure` virtual machine.

At the start of a lab exercise, if an instruction to reset all virtual machines appears, click DELETE LAB to delete the classroom environment. After it has been deleted, click PROVISION LAB to create a fresh version of the classroom systems.

The Autostop Timer

The Red Hat Online Learning enrollment entitles students to a certain amount of computer time. To help conserve time, the ROL classroom has an associated countdown timer, which will shut down the classroom environment when the timer expires.

To adjust the timer, click MODIFY. A New Autostop Time dialog opens. Set the autostop time in hours and minutes (note: there is a ten hour maximum time). Click ADJUST TIME to adjust the time accordingly.

INTERNATIONALIZATION

PER-USER LANGUAGE SELECTION

Your users might prefer to use a different language for their desktop environment than the system-wide default. They might also want to use a different keyboard layout or input method for their account.

Language Settings

In the GNOME desktop environment, the user might be prompted to set their preferred language and input method on first login. If not, then the easiest way for an individual user to adjust their preferred language and input method settings is to use the Region & Language application.

You can start this application in two ways. You can run the command **gnome-control-center region** from a terminal window, or on the top bar, from the system menu in the right corner, select the settings button (which has a crossed screwdriver and wrench for an icon) from the bottom left of the menu.

In the window that opens, select Region & Language. Click the Language box and select the preferred language from the list that appears. This also updates the Formats setting to the default for that language. The next time you log in, these changes will take full effect.

These settings affect the GNOME desktop environment and any applications such as **gnome-terminal** that are started inside it. However, by default they do not apply to that account if accessed through an **ssh** login from a remote system or a text-based login on a virtual console (such as **tty5**).



NOTE

You can make your shell environment use the same **LANG** setting as your graphical environment, even when you log in through a text-based virtual console or over **ssh**. One way to do this is to place code similar to the following in your `~/.bashrc` file. This example code will set the language used on a text login to match the one currently set for the user's GNOME desktop environment:

```
i=$(grep 'Language=' /var/lib/AccountsService/users/${USER} \
    | sed 's/Language=//')
if [ "$i" != "" ]; then
    export LANG=$i
fi
```

Japanese, Korean, Chinese, and other languages with a non-Latin character set might not display properly on text-based virtual consoles.

Individual commands can be made to use another language by setting the **LANG** variable on the command line:

```
[user@host ~]$ LANG=fr_FR.utf8 date
```

jeu. avril 25 17:55:01 CET 2019

Subsequent commands will revert to using the system's default language for output. The **locale** command can be used to determine the current value of LANG and other related environment variables.

Input Method Settings

GNOME 3 in Red Hat Enterprise Linux 7 or later automatically uses the IBus input method selection system, which makes it easy to change keyboard layouts and input methods quickly.

The Region & Language application can also be used to enable alternative input methods. In the Region & Language application window, the Input Sources box shows what input methods are currently available. By default, English (US) may be the only available method. Highlight English (US) and click the keyboard icon to see the current keyboard layout.

To add another input method, click the + button at the bottom left of the Input Sources window. An Add an Input Source window will open. Select your language, and then your preferred input method or keyboard layout.

When more than one input method is configured, the user can switch between them quickly by typing **Super+Space** (sometimes called **Windows+Space**). A *status indicator* will also appear in the GNOME top bar, which has two functions: It indicates which input method is active, and acts as a menu that can be used to switch between input methods or select advanced features of more complex input methods.

Some of the methods are marked with gears, which indicate that those methods have advanced configuration options and capabilities. For example, the Japanese Japanese (Kana Kanji) input method allows the user to pre-edit text in Latin and use **Down Arrow** and **Up Arrow** keys to select the correct characters to use.

US English speakers may also find this useful. For example, under English (United States) is the keyboard layout English (international AltGr dead keys), which treats **AltGr** (or the right **Alt**) on a PC 104/105-key keyboard as a "secondary shift" modifier key and dead key activation key for typing additional characters. There are also Dvorak and other alternative layouts available.



NOTE

Any Unicode character can be entered in the GNOME desktop environment if you know the character's Unicode code point. Type **Ctrl+Shift+U**, followed by the code point. After **Ctrl+Shift+U** has been typed, an underlined **u** will be displayed to indicate that the system is waiting for Unicode code point entry.

For example, the lowercase Greek letter lambda has the code point U+03BB, and can be entered by typing **Ctrl+Shift+U**, then **03BB**, then **Enter**.

SYSTEM-WIDE DEFAULT LANGUAGE SETTINGS

The system's default language is set to US English, using the UTF-8 encoding of Unicode as its character set (**en_US.utf8**), but this can be changed during or after installation.

From the command line, the **root** user can change the system-wide locale settings with the **localectl** command. If **localectl** is run with no arguments, it displays the current system-wide locale settings.

To set the system-wide default language, run the command **localectl set-locale** **LANG=locale**, where *locale* is the appropriate value for the LANG environment variable from the "Language Codes Reference" table in this chapter. The change will take effect for users on their next login, and is stored in **/etc/locale.conf**.

```
[root@host ~]# localectl set-locale LANG=fr_FR.utf8
```

In GNOME, an administrative user can change this setting from Region & Language by clicking the Login Screen button at the upper-right corner of the window. Changing the Language of the graphical login screen will also adjust the system-wide default language setting stored in the **/etc/locale.conf** configuration file.



IMPORTANT

Text-based virtual consoles such as **tty4** are more limited in the fonts they can display than terminals in a virtual console running a graphical environment, or pseudoterminals for **ssh** sessions. For example, Japanese, Korean, and Chinese characters may not display as expected on a text-based virtual console. For this reason, you should consider using English or another language with a Latin character set for the system-wide default.

Likewise, text-based virtual consoles are more limited in the input methods they support, and this is managed separately from the graphical desktop environment. The available global input settings can be configured through **localectl** for both text-based virtual consoles and the graphical environment. See the **localectl(1)** and **vconsole.conf(5)** man pages for more information.

LANGUAGE PACKS

Special RPM packages called *langpacks* install language packages that add support for specific languages. These langpacks use dependencies to automatically install additional RPM packages containing localizations, dictionaries, and translations for other software packages on your system.

To list the langpacks that are installed and that may be installed, use **yum list langpacks-***:

```
[root@host ~]# yum list langpacks-*  
Updating Subscription Management repositories.  
Updating Subscription Management repositories.  
Installed Packages  
langpacks-en.noarch      1.0-12.el8        @AppStream  
Available Packages  
langpacks-af.noarch       1.0-12.el8        rhel-8-for-x86_64-appstream-rpms  
langpacks-am.noarch       1.0-12.el8        rhel-8-for-x86_64-appstream-rpms  
langpacks-ar.noarch       1.0-12.el8        rhel-8-for-x86_64-appstream-rpms  
langpacks-as.noarch       1.0-12.el8        rhel-8-for-x86_64-appstream-rpms  
langpacks-ast.noarch      1.0-12.el8        rhel-8-for-x86_64-appstream-rpms  
...output omitted...
```

To add language support, install the appropriate langpacks package. For example, the following command adds support for French:

```
[root@host ~]# yum install langpacks-fr
```

Introduction

Use **yum repoquery --whatsonplements** to determine what RPM packages may be installed by a langpack:

```
[root@host ~]# yum repoquery --whatsonplements langpacks-fr
Updating Subscription Management repositories.
Updating Subscription Management repositories.
Last metadata expiration check: 0:01:33 ago on Wed 06 Feb 2019 10:47:24 AM CST.
glibc-langpack-fr-0:2.28-18.el8.x86_64
gnome-getting-started-docs-fr-0:3.28.2-1.el8.noarch
 hunspell-fr-0:6.2-1.el8.noarch
 hyphen-fr-0:3.0-1.el8.noarch
 libreoffice-langpack-fr-1:6.0.6.1-9.el8.x86_64
 man-pages-fr-0:3.70-16.el8.noarch
 mythes-fr-0:2.3-10.el8.noarch
```

**IMPORTANT**

Langpacks packages use RPM *weak dependencies* in order to install supplementary packages only when the core package that needs it is also installed.

For example, when installing *langpacks-fr* as shown in the preceding examples, the *mythes-fr* package will only be installed if the *mythes* thesaurus is also installed on the system.

If *mythes* is subsequently installed on that system, the *mythes-fr* package will also automatically be installed due to the weak dependency from the already installed *langpacks-fr* package.

**REFERENCES**

locale(7), localectl(1), locale.conf(5), vconsole.conf(5), unicode(7), and utf-8(7) man pages

Conversions between the names of the graphical desktop environment's X11 layouts and their names in **localectl** can be found in the file **/usr/share/X11/xkb/rules/base.lst**.

LANGUAGE CODES REFERENCE

**NOTE**

This table might not reflect all langpacks available on your system. Use **yum info langpacks-SUFFIX** to get more information about any particular langpacks package.

Language Codes

LANGUAGE	LANGPACKS SUFFIX	\$LANG VALUE
English (US)	en	en_US.utf8

LANGUAGE	LANGPACKS SUFFIX	\$LANG VALUE
Assamese	as	as_IN.utf8
Bengali	bn	bn_IN.utf8
Chinese (Simplified)	zh_CN	zh_CN.utf8
Chinese (Traditional)	zh_TW	zh_TW.utf8
French	fr	fr_FR.utf8
German	de	de_DE.utf8
Gujarati	gu	gu_IN.utf8
Hindi	hi	hi_IN.utf8
Italian	it	it_IT.utf8
Japanese	ja	ja_JP.utf8
Kannada	kn	kn_IN.utf8
Korean	ko	ko_KR.utf8
Malayalam	ml	ml_IN.utf8
Marathi	mr	mr_IN.utf8
Odia	or	or_IN.utf8
Portuguese (Brazilian)	pt_BR	pt_BR.utf8
Punjabi	pa	pa_IN.utf8
Russian	ru	ru_RU.utf8
Spanish	es	es_ES.utf8
Tamil	ta	ta_IN.utf8
Telugu	te	te_IN.utf8

CHAPTER 1

INTRODUCING CONTAINER TECHNOLOGY

GOAL

Describe how applications run in containers orchestrated by Red Hat OpenShift Container Platform.

OBJECTIVES

- Describe the difference between container applications and traditional deployments.
- Describe the basics of container architecture.
- Describe the benefits of orchestrating applications and OpenShift Container Platform.

SECTIONS

- Overview of Container Technology (and Quiz)
- Overview of Container Architecture (and Quiz)
- Overview of Kubernetes and OpenShift (and Quiz)

OVERVIEW OF CONTAINER TECHNOLOGY

OBJECTIVES

After completing this section, students should be able to describe the difference between container applications and traditional deployments.

CONTAINERIZED APPLICATIONS

Software applications typically depend on other libraries, configuration files, or services that are provided by the runtime environment. The traditional runtime environment for a software application is a physical host or virtual machine, and application dependencies are installed as part of the host.

For example, consider a Python application that requires access to a common shared library that implements the TLS protocol. Traditionally, a system administrator installs the required package that provides the shared library before installing the Python application.

The major drawback to traditionally deployed software application is that the application's dependencies are entangled with the runtime environment. An application may break when any updates or patches are applied to the base operating system (OS).

For example, an OS update to the TLS shared library removes TLS 1.0 as a supported protocol. This breaks the deployed Python application because it is written to use the TLS 1.0 protocol for network requests. This forces the system administrator to roll back the OS update to keep the application running, preventing other applications from using the benefits of the updated package. Therefore, a company developing traditional software applications may require a full set of tests to guarantee that an OS update does not affect applications running on the host.

Furthermore, a traditionally deployed application must be stopped before updating the associated dependencies. To minimize application downtime, organizations design and implement complex systems to provide high availability of their applications. Maintaining multiple applications on a single host often becomes cumbersome, and any deployment or update has the potential to break one of the organization's applications.

Figure 1.1 describes the difference between applications running as containers and applications running on the host operating system.

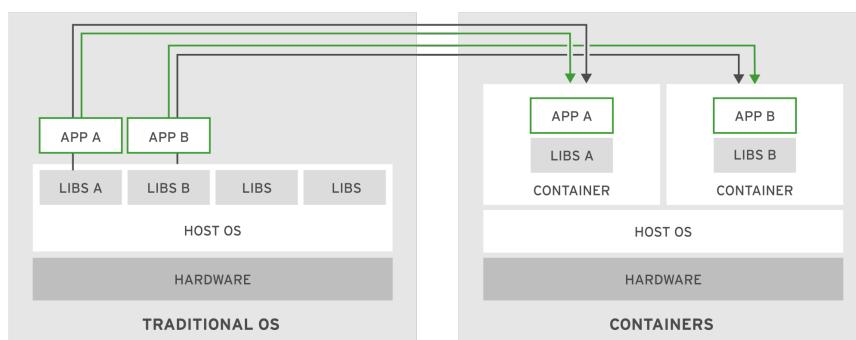


Figure 1.1: Container versus operating system differences

Alternatively, a software application can be deployed using a *container*. A container is a set of one or more processes that are isolated from the rest of the system. Containers provide many of the same benefits as virtual machines, such as security, storage, and network isolation. Containers require far fewer hardware resources and are quick to start and terminate. They also isolate the libraries and the runtime resources (such as CPU and storage) for an application to minimize the impact of any OS update to the host OS, as described in Figure 1.1.

The use of containers not only helps with the efficiency, elasticity, and reusability of the hosted applications, but also with application portability. The *Open Container Initiative* provides a set of industry standards that define a container runtime specification and a container image specification. The image specification defines the format for the bundle of files and metadata that form a container image. When you build an application as a container image, which complies with the OCI standard, you can use any OCI-compliant container engine to execute the application.

There are many *container engines* available to manage and execute individual containers, including Rocket, Drawbridge, LXC, Docker, and Podman. Podman is available in Red Hat Enterprise Linux 7.6 and later, and is used in this course to start, manage, and terminate individual containers.

The following are other major advantages to using containers:

Low hardware footprint

Containers use OS internal features to create an isolated environment where resources are managed using OS facilities such as namespaces and cgroups. This approach minimizes the amount of CPU and memory overhead compared to a virtual machine hypervisor. Running an application in a VM is a way to create isolation from the running environment, but it requires a heavy layer of services to support the same low hardware footprint isolation provided by containers.

Environment isolation

Containers work in a closed environment where changes made to the host OS or other applications do not affect the container. Because the libraries needed by a container are self-contained, the application can run without disruption. For example, each application can exist in its own container with its own set of libraries. An update made to one container does not affect other containers.

Quick deployment

Containers deploy quickly because there is no need to install the entire underlying operating system. Normally, to support the isolation, a new OS installation is required on a physical host or VM, and any simple update might require a full OS restart. A container restart does not require stopping any services on the host OS.

Multiple environment deployment

In a traditional deployment scenario using a single host, any environment differences could break the application. Using containers, however, all application dependencies and environment settings are encapsulated in the container image.

Reusability

The same container can be reused without the need to set up a full OS. For example, the same database container that provides a production database service can be used by each developer to create a development database during application development. Using containers, there is no longer a need to maintain separate production and development database servers. A single container image is used to create instances of the database service.

Often, a software application with all of its dependent services (databases, messaging, file systems) are made to run in a single container. This can lead to the same problems associated with traditional software deployments to virtual machines or physical hosts. In these instances, a multicontainer deployment may be more suitable.

Further, containers are an ideal approach when using microservices for application development. Each service is encapsulated in a lightweight and reliable container environment that can be deployed to a production or development environment. The collection of containerized services required by an application can be hosted on a single machine, removing the need to manage a machine for each service.

In contrast, many applications are not well suited for a containerized environment. For example, applications accessing low-level hardware information, such as memory, file systems, and devices may be unreliable due to container limitations.



REFERENCES

Home - Open Containers Initiative

<https://www.opencontainers.org/>

► QUIZ

OVERVIEW OF CONTAINER TECHNOLOGY

Choose the correct answers to the following questions:

► 1. **Which two options are examples of software applications that might run in a container? (Choose two.)**

- a. A database-driven Python application accessing services such as a MySQL database, a file transfer protocol (FTP) server, and a web server on a single physical host.
- b. A Java Enterprise Edition application, with an Oracle database, and a message broker running on a single VM.
- c. An I/O monitoring tool responsible for analyzing the traffic and block data transfer.
- d. A memory dump application tool capable of taking snapshots from all the memory CPU caches for debugging purposes.

► 2. **Which two of the following use cases are best suited for containers? (Choose two.)**

- a. A software provider needs to distribute software that can be reused by other companies in a fast and error-free way.
- b. A company is deploying applications on a physical host and would like to improve its performance by using containers.
- c. Developers at a company need a disposable environment that mimics the production environment so that they can quickly test the code they develop.
- d. A financial company is implementing a CPU-intensive risk analysis tool on their own containers to minimize the number of processors needed.

► 3. **A company is migrating their PHP and Python applications running on the same host to a new architecture. Due to internal policies, both are using a set of custom made shared libraries from the OS, but the latest update applied to them as a result of a Python development team request broke the PHP application. Which two architectures would provide the best support for both applications? (Choose two.)**

- a. Deploy each application to different VMs and apply the custom made shared libraries individually to each VM host.
- b. Deploy each application to different containers and apply the custom made shared libraries individually to each container.
- c. Deploy each application to different VMs and apply the custom made shared libraries to all VM hosts.
- d. Deploy each application to different containers and apply the custom made shared libraries to all containers.

► **4. Which three kinds of applications can be packaged as containers for immediate consumption? (Choose three.)**

- a. A virtual machine hypervisor
- b. A blog software, such as WordPress
- c. A database
- d. A local file system recovery tool
- e. A web server

► SOLUTION

OVERVIEW OF CONTAINER TECHNOLOGY

Choose the correct answers to the following questions:

► **1. Which two options are examples of software applications that might run in a container? (Choose two.)**

- a. A database-driven Python application accessing services such as a MySQL database, a file transfer protocol (FTP) server, and a web server on a single physical host.
- b. A Java Enterprise Edition application, with an Oracle database, and a message broker running on a single VM.
- c. An I/O monitoring tool responsible for analyzing the traffic and block data transfer.
- d. A memory dump application tool capable of taking snapshots from all the memory CPU caches for debugging purposes.

► **2. Which two of the following use cases are best suited for containers? (Choose two.)**

- a. A software provider needs to distribute software that can be reused by other companies in a fast and error-free way.
- b. A company is deploying applications on a physical host and would like to improve its performance by using containers.
- c. Developers at a company need a disposable environment that mimics the production environment so that they can quickly test the code they develop.
- d. A financial company is implementing a CPU-intensive risk analysis tool on their own containers to minimize the number of processors needed.

► **3. A company is migrating their PHP and Python applications running on the same host to a new architecture. Due to internal policies, both are using a set of custom made shared libraries from the OS, but the latest update applied to them as a result of a Python development team request broke the PHP application. Which two architectures would provide the best support for both applications? (Choose two.)**

- a. Deploy each application to different VMs and apply the custom made shared libraries individually to each VM host.
- b. Deploy each application to different containers and apply the custom made shared libraries individually to each container.
- c. Deploy each application to different VMs and apply the custom made shared libraries to all VM hosts.
- d. Deploy each application to different containers and apply the custom made shared libraries to all containers.

► **4. Which three kinds of applications can be packaged as containers for immediate consumption? (Choose three.)**

- a. A virtual machine hypervisor
- b. A blog software, such as WordPress
- c. A database
- d. A local file system recovery tool
- e. A web server

OVERVIEW OF CONTAINER ARCHITECTURE

OBJECTIVES

After completing this section, students should be able to:

- Describe the architecture of Linux containers.
- Install the **podman** utility to manage containers.

INTRODUCING CONTAINER HISTORY

Containers have quickly gained popularity in recent years. However, the technology behind containers has been around for a relatively long time. In 2001, Linux introduced a project named VServer. VServer was the first attempt at running complete sets of processes inside a single server with a high degree of isolation.

From VServer, the idea of isolated processes further evolved and became formalized around the following features of the Linux kernel:

Namespaces

The kernel can isolate specific system resources, usually visible to all processes, by placing the resources within a namespace. Inside a namespace, only processes that are members of that namespace can see those resources. Namespaces can include resources like network interfaces, the process ID list, mount points, IPC resources, and the system's host name information.

Control groups (cgroups)

Control groups partition sets of processes and their children into groups to manage and limit the resources they consume. Control groups place restrictions on the amount of system resources processes might use. Those restrictions keep one process from using too many resources on the host.

Seccomp

Developed in 2005 and introduced to containers circa 2014, Seccomp limits how processes could use system calls. Seccomp defines a security profile for processes, whitelisting the system calls, parameters and file descriptors they are allowed to use.

SELinux

SELinux (Security-Enhanced Linux) is a mandatory access control system for processes. Linux kernel uses SELinux to protect processes from each other and to protect the host system from its running processes. Processes run as a confined SELinux type that has limited access to host system resources.

All of these innovations and features focus around a basic concept: enabling processes to run isolated while still accessing system resources. This concept is the foundation of container technology and the basis for all container implementations. Nowadays, containers are processes in Linux kernel making use of those security features to create an isolated environment. This environment forbids isolated processes from misusing system or other container resources.

A common use case of containers is having several replicas for the same service (for example, a database server) in the same host. Each replica has isolated resources (file system, ports, memory),

so no need for the service to handle resource sharing. Isolation guarantees that a malfunctioning or harmful service does not impact other services or containers in the same host, nor in the underlying system.

DESCRIBING LINUX CONTAINER ARCHITECTURE

From the Linux kernel perspective, a container is a process with restrictions. However, instead of running a single binary file, a container runs an *image*. An image is a file-system bundle that contains all dependencies required to execute a process: files in the file system, installed packages, available resources, running processes, and kernel modules.

Like executable files are the foundation for running processes, images are the foundation for running containers. Running containers use an immutable view of the image, allowing multiple containers to reuse the same image simultaneously. As images are files, they can be managed by versioning systems, improving automation on container and image provisioning.

Container images need to be locally available for the container runtime to execute them, but the images are usually stored and maintained in an *image repository*. An image repository is just a service - public or private - where images can be stored, searched and retrieved. Other features provided by image repositories are remote access, image metadata, authorization or image version control.

There are many different image repositories available, each one offering different features:

- Red Hat Container Catalog [<https://registry.redhat.io>]
- Docker Hub [<https://hub.docker.com>]
- Red Hat Quay [<https://quay.io/>]
- Google Container Registry [<https://cloud.google.com/container-registry/>]
- Amazon Elastic Container Registry [<https://aws.amazon.com/ecr/>]

This course uses a private image registry in a virtual machine where all the required images are stored for faster consumption.

MANAGING CONTAINERS WITH PODMAN

Containers, images, and image registries need to be able to interact with each other. For example, you need to be able to build images and put them into image registries. You also need to be able to retrieve an image from the image registry and build a container from that image.

Podman is an open source tool for managing containers and container images and interacting with image registries. It offers the following key features:

- It uses image format specified by the Open Container Initiative [<https://www.opencontainers.org>] (*OCI*). Those specifications define a standard, community-driven, non-proprietary image format.
- Podman stores local images in local file-system. Doing so avoids unnecessary client/server architecture or having daemons running on local machine.
- Podman follows the same command patterns as the Docker CLI, so there is no need to learn a new toolset.
- Podman is compatible with Kubernetes. Kubernetes can use Podman to manage its containers.

Currently, Podman is only available on Linux systems. To install Podman in Red Hat Enterprise Linux, Fedora or similar RPM-based systems, run **sudo yum install podman** or **sudo dnf install podman**.



REFERENCES

Red Hat Container Catalog

<https://registry.redhat.io>

Podman site

<https://podman.io/>

Open Container Initiative

<https://www.opencontainers.org>

► QUIZ

OVERVIEW OF CONTAINER ARCHITECTURE

Choose the correct answers to the following questions:

- ▶ **1. Which three of the following Linux features are used for running containers? (Choose three.)**
 - a. Namespaces
 - b. Integrity Management
 - c. Security-Enhanced Linux
 - d. Control Groups
- ▶ **2. Which of the following best describes a container image?**
 - a. A virtual machine image from which a container will be created.
 - b. A container blueprint from which a container will be created.
 - c. A runtime environment where an application will run.
 - d. The container's index file used by a registry.
- ▶ **3. Which three of the following components are common across container architecture implementations? (Choose three.)**
 - a. Container runtime
 - b. Container permissions
 - c. Container images
 - d. Container registries
- ▶ **4. What is a container in relation to the Linux kernel?**
 - a. A virtual machine.
 - b. An isolated process with regulated resource access.
 - c. A set of file-system layers exposed by UnionFS.
 - d. An external service providing container images.

► SOLUTION

OVERVIEW OF CONTAINER ARCHITECTURE

Choose the correct answers to the following questions:

- ▶ **1. Which three of the following Linux features are used for running containers? (Choose three.)**
 - a. Namespaces
 - b. Integrity Management
 - c. Security-Enhanced Linux
 - d. Control Groups
- ▶ **2. Which of the following best describes a container image?**
 - a. A virtual machine image from which a container will be created.
 - b. A container blueprint from which a container will be created.
 - c. A runtime environment where an application will run.
 - d. The container's index file used by a registry.
- ▶ **3. Which three of the following components are common across container architecture implementations? (Choose three.)**
 - a. Container runtime
 - b. Container permissions
 - c. Container images
 - d. Container registries
- ▶ **4. What is a container in relation to the Linux kernel?**
 - a. A virtual machine.
 - b. An isolated process with regulated resource access.
 - c. A set of file-system layers exposed by UnionFS.
 - d. An external service providing container images.

OVERVIEW OF KUBERNETES AND OPENSHIFT

OBJECTIVES

After completing this section, students should be able to:

- Identify the limitations of Linux containers and the need for container orchestration.
- Describe the Kubernetes container orchestration tool.
- Describe Red Hat OpenShift Container Platform (RHOCP).

LIMITATIONS OF CONTAINERS

Containers provide an easy way to package and run services. As the number of containers managed by an organization grows, the work of manually starting them rises exponentially along with the need to quickly respond to external demands.

When using containers in a production environment, enterprises often require:

- Easy communication between a large number of services.
- Resource limits on applications regardless of the number of containers running them.
- Respond to application usage spikes to increase or decrease running containers.
- React to service deterioration.
- Gradually roll out a new release to a set of users.

Enterprises often require a container orchestration technology because container runtimes (such as Podman) do not adequately address the above requirements.

KUBERNETES OVERVIEW

Kubernetes is an orchestration service that simplifies the deployment, management, and scaling of containerized applications.

The smallest unit manageable in Kubernetes is a pod. A pod consists of one or more containers with its storage resources and IP that represent a single application. Kubernetes also uses pods to orchestrate the containers inside it and to limit its resources as a single unit.

KUBERNETES FEATURES

Kubernetes offers the following features on top of a container infrastructure:

Service discovery and load balancing

Kubernetes enables inter-service communication by assigning a single DNS entry to each set of containers. This way, the requesting service only needs to know the target's DNS name, allowing the cluster to change the container's location and IP address, leaving the service unaffected. This permits load-balancing the request across the pool of containers providing the service. For example, Kubernetes can evenly split incoming requests to a MySQL service taking into account the availability of the pods.

Horizontal scaling

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

Self-healing

Kubernetes can use user-defined health checks to monitor containers to restart and reschedule them in case of failure.

Automated rollout

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

Secrets and configuration management

You can manage configuration settings and secrets of your applications without rebuilding containers. Application secrets can be user names, passwords, and service endpoints; any configuration settings that need to be kept private.

Operators

Operators are packaged Kubernetes applications that also bring the knowledge of the application's life cycle into the Kubernetes cluster. Applications packaged as Operators use the Kubernetes API to update the cluster's state reacting to changes in the application state.

OPENSHIFT OVERVIEW

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

Beginning with Red Hat OpenShift v4, hosts in an OpenShift cluster all use Red Hat Enterprise Linux CoreOS as the underlying operating system.

Throughout this course, the terms RHOCP and OpenShift are used to refer to the Red Hat OpenShift Container Platform.

OPENSHIFT FEATURES

OpenShift adds the following features to a Kubernetes cluster:

Integrated developer workflow

RHOCP integrates a built-in container registry, CI/CD pipelines, and S2I; a tool to build artifacts from source repositories to container images.

Routes

Easily expose services to the outside world.

Metrics and logging

Include built-in and self-analyzing metrics service and aggregated logging.

Unified UI

OpenShift brings unified tools and a UI to manage all the different capabilities.



REFERENCES

Production-Grade Container Orchestration - Kubernetes

<https://kubernetes.io/>

OpenShift: Container Application Platform by Red Hat, Built on Docker and Kubernetes

<https://www.openshift.com/>

► QUIZ

DESCRIBING KUBERNETES AND OPENSIFT

Choose the correct answers to the following questions:

► 1. **Which three of the following statements are correct regarding container limitations?**

(Choose three.)

- a. Containers are easily orchestrated in large numbers.
- b. Lack of automation increases response time to problems.
- c. Containers do not manage application failure inside them.
- d. Containers are not load-balanced.
- e. Containers are heavily isolated packaged applications.

► 2. **Which two of the following statements are correct regarding Kubernetes? (Choose two.)**

- a. Kubernetes is a container.
- b. Kubernetes can only use Docker containers.
- c. Kubernetes is a container orchestration system.
- d. Kubernetes simplifies management, deployment, and scaling of containerized applications.
- e. Applications managed in a Kubernetes cluster are harder to maintain.

► 3. **Which three of the following statements are true regarding Red Hat OpenShift v4?**

(Choose three.)

- a. OpenShift provides additional features to a Kubernetes infrastructure.
- b. Kubernetes and OpenShift are mutually exclusive.
- c. OpenShift hosts use Red Hat Enterprise Linux as the base operating system.
- d. OpenShift simplifies development incorporating a Source-to-Image technology and CI/CD pipelines.
- e. OpenShift simplifies routing and load balancing.

► 4. **Which statement is not correct regarding the Operator Framework?**

- a. The Operator Framework simplifies building Kubernetes applications.
- b. The Operator Framework is incompatible with Helm charts.
- c. It reduces development work to business logic code.
- d. It supports deployment to multiple Kubernetes clusters.

► SOLUTION

DESCRIBING KUBERNETES AND OPENSHIFT

Choose the correct answers to the following questions:

- ▶ **1. Which three of the following statements are correct regarding container limitations?**
(Choose three.)
 - a. Containers are easily orchestrated in large numbers.
 - b. Lack of automation increases response time to problems.
 - c. Containers do not manage application failure inside them.
 - d. Containers are not load-balanced.
 - e. Containers are heavily isolated packaged applications.
- ▶ **2. Which two of the following statements are correct regarding Kubernetes? (Choose two.)**
 - a. Kubernetes is a container.
 - b. Kubernetes can only use Docker containers.
 - c. Kubernetes is a container orchestration system.
 - d. Kubernetes simplifies management, deployment, and scaling of containerized applications.
 - e. Applications managed in a Kubernetes cluster are harder to maintain.
- ▶ **3. Which three of the following statements are true regarding Red Hat OpenShift v4?**
(Choose three.)
 - a. OpenShift provides additional features to a Kubernetes infrastructure.
 - b. Kubernetes and OpenShift are mutually exclusive.
 - c. OpenShift hosts use Red Hat Enterprise Linux as the base operating system.
 - d. OpenShift simplifies development incorporating a Source-to-Image technology and CI/CD pipelines.
 - e. OpenShift simplifies routing and load balancing.
- ▶ **4. Which statement is not correct regarding the Operator Framework?**
 - a. The Operator Framework simplifies building Kubernetes applications.
 - b. The Operator Framework is incompatible with Helm charts.
 - c. It reduces development work to business logic code.
 - d. It supports deployment to multiple Kubernetes clusters.

SUMMARY

In this chapter, you learned:

- Containers are an isolated application runtime created with very little overhead.
- A container image packages an application with all of its dependencies, making it easier to run the application in different environments.
- Applications such as Podman create containers using features of the standard Linux kernel.
- Container image registries are the preferred mechanism for distributing container images to multiple users and hosts.
- OpenShift orchestrates applications composed of multiple containers using Kubernetes.
- Kubernetes manages load balancing, high availability, and persistent storage for containerized applications.
- OpenShift adds to Kubernetes multitenancy, security, ease of use, and continuous integration and continuous development features.
- OpenShift routes enable external access to containerized applications in a manageable way.

CHAPTER 2

CREATING CONTAINERIZED SERVICES

GOAL

Provision a service using container technology.

OBJECTIVES

- Create a database server from a container image.

SECTIONS

- Provisioning a Containerized Database Server (and Guided Exercise)

LAB

- Creating Containerized Services

PROVISIONING CONTAINERIZED SERVICES

OBJECTIVES

After completing this section, students should be able to:

- Search for and fetch container images with Podman.
- Run and configure containers locally.
- Use the Red Hat Container Catalog.

FETCHING CONTAINER IMAGES WITH PODMAN

Applications can run inside containers as a way to provide them with an isolated and controlled execution environment. Running a containerized application, that is, running an application inside a container, requires a container image, a file system bundle providing all application files, libraries, and dependencies the application needs to run. Container images can be found in image registries: services that allow users to search and retrieve container images. Podman users can use the **search** subcommand to find available images from remote or local registries:

```
[student@workstation ~]$ sudo podman search rhel
INDEX      NAME          DESCRIPTION  STARS OFFICIAL AUTOMATED
redhat.com  registry.access.redhat.com/rhel This plat... 0
...output omitted...
```

After you have found an image, you can use Podman to download it. When using the **pull** subcommand, Podman fetches the image and saves it locally for future use:

```
[student@workstation ~]$ sudo podman pull rhel
Trying to pull registry.access.redhat.com/rhel...Getting image source signatures
Copying blob sha256: ...output omitted...
  72.25 MB / 72.25 MB [=====] 8s
Copying blob sha256: ...output omitted...
  1.20 KB / 1.20 KB [=====] 0s
Copying config sha256: ...output omitted...
  6.30 KB / 6.30 KB [=====] 0s
Writing manifest to image destination
Storing signatures
699d44bc6ea2b9fb23e7899bd4023d3c83894d3be64b12e65a3fe63e2c70f0ef
```

Container images are named based on the following syntax:

registry_name/user_name/image_name:tag

- First **registry_name**, the name of the registry storing the image. It is usually the FQDN of the registry.
- **user_name** stands for the user or organization the image belongs to.
- The **image_name** should be unique in user namespace.

- The tag identifies the image version. If the image name includes no image tag, **latest** is assumed.

**NOTE**

This classroom's Podman installation uses a local registry accessible at `registry.lag.example.com` by default.

After retrieval, Podman stores images locally and you can list them with the **images** subcommand:

```
[student@workstation containers]$ sudo podman images
REPOSITORY          TAG      IMAGE ID   CREATED        SIZE
registry.access.redhat.com/rhel    latest    699d44bc6ea2  4 days ago   214MB
...output omitted...
```

RUNNING CONTAINERS

The **podman run** command runs a container locally based on an image. At a minimum, the command requires the name of the image to execute in the container.

The container image specifies a process that starts inside the container known as the *entry point*. The **podman run** command uses all parameters after the image name as the entry point command for the container. The following example starts a container from a Red Hat Enterprise Linux image. It sets the entry point for this container to the **echo "Hello world"** command.

```
[student@workstation containers]$ sudo podman run rhe17:7.5 echo "Hello world"
Hello world
```

To start a container image as a background process, pass the **-d** option to the **podman run** command:

```
[student@workstation ~]$ sudo podman run -d rhsc1/httpd-24-rhe17:2.4-36.8
ff4ec6d74e9b2a7b55c49f138e56f8bc46fe2a09c23093664fea7feb3dfa1b2
[student@workstation ~]$ sudo podman inspect -l \
> -f "{{.NetworkSettings.IPAddress}}"
10.88.0.68
[student@workstation ~]$ curl http://10.88.0.68:8080
...output omitted...
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
...output omitted...
<title>
Test Page for the Apache HTTP Server on Red Hat Enterprise Linux
</title>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<style type="text/css">
...output omitted...
```

The previous example ran a containerized Apache HTTP server in the background. Then, the example uses the **podman inspect** command to retrieve the container's internal IP address from container metadata. Finally, it uses the IP address to fetch the root page from Apache HTTP server. This response proves the container is still up and running after the **podman run** command.

NOTE

 Most Podman subcommands accept the **-l** flag (**l** for latest) as a replacement for the container id. This flag applies the command to the latest used container in any Podman command.

NOTE

 If the image to be executed is not available locally when using the **podman run** command, Podman automatically uses **pull** to download the image.

When referencing the container, Podman recognizes a container either with the container name or the generated container id. Use the **--name** option to set the container name when running the container with Podman. Container names must be unique. If the **podman run** command includes no container name, Podman generates a unique random name.

If the images require interacting with the user with console input, Podman can redirect container input and output streams to the console. The **run** subcommand requires the **-t** and **-i** flags (or, in short, **-it** flag) to enable interactivity.

NOTE

 Many Podman flags also have an alternative long form; some of these are explained below.

- **-t** is equivalent to **--tty**, meaning a pseudo-tty (pseudo-terminal) is to be allocated for the container.
- **-i** is the same as **--interactive**. When used, standard input is kept open into the container.
- **-d**, or its long form **--detach**, means the container runs in the background (detached). Podman then prints the container id.

See the Podman documentation for the complete list of flags.

The following example starts a Bash terminal *inside* the container, and interactively runs some commands in it:

```
[student@workstation ~]$ sudo podman run -it rhel7:7.5 /bin/bash
bash-4.2# ls
...output omitted...
bash-4.2# whoami
root
bash-4.2# exit
exit
[student@workstation ~]$
```

Some containers need or can use external parameters provided at startup. The most common approach for providing and consuming those parameters is through environment variables. Podman can inject environment variables into containers at startup by adding the **-e** flag to the **run** subcommand:

```
[student@workstation ~]$ sudo podman run -e GREET=Hello -e NAME=RedHat \
> rhel7:7.5 printenv GREET NAME
Hello
RedHat
[student@workstation ~]$
```

The previous example starts a RHEL image container that prints the two environment variables provided as parameters. Another use case for environment variables is setting up credentials into a MySQL database server:

```
[root@workstation ~]# sudo podman run --name mysql-custom \
> -e MYSQL_USER=redhat -e MYSQL_PASSWORD=r3dh4t \
> -d mysql:5.5
```

USING THE RED HAT CONTAINER CATALOG

Red Hat maintains its repository of finely tuned container images. Using this repository provides customers with a layer of protection and reliability against known vulnerabilities, which could potentially be caused by untested images. The standard **podman** command is compatible with the Red Hat Container Catalog. The Red Hat Container Catalog provides a user-friendly interface for searching and exploring container images from the Red Hat repository.

The Container Catalog also serves as a single interface, providing access to different aspects of all the available container images in the repository. It is useful in determining the best image among multiple versions of container images given health index grades. The health index grade indicates how current an image is, and whether it contains the latest security updates.

The Container Catalog also gives access to the errata documentation of an image. It describes the latest bug fixes and enhancements in each update. It also suggests the best technique for pulling an image on each operating system.

The following images highlight some of the features of the Red Hat Container Catalog.

The screenshot shows the Red Hat Container Catalog interface. At the top, there are navigation links for SUBSCRIPTIONS, DOWNLOADS, CONTAINERS, and SUPPORT CASES. The main header includes the Red Hat logo, a CUSTOMER PORTAL link, and tabs for Products & Services, Tools, Security, and Community. A search bar at the top right contains the word "Apache". Below the search bar, a dropdown menu lists "Products - See all 4 Product results" and "Image Repositories - See all 13 Image Repository results". The products listed under "Products" include Red Hat JBoss Web Server, JBoss A-MQ, ScaleOut StateServer, and Hazelcast. Under "Image Repositories", items like Apache httpd 2.4- rhscl/httpd-24-rhel7, Apache 2.4 with PHP 7.0- rhscl/php-70-rhel7, and Apache 2.4 with mod_perl/5.24- rhscl/perl-524-rhel7 are shown. On the left, there are sections for "Recently Updated" and "Your trust". The "Recently Updated" section highlights "rhpam-7/rhpam70-businesscentral-openshift" as the platform for authoring business assets. The "Your trust" section shows a KIE Server entry. The bottom of the page features a footer with links to various Red Hat services and support.

Figure 2.1: Red Hat Container Catalog home page

As displayed above, typing **Apache** in the search box of the Container Catalog displays a suggested list of products and image repositories matching the search pattern. To access the **Apache httpd 2.4** image page, select **Apache httpd 2.4-rhscl/httpd-24-rhel7** from the suggested list.

Apache httpd 2.4 ☆

by Red Hat, Inc. | in Product Red Hat Enterprise Linux

This screenshot shows the detailed overview page for the Apache httpd 2.4 image. At the top, the URL is registry.access.redhat.com/rhscl/httpd-24-rhel7 and it was updated 6 days ago. The image has a Health Index of A. Below the header, there are tabs for Overview (which is selected), Get Latest Image, Tech Details, Documentation, and Tags. The Overview section contains a "Description" block stating that Apache httpd 2.4 is a powerful, efficient, and extensible web server. It supports various features like server-side programming language support and authentication schemes. The "Evaluate Image" section provides a preview and a link to log in to OpenShift Online. The "Repository Specifications" section lists details such as Registry (registry.access.redhat.com), Namespace/Repository (rhscl/httpd-24-rhel7), Release Category (Generally Available), and Application Categories (Web Services). The "Most recent tag" section shows the latest tag is 2.4-55, which was updated 6 days ago. Other tags listed include 2.4-55 (Signed, Unprivileged), Health Index A, Security, and Size (111.9 MB). The RPM Packages section shows 222 packages.

Figure 2.2: Apache httpd 2.4 (rhscl/httpd-24-rhel7) overview image page

CHAPTER 2 | Creating Containerized Services

The *Apache httpd 2.4* panel displays image details and several tabs. This page states that Red Hat maintains the image repository. It also indicates that the image repository relates to Red Hat Enterprise Linux. Under the *Overview* tab, there are other details:

- *Description*: A summary of the image's capabilities.
- *Evaluate Image*: Using **OpenShift Online** (a public PaaS cloud), users can try out the image to validate the functional state of the application in the image.
- *Most Recent Tag*: When the image received its latest update, the latest tag applied to the image, the health of the image, and more.

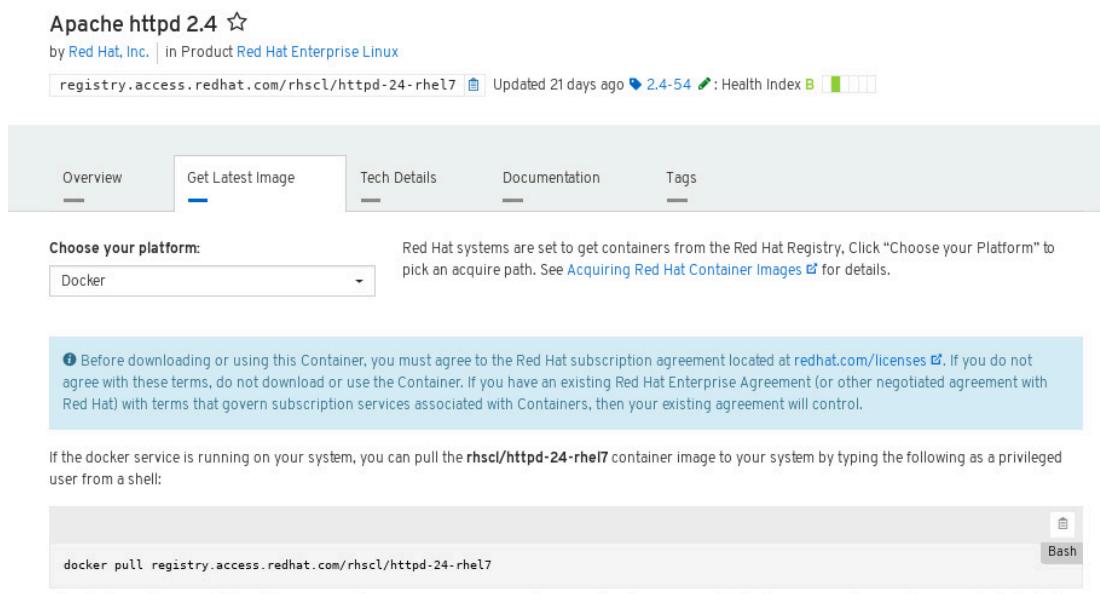


Figure 2.3: Apache httpd 2.4 (rhscl/httpd-24-rhel7) latest image page

The *Get Latest Image* tab provides the procedure to get the most current version of the image. Specify the intended platform for the image from the *Choose your platform* menu, and the page provides the appropriate command to execute to retrieve the image.



REFERENCES

Red Hat Container Catalog
<https://registry.redhat.io>

Quay.io website
<https://quay.io>

► GUIDED EXERCISE

CREATING A MYSQL DATABASE INSTANCE

In this exercise, you will start a MySQL database inside a container, and then create and populate a database.

OUTCOMES

You should be able to start a database from a container image and store information inside the database.

BEFORE YOU BEGIN

Open a terminal on `workstation` as the `student` user and run the following command:

```
[student@workstation ~]$ lab container-create start
```

► 1. Create a MySQL container instance.

- 1.1. Start a container from the Red Hat Software Collections Library MySQL image.

```
[student@workstation ~]$ sudo podman run --name mysql-basic \
> -e MYSQL_USER=user1 -e MYSQL_PASSWORD=mypa55 \
> -e MYSQL_DATABASE=items -e MYSQL_ROOT_PASSWORD=r00tpa55 \
> -d rh scl/mysql57-rhel7:5.7-3.14
Trying to pull ...output omitted...
Copying blob sha256:e373541...output omitted...
69.66 MB / 69.66 MB [=====] 8s
Copying blob sha256:c5d2e94...output omitted...
1.20 KB / 1.20 KB [=====] 0s
Copying blob sha256:b3949ae...output omitted...
62.03 MB / 62.03 MB [=====] 8s
Writing manifest to image destination
Storing signatures
92eaa6b67da0475745b2beffa7e0895391ab34ab3bf1ded99363bb09279a24a0
```

This command downloads the MySQL container image with the **5.7-3.14** tag, and then starts a container-based image. It creates a database named `items`, owned by a user named `user1` with `mypa55` as the password. The database administrator password is set to `r00tpa55` and the container runs in the background.

- 1.2. Verify that the container started without errors.

```
[student@workstation ~]$ sudo podman ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS              NAMES
92eaa6b67da0        ...output omitted...   mysql57-rhel7    ...output omitted...   mysql-basic
```

- 2. Access the container sandbox by running the following command:

```
[student@workstation ~]$ sudo podman exec -it mysql-basic /bin/bash  
bash-4.2$
```

This command starts a Bash shell, running as the `mysql` user inside the **MySQL** container.

- 3. Add data to the database.

- 3.1. Connect to MySQL as the database administrator user (root).

Run the following command from the container terminal to connect to the database:

```
bash-4.2$ mysql -uroot  
Welcome to the MySQL monitor. Commands end with ; or \g.  
...output omitted...  
mysql>
```

The `mysql` command opens the MySQL database interactive prompt. Run the following command to determine the database availability:

```
mysql> show databases;  
+-----+  
| Database |  
+-----+  
| information_schema |  
| items |  
| mysql |  
| performance_schema |  
| sys |  
+-----+  
5 rows in set (0.01 sec)
```

- 3.2. Create a new table in the `items` database. Run the following command to access the database.

```
mysql> use items;  
Database changed
```

- 3.3. Create a table called `Projects` in the `items` database.

```
mysql> CREATE TABLE Projects (id int(11) NOT NULL,  
-> name varchar(255) DEFAULT NULL,  
-> code varchar(255) DEFAULT NULL,  
-> PRIMARY KEY (id));
```

```
Query OK, 0 rows affected (0.01 sec)
```

You can optionally use the `~/D0180/solutions/container-create/create_table.txt` file to copy and paste the `CREATE TABLE` MySQL statement as given above.

- 3.4. Use the `show tables` command to verify that the table was created.

```
mysql> show tables;
+-----+
| Tables_in_items |
+-----+
| Projects        |
+-----+
1 row in set (0.00 sec)
```

- 3.5. Use the `insert` command to insert a row into the table.

```
mysql> insert into Projects (id, name, code) values (1, 'DevOps', 'D0180');
Query OK, 1 row affected (0.02 sec)
```

- 3.6. Use the `select` command to verify that the project information was added to the table.

```
mysql> select * from Projects;
+---+-----+---+
| id | name      | code   |
+---+-----+---+
| 1  | DevOps    | D0180 |
+---+-----+---+
1 row in set (0.00 sec)
```

- 3.7. Exit from the MySQL prompt and the MySQL container:

```
mysql> exit
Bye
bash-4.2$ exit
exit
```

Finish

On workstation, run the `lab container-create finish` script to complete this lab.

```
[student@workstation ~]$ lab container-create finish
```

This concludes the exercise.

► LAB

CREATING CONTAINERIZED SERVICES

PERFORMANCE CHECKLIST

In this lab, you create an Apache HTTP Server container with a custom welcome page.

OUTCOMES

You should be able to start and customize a container using a container image.

BEFORE YOU BEGIN

Open a terminal on **workstation** as the **student** user and run the following command:

```
[student@workstation ~]$ lab container-review start
```

1. Start a container named **httpd-basic** in the background, and forward port 8080 to port 80 in the container. Use the **httpd** container image with the **2.4** tag.



NOTE

Use the **-p 8080:80** option with **sudo podman run** command to forward the port.

This command starts the Apache HTTP server in the background and returns to the Bash prompt.

2. Test the **httpd-basic** container.

From **workstation**, attempt to access **http://localhost:8080** using any web browser.

An "**It works!**" message is displayed, which is the **index.html** page from the Apache HTTP server container running on **workstation**.

3. Customize the **httpd-basic** container to display **Hello World** as the message. The container's message is stored in the file **/usr/local/apache2/htdocs/index.html**.

- 3.1. Start a Bash session inside the container.

- 3.2. From the Bash session, verify the **index.html** file under **/usr/local/apache2/htdocs** directory using the **ls -la** command.

- 3.3. Change the **index.html** file to contain the text **Hello World**, replacing all of the existing content.

- 3.4. Attempt to access **http://localhost:8080** again, and verify that the web page has been updated.

Evaluation

Grade your work by running the **lab container-review grade** command on your workstation machine. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab container-review grade
```

Finish

On workstation, run the **lab container-review finish** script to complete this lab.

```
[student@workstation ~]$ lab container-review finish
```

This concludes the lab.

► SOLUTION

CREATING CONTAINERIZED SERVICES

PERFORMANCE CHECKLIST

In this lab, you create an Apache HTTP Server container with a custom welcome page.

OUTCOMES

You should be able to start and customize a container using a container image.

BEFORE YOU BEGIN

Open a terminal on **workstation** as the **student** user and run the following command:

```
[student@workstation ~]$ lab container-review start
```

- Start a container named **httpd-basic** in the background, and forward port 8080 to port 80 in the container. Use the **httpd** container image with the **2.4** tag.



NOTE

Use the **-p 8080:80** option with **sudo podman run** command to forward the port.

Run the following command:

```
[student@workstation ~]$ sudo podman run -d -p 8080:80 \
> --name httpd-basic httpd:2.4
...output omitted...
Copying blob sha256:743f2d6...output omitted...
21.45 MB / 21.45 MB [=====] 1s
Copying blob sha256:c92eb69...output omitted...
155 B / 155 B [=====] 0s
Copying blob sha256:2211b05...output omitted...
9.86 MB / 9.86 MB [=====] 0s
Copying blob sha256:aed1801...output omitted...
15.78 MB / 15.78 MB [=====] 1s
Copying blob sha256:7c472a4...output omitted...
300 B / 300 B [=====] 0s
Copying config sha256:b7cc370...output omitted...
7.18 KB / 7.18 KB [=====] 0s
Writing manifest to image destination
Storing signatures
b51444e3b1d7aa94b3a4a54485d76a0a094cbfac89c287d360890a3d2779a5a
```

This command starts the Apache HTTP server in the background and returns to the Bash prompt.

2. Test the **httpd-basic** container.

From workstation, attempt to access `http://localhost:8080` using any web browser.

An "**It works!**" message is displayed, which is the `index.html` page from the Apache HTTP server container running on workstation.

```
[student@workstation ~]$ curl http://localhost:8080
<html><body><h1>It works!</h1></body></html>
```

3. Customize the **httpd-basic** container to display **Hello World** as the message. The container's message is stored in the file `/usr/local/apache2/htdocs/index.html`.

3.1. Start a Bash session inside the container.

Run the following command:

```
[student@workstation ~]$ sudo podman exec -it httpd-basic /bin/bash
root@b51444e3b1d7:/usr/local/apache2#
```

3.2. From the Bash session, verify the `index.html` file under `/usr/local/apache2/htdocs` directory using the `ls -la` command.

```
root@b51444e3b1d7:/usr/local/apache2# ls -la /usr/local/apache2/htdocs
total 4
drwxr-sr-x. 2 root      www-data 24 May  8 02:05 .
drwxr-sr-x. 1 www-data www-data 18 May  8 02:05 ..
-rw-r--r--. 1 root      src       45 Jun 11  2007 index.html
```

3.3. Change the `index.html` file to contain the text **Hello World**, replacing all of the existing content.

From the Bash session in the container, run the following command:

```
root@b51444e3b1d7:/usr/local/apache2# echo \
> "Hello World" > /usr/local/apache2/htdocs/index.html
```

3.4. Attempt to access `http://localhost:8080` again, and verify that the web page has been updated.

```
root@b51444e3b1d7:/usr/local/apache2# exit
[student@workstation ~]$ curl http://localhost:8080
Hello World
```

Evaluation

Grade your work by running the `lab container-review grade` command on your workstation machine. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab container-review grade
```

Finish

On workstation, run the `lab container-review finish` script to complete this lab.

```
[student@workstation ~]$ lab container-review finish
```

This concludes the lab.

SUMMARY

In this chapter, you learned:

- Podman allows users to search for and download images from local or remote registries.
- The **podman run** command creates and starts a container from a container image.
- Containers are executed in the background by using the **-d** flag, or interactively by using the **-it** flag.
- Some container images require environment variables that are set using the **-e** option from the **podman run** command.
- Red Hat Container Catalog assists in searching, exploring, and analyzing container images from Red Hat's official container image repository.

CHAPTER 3

MANAGING CONTAINERS

GOAL

Modify prebuilt container images to create and manage containerized services.

OBJECTIVES

- Manage a container's life cycle from creation to deletion.
- Save container application data with persistent storage.
- Describe how to use port forwarding to access a container.

SECTIONS

- Managing the Life Cycle of Containers (and Guided Exercise)
- Attaching Persistent Storage to Containers (and Guided Exercise)
- Accessing Containers (and Guided Exercise)

LAB

- Managing Containers

MANAGING THE LIFE CYCLE OF CONTAINERS

OBJECTIVES

After completing this section, students should be able to manage the life cycle of a container from creation to deletion.

CONTAINER LIFE CYCLE MANAGEMENT WITH PODMAN

In previous chapters you learned how to use Podman to create a containerized service. Now you will dive deeper into commands and strategies that you can use to manage a container's life cycle. Podman allows you not only to run containers, but also to make them run in the background, execute new processes inside them, and provide them with resources such as file system volumes or a network.

Podman, implemented by the **podman** command, provides a set of subcommands to create and manage containers. Developers use those subcommands to manage container and container images life cycle. The following figure shows a summary of the most commonly used subcommands that change container and image state.

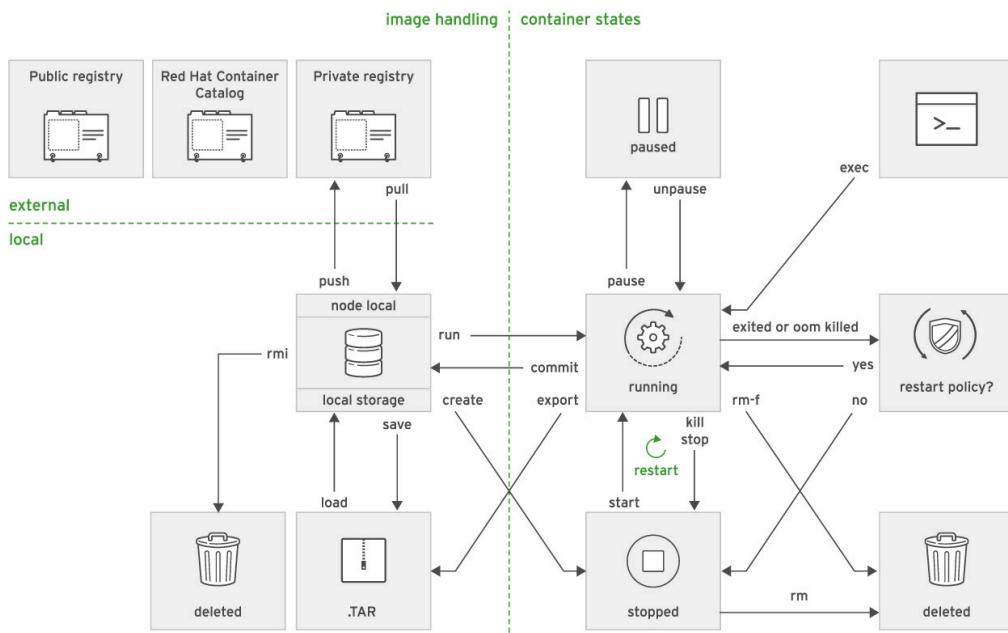


Figure 3.1: Podman managing subcommands

Podman also provides a set of useful subcommands to obtain information about running and stopped containers. You can use those 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.

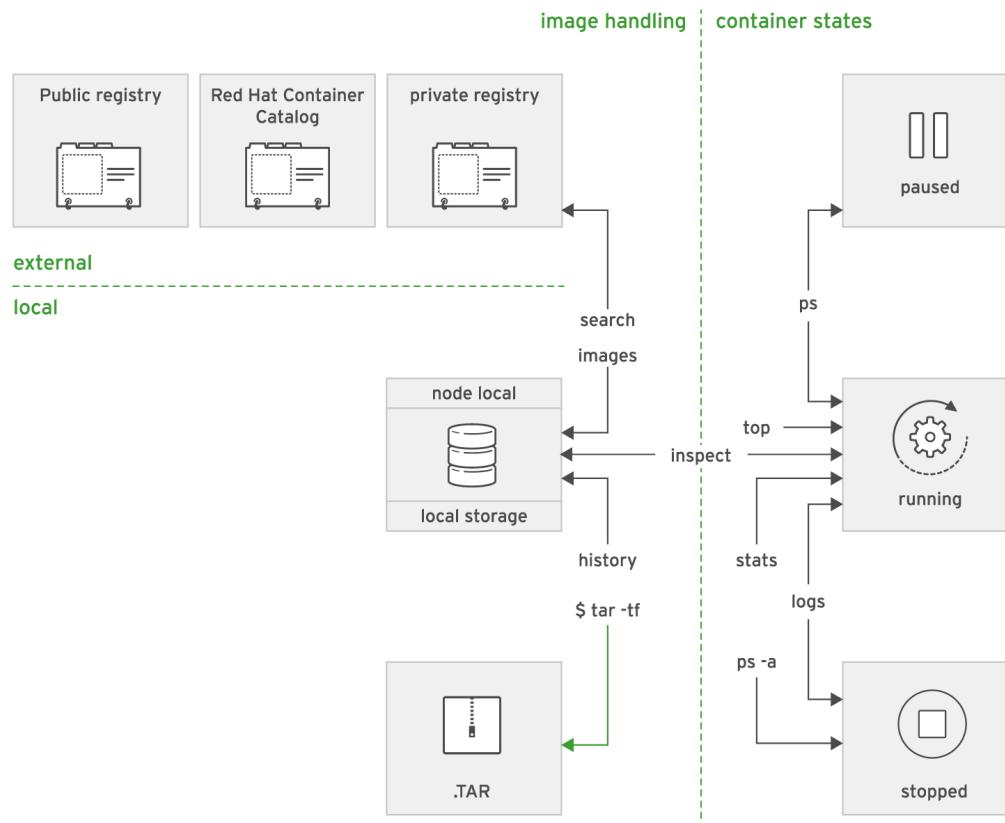


Figure 3.2: Podman query subcommands

Use these two figures as a reference while you learn about Podman subcommands in this course.

CREATING CONTAINERS

The **podman run** command creates a new container from an image and starts a process inside the new container. If the container image is not available locally, this command attempts to download the image using the configured image repository:

```
[student@workstation ~]$ sudo podman run rhscl/httpd-24-rhel7
Trying to pull regist...httpd-24-rhel7:latest...Getting image source signatures
Copying blob sha256:23113...b0be82
72.21 MB / 72.21 MB [=====] 7s
...output omitted...AH00094: Command line: 'httpd -D FOREGROUND'
^C
```

In the previous output sample, the container was started with a non interactive process (without the **-it** option) and is running in the foreground because it was not started with the **-d** option. Stopping the resulting process with **Ctrl+C (SIGINT)** therefore stops both the container process as well as the container itself.

Podman identifies containers by a unique container ID or container name. The **podman ps** command displays the container ID and names for all actively running containers:

[student@workstation ~]\$ sudo podman ps				
CONTAINER ID	IMAGE	COMMAND	...	NAMES

```
47c9aad6049①    rhscl/httpd-24-rhel7 "httpd -D FOREGROUND" ... focused_fermat②
```

- ① The container ID is unique and generated automatically.
- ② The container name can be manually specified, otherwise it is generated automatically. This name must be unique or the **run** command fails.

The **podman run** command automatically generates a unique, random ID. It also generates a random container name. To define the container name explicitly, use the **--name** option when running a container:

```
[student@workstation ~]$ sudo podman run --name my-httdp-container do180/httpd
...output omitted...AH00094: Command line: 'httpd -D FOREGROUND'
```



NOTE

The name must be unique. Podman throws an error if the name is already in use, including stopped containers.

Another important feature is the ability to run the container as a daemon process in the background. The **-d** option is responsible for running in detached mode. When using this option, Podman returns the container ID on the screen, allowing you to continue to run commands in the same terminal while the container runs in the background:

```
[student@workstation ~]$ sudo podman run --name my-httdp-container -d do180/httpd
77d4b7b8ed1fd57449163bcb0b78d205e70d2314273263ab941c0c371ad56412
```

The container image specifies the command to run to start the containerized process, known as the entry point. The **podman run** command can override this entry point by including the command after the container image:

```
[student@workstation ~]$ sudo podman run do180/httpd ls /tmp
anaconda-post.log
ks-script-1j4CXN
yum.log
```

The specified command must be executable inside the container image.



NOTE

Because a specified command appears in the previous example, the container skips the entry point for the **httpd** image. Hence, the **httpd** service does not start.

Some containers need to run as an interactive shell or process. This includes containers running processes that need user input (such as entering commands), and processes that generate output through standard output. The following example starts an interactive **bash** shell in a **do180/httpd** container:

```
[student@workstation ~]$ sudo podman run -it do180/httpd /bin/bash
bash-4.2#
```

The **-t** and **-i** options enable terminal redirection for interactive text-based programs. The **-t** option allocates a pseudo-tty (a terminal) and attaches it to the standard input of the container. The **-i** option keeps the container's standard input open, even if it was detached, so the main process can continue waiting for input.

RUNNING COMMANDS IN A CONTAINER

When a container starts, it executes the entry point command. However, it may be necessary to execute other commands to manage the running container. Some typical use case are shown below:

- Executing an interactive shell in an already running container.
- Running processes that update or display the container's files.
- Starting new background processes inside the container.

The **podman exec** command starts an additional process inside an already running container:

```
[student@workstation ~]$ sudo podman exec 7ed6e671a600 cat /etc/hostname  
7ed6e671a600
```

The previous example uses the container ID to execute the command.

Podman remembers the last container used in any command. Developers can skip writing this container's ID or name in later Podman commands by replacing the container id by the **-1** option:

```
[student@workstation ~]$ sudo podman exec my-httdp-container cat /etc/hostname  
7ed6e671a600  
[student@workstation ~]$ sudo podman exec -1 cat /etc/hostname  
7ed6e671a600
```

MANAGING CONTAINERS

Creating and starting a container is just the first step of the container's life cycle. A container's life cycle also includes stopping, restarting, or finally removing it. Users can also examine the container status and metadata for debugging, updating, or reporting purposes.

Podman provides the following commands for managing containers:

- **podman ps**: This command lists running containers:

```
[student@workstation ~]$ sudo podman ps  
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES  
77d4b7b8ed1f① do180/httpd② "httpd..."③ ...ago④ Up...⑤ 80/tcp⑥ my-htt...⑦
```

- ➊ Each container, when created, gets a **container ID**, which is a hexadecimal number. This ID looks like an image ID but is unrelated.
- ➋ Container image that was used to start the container.
- ➌ Command executed when the container started.
- ➍ Date and time the container was started.
- ➎ Total container uptime, if still running, or time since terminated.
- ➏ Ports that were exposed by the container or any port forwarding that might be configured.

- ⑦ The container name.

Podman does not discard stopped containers immediately. Podman preserves their local file systems and other states for facilitating *postmortem* analysis. Option **-a** lists all containers, including stopped ones:

```
[student@workstation ~]$ sudo podman ps -a
CONTAINER ID  IMAGE          COMMAND   CREATED    STATUS      PORTS     NAMES
4829d82fbbff  do180/httpd  "httpd..."  ...ago     Exited (0)...  my-httpd...
```

**NOTE**

While creating containers, Podman aborts if the container name is already in use, even if the container is in a “stopped” status. This option can help to avoid duplicated container names.

- **podman inspect**: This command lists metadata about a running or stopped container. The command produces **JSON** output:

```
[student@workstation ~]$ sudo podman inspect my-httpd-container
[
{
  "Id": "980e45...76c8be",
  ...output omitted...
  "NetworkSettings": {
    "Bridge": "",
    "EndpointID": "483fc9...5d801a",
    "Gateway": "172.17.42.1",
    "GlobalIPv6Address": "",
    "GlobalIPv6PrefixLen": 0,
    "HairpinMode": false,
    "IPAddress": "172.17.0.9",
  ...output omitted...
}
```

This command allows formatting of the output string using the given Go template with the **-f** option. For example, to retrieve only the IP address, use the following command:

```
[student@workstation ~]$ sudo podman inspect \
> -f '{{ .NetworkSettings.IPAddress }}' my-httpd-container
172.17.0.9
```

- **podman stop**: This command stops a running container gracefully:

```
[student@workstation ~]$ sudo podman stop my-httpd-container
77d4b7b8ed1fd57449163bcb0b78d205e70d2314273263ab941c0c371ad56412
```

Using **podman stop** is easier than finding the container start process on the host OS and killing it.

- **podman kill**: This command sends Unix signals to the main process in the container. If no signal is specified, it sends the **SIGKILL** signal, terminating the main process and the container.

```
[student@workstation ~]$ sudo podman kill my-httdp-container
77d4b7b8ed1fd57449163bcb0b78d205e70d2314273263ab941c0c371ad56412
```

You can specify the signal with the **-s** option:

```
[student@workstation ~]$ sudo podman kill -s SIGKILL my-httdp-container
77d4b7b8ed1fd57449163bcb0b78d205e70d2314273263ab941c0c371ad56412
```

Any Unix signal can be sent to the main process. Podman accepts either the signal name and number. The following table shows several useful signals:

SIGNAL	VALUE	DEFAULT ACTION	COMMENT
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from abort(3)
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at tty
SIGTTIN	21,21,26	Stop	tty input for background process

SIGNAL	VALUE	DEFAULT ACTION	COMMENT
SIGTTOU	22,22,27	Stop	tty output for background process

 **NOTE**

Term

Terminate the process.

Core

Terminate the process and generate a core dump.

Ign

Signal is ignored.

Stop

Stop the process.

- **podman restart:** This command restarts a stopped container:

```
[student@workstation ~]$ sudo podman restart my-httdp-container
77d4b7b8ed1fd57449163bcb0b78d205e70d2314273263ab941c0c371ad56412
```

The **podman restart** command creates a new container with the same container ID, reusing the stopped container state and file system.

- **podman rm:** This command deletes a container and discards its state and file system:

```
[student@workstation ~]$ sudo podman rm my-httdp-container
77d4b7b8ed1fd57449163bcb0b78d205e70d2314273263ab941c0c371ad56412
```

The **-f** option of the **rm** subcommand instructs Podman to remove the container even if not stopped. This option terminates the container forcefully and then removes it. Using **-f** option is equivalent to **podman kill** and **podman rm** commands together.

You can delete all containers at the same time. Many **podman** subcommands accept the **-a** option. This option indicates using the subcommand on all available containers or images. The following example removes all containers:

```
[student@workstation ~]$ sudo podman rm -a
5fd8e98ec7eab567eabe84943fe82e99fdfc91d12c65d99ec760d5a55b8470d6
716fd687f65b0957edac73b84b3253760e915166d3bc620c4aec8e5f4eadfe8e
86162c906b44f4cb63ba2e3386554030dcba6abedbce9e9fcad60aa9f8b2d5d4
```

Before deleting all containers, all running containers must be in a “stopped” status. You can use the following command to stop all containers:

```
[student@workstation ~]$ sudo podman stop -a
5fd8e98ec7eab567eabe84943fe82e99fdfc91d12c65d99ec760d5a55b8470d6
716fd687f65b0957edac73b84b3253760e915166d3bc620c4aec8e5f4eadfe8e
86162c906b44f4cb63ba2e3386554030dcba6abedbce9e9fcad60aa9f8b2d5d4
```

**NOTE**

The **inspect**, **stop**, **kill**, **restart**, and **rm** subcommands can use the container ID instead of the container name.

**REFERENCES****Unix Posix Signals man page**

<http://man7.org/linux/man-pages/man7/signal.7.html>

► GUIDED EXERCISE

MANAGING A MYSQL CONTAINER

In this exercise, you will create and manage a MySQL® database container.

OUTCOMES

You should be able to create and manage a MySQL database container.

BEFORE YOU BEGIN

Make sure that **workstation** has the **podman** command available and is correctly set up by running the following command from a terminal window:

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

- 1. Open a terminal window from the **workstation** VM (Applications → Utilities → Terminal) and run the following command:

```
[student@workstation ~]$ sudo podman run --name mysql-db rhscl/mysql-57-rhel7
Trying to pull ...output omitted...
...output omitted...
Writing manifest to image destination
Storing signatures
You must either specify the following environment variables:
  MYSQL_USER (regex: '^[a-zA-Z0-9_-]+$')
  MYSQL_PASSWORD (regex: '^([a-zA-Z0-9_-!@#$%^&*()=-<>, .?;:|]+)$')
  MYSQL_DATABASE (regex: '^([a-zA-Z0-9_-]+)$')
Or the following environment variable:
  MYSQL_ROOT_PASSWORD (regex: '^([a-zA-Z0-9_-!@#$%^&*()=-<>, .?;:|]+)$')
Or both.
Optional Settings:
...output omitted...

For more information, see https://github.com/sclorg/mysql-container
```

This command downloads the MySQL database container image and tries to start it, but it does not start. This is because the image requires several environment variables to be provided.



NOTE

If you try to run the container as a daemon (**-d**), the error message about the required variables is not displayed. However, this message is included as part of the container logs, which can be viewed using the following command:

```
[student@workstation ~]$ sudo podman logs mysql-db
```

- 2. Start the container again, providing the required variables. Give it a name of **mysql**. Specify each variable using the **-e** parameter.

**NOTE**

Make sure you start the new container with the correct name.

```
[student@workstation ~]$ sudo podman run --name mysql \
> -d -e MYSQL_USER=user1 -e MYSQL_PASSWORD=mypa55 \
> -e MYSQL_DATABASE=items -e MYSQL_ROOT_PASSWORD=r00tpa55 \
> rhsc1/mysql-57-rhel7
```

The output is the container ID for the **mysql** container. Below is an example of the output.

```
a49dba9ff17f2b5876001725b581fdd331c9ab8b9eda21cc2a2899c23f078509
```

- 3. Verify that the container was started correctly. Run the following command:

```
[student@workstation ~]$ sudo podman ps
CONTAINER ID ...output omitted... STATUS PORTS NAMES
a49dba9ff17f ...output omitted... Up About a minute ago mysql
```

The container ID shown is a shortened from the container ID displayed in the previous command.

- 4. Inspect the container metadata to obtain the IP address from the MySQL database:

```
[student@workstation ~]$ sudo podman inspect \
> -f '{{ .NetworkSettings.IPAddress }}' mysql
10.88.0.6
```

The IP address of your container may differ from the one shown above (**10.88.0.6**).

**NOTE**

You can get other important information with the **podman inspect** command. For example, if you forget the root password, it is available in the **Env** section.

- 5. Create the **Projects** table:

You are connected to the **items** database. Create a new table by using one of the following:

- Connect to the MySQL database and type the **CREATE TABLE** command.
 1. Connect to the MySQL database from the host. Change the IP address in the command below to match the IP address of your **mysql** container:

```
[student@workstation ~]$ mysql -uuser1 -h 10.88.0.6 -p items
```

Enter password:

Use **mypa55** as the password.

```
Welcome to the MariaDB monitor. Commands end with ; or \g.
Your MySQL connection id is 2
Server version: 5.7.16 MySQL Community Server (GPL)

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MySQL [items]>
```

2. Type or copy the following SQL commands:

```
MySQL [items]> CREATE TABLE Projects (id int(11) NOT NULL, \
-> name varchar(255) DEFAULT NULL, code varchar(255) DEFAULT NULL, \
-> PRIMARY KEY (id));
Query OK, 0 rows affected (0.05 sec)
```

3. Insert a row into the table:

```
MySQL [items]> insert into Projects (id, name, code) values (1,'DevOps','D0180');
Query OK, 1 row affected (0.09 sec)
```

4. Exit from the MySQL prompt:

```
MySQL [items]> exit
```

- Alternatively you can create the database and insert the row by using the provided file:

```
[student@workstation ~]$ mysql -uuser1 -h 10.88.0.6 \
> -pmypa55 items < D0180/labs/manage-lifecycle/db.sql
```

- 6. Create another container using the same container image as the previous container executing the **/bin/bash** shell:

```
[student@workstation ~]$ sudo podman run --name mysql-2 \
> -it rhscl/mysql-57-rhel7 /bin/bash
bash-4.2$
```

- 7. Try to connect to the MySQL database in the new container:

```
bash-4.2$ mysql -uroot
```

The following error is displayed:

```
ERROR 2002 (HY000): Can't connect to local MySQL ...output omitted...
```

The reason for this error is that the MySQL database server is not running, because when we created the new container we changed the entry point responsible for starting the database to **/bin/bash**.

- 8. Exit from the **bash** shell:

```
bash-4.2$ exit
```

- 9. Verify that the container **mysql-2** is not running:

```
[student@workstation ~]$ sudo podman ps -a \
> --format="table {{.ID}} {{.Names}} {{.Status}}"
CONTAINER ID NAMES      STATUS
2871e392af02 mysql-2   Exited (1) 19 seconds ago
a49dba9ff17f mysql     Up 10 minutes ago
c053c7e09c21 mysql-db  Exited (1) 44 minutes ago
```

- 10. Execute commands in the detached container. Use **mypa55** for the password:

```
[student@workstation ~]$ sudo podman exec mysql /bin/bash \
> -c 'mysql -uuser1 -p -e "select * from items.Projects;"'
Enter password: mypa55
id name code
1 DevOps D0180
```

The previous command runs a bash interpreter in the **mysql** container. Then, the command instructs bash to run **mysql** interpreter, that receives the SQL query to fetch data from the database.

Finish

On workstation, run the **lab manage-lifecycle finish** script to complete this exercise.

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

This concludes the exercise.

ATTACHING PERSISTENT STORAGE TO CONTAINERS

OBJECTIVES

After completing this section, students should be able to:

- Save application data across container restarts through the use of persistent storage.
- Configure host directories for use as container volumes.
- Mount a volume inside the container.

PREPARING PERMANENT STORAGE LOCATIONS

Container storage is said to be *ephemeral*, meaning its contents are not preserved after the container is removed. Containerized applications work on the assumption that they always start with empty storage, and this makes creating and destroying containers relatively inexpensive operations.

Previously in this course, container images were characterized as *immutable* and *layered*, meaning that they are never changed, but rather composed of layers that add or override the contents of layers below.

A running container gets a new layer over its base container image, and this layer is the *container storage*. At first, this layer is the only read/write storage available for the container, and it is used to create working files, temporary files, and log files. Those files are considered volatile. An application does not stop working if they are lost. The container storage layer is exclusive to the running container, so if another container is created from the same base image, it gets another read/write layer. This ensures the each container's resources are isolated from other similar containers.

Ephemeral container storage is *not* sufficient for applications that need to keep data over restarts, such as databases. To support such applications, the administrator must provide a container with persistent storage.

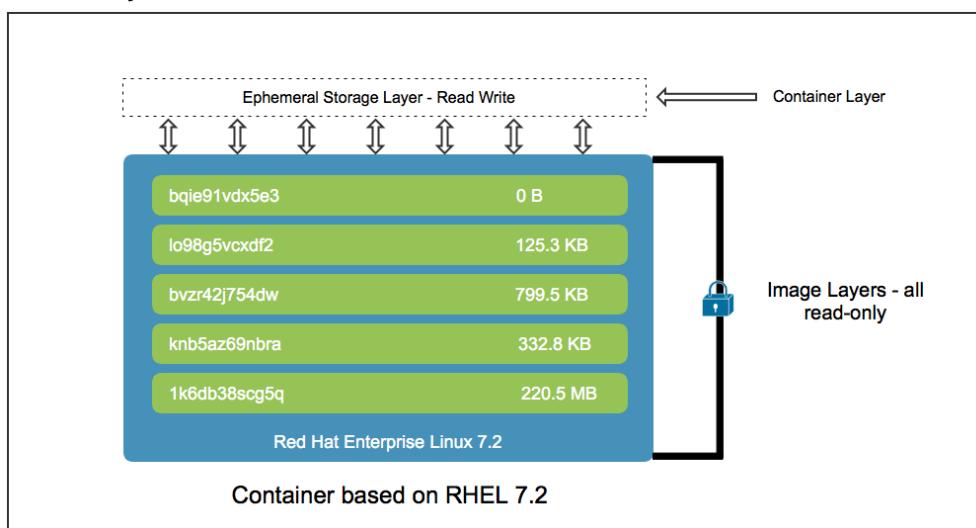


Figure 3.3: Container layers

Containerized applications should not try to use the container storage to store persistent data, because they cannot control how long its contents will be preserved. Even if it were possible to keep container storage indefinitely, the layered file system does not perform well for intensive I/O workloads and would not be adequate for most applications requiring persistent storage.

Reclaiming Storage

Podman keeps old stopped container storage available to be used by troubleshooting operations, such as reviewing failed container logs for error messages.

If the administrator needs to reclaim old container storage, the container can then be deleted using **podman rm *container_id***. This command also deletes the container storage. The stopped container IDs can be found using **podman ps -a** command.

Preparing the Host Directory

Podman can mount host directories inside a running container. The containerized application sees these host directories as part of the container storage, much like regular applications see a remote network volume as if it were part of the host file system. But these host directories' contents are not reclaimed after the container is stopped, and they can be mounted to new containers whenever needed.

For example, a database container can use a host directory to store database files. If this database container fails, Podman can create a new container using the same host directory, keeping the database data available to client applications. To the database container, it does not matter where this host directory is stored from the host point of view; it could be anything from a local hard disk partition to a remote networked file system.

A container runs as a host operating system process, under a host operating system user and group ID, so the host directory needs to be configured with ownership and permissions allowing access to the container. In RHEL, the host directory also needs to be configured with the appropriate SELinux context, which is **container_file_t**. Podman uses the **container_file_t** SELinux context to restrict which files of the host system the container is allowed to access. This avoids information leakage between the host system and the applications running inside containers.

One way to set up the host directory is described below:

1. Create a directory with owner and group **root**:

```
[student@workstation ~]$ sudo mkdir /var/dbfiles
```

2. The user running processes in the container must be capable of writing files to the directory. If the host machine does not have exactly the same user defined, the permission should be defined with the numeric user ID (UID) from the container. In the case of the Red Hat-provided MySQL service, the UID is 27:

```
[student@workstation ~]$ sudo chown -R 27:27 /var/dbfiles
```

3. Apply the **container_file_t** context to the directory (and all subdirectories) to allow containers access to all of its contents.

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

4. Apply the SELinux container policy that you set up in the first step to the newly created directory:

```
[student@workstation ~]$ restorecon -Rv /var/dbfiles
```

The host directory must be configured *before* starting the container that uses the directory.

Mounting a Volume

After creating and configuring the host directory, the next step is to mount this directory to a container. To bind mount a host directory to a container, add the **-v** option to the **podman run** command, specifying the host directory path and the container storage path, separated by a colon (:).

For example, to use the **/var/dbfiles** host directory for MySQL server database files, which are expected to be under **/var/lib/mysql** inside a MySQL container image named **mysql**, use the following command:

```
[student@workstation ~]$ sudo podman run -v /var/dbfiles:/var/lib/mysql mysql
```

In the previous command, if the **/var/lib/mysql** already exists inside the **mysql** container image, the **/var/dbfiles** mount overlays but does not remove the content from the container image. If the mount is removed, the original content is accessible again.

► GUIDED EXERCISE

PERSISTING A MYSQL DATABASE

In this exercise, you will create a container that stores the MySQL database data into a host directory.

OUTCOMES

You should be able to deploy container with a persistent database.

BEFORE YOU BEGIN

The workstation should not have any container images running. Run the following command on workstation:

```
[student@workstation ~]$ lab manage-storage start
```

- ▶ 1. Open a terminal window on workstation (Applications → System Tools → Terminal).
- ▶ 2. Create the **/var/local/mysql** directory with the correct SELinux context and permissions.
 - 2.1. Create the **/var/local/mysql** directory.

```
[student@workstation ~]$ sudo mkdir -pv /var/local/mysql  
mkdir: created directory '/var/local/mysql'
```

- 2.2. Add the appropriate SELinux context for the **/var/local/mysql** directory and its contents.

```
[student@workstation ~]$ sudo semanage fcontext -a \  
> -t container_file_t '/var/local/mysql(/.*)?'
```

- 2.3. Apply the SELinux policy to the newly created directory.

```
[student@workstation ~]$ sudo restorecon -R /var/local/mysql
```

- 2.4. Verify that the SELinux context type for the **/var/local/mysql** directory is **container_file_t**.

```
[student@workstation ~]$ ls -dz /var/local/mysql  
drwxr-xr-x. root root unconfined_u:object_r:container_file_t:s0 /var/local/mysql
```

- 2.5. Change the owner of the **/var/local/mysql** directory to the **mysql** user and **mysql** group:

```
[student@workstation ~]$ sudo chown -Rv 27:27 /var/local/mysql
changed ownership of '/var/local/mysql' from root:root to 27:27
```

**NOTE**

The user running processes in the container must be capable of writing files to the directory. If the host machine does not have exactly the same user defined, the permission should be defined with the numeric user ID (UID) from the container. For the MySQL service provided by Red Hat, the UID is 27.

► 3. Create a MySQL container instance with persistent storage.

3.1. Pull the MySQL container image from the internal registry:

```
[student@workstation ~]$ sudo podman pull rhsc1/mysql-57-rhel7
Trying to pull ...output omitted...rhsc1/mysql-57-rhel7...output omitted...
...output omitted...
Writing manifest to image destination
Storing signatures
4ae3a3f4f409a8912cab9fbf71d3564d011ed2e68f926d50f88f2a3a72c809c5
```

3.2. Create a new container specifying the mount point to store the MySQL database data:

```
[student@workstation ~]$ sudo podman run --name persist-db \
> -d -v /var/local/mysql:/var/lib/mysql/data \
> -e MYSQL_USER=user1 -e MYSQL_PASSWORD=mypa55 \
> -e MYSQL_DATABASE=items -e MYSQL_ROOT_PASSWORD=r00tpa55 \
> rhsc1/mysql-57-rhel7
```

This command mounts the host **/var/local/mysql** directory in the container **/var/lib/mysql/data** directory. The **/var/lib/mysql/data** is the directory where the MySQL database stores the data.

3.3. Verify that the container was started correctly. Run the following command:

```
[student@workstation ~]$ sudo podman ps \
> --format="table {{.ID}} {{.Names}} {{.Status}}"
CONTAINER ID NAMES STATUS
ce637c4da16 persist-db Up 3 minutes ago
```

► 4. Verify that the **/var/local/mysql** directory contains an **items** directory:

```
[student@workstation ~]$ ls -l /var/local/mysql
total 41032
-rw-r----. 1 27 27 56 Jun 13 07:50 auto.cnf
-rw-----. 1 27 27 1676 Jun 13 07:50 ca-key.pem
-rw-r--r--. 1 27 27 1075 Jun 13 07:50 ca.pem
-rw-r--r--. 1 27 27 1079 Jun 13 07:50 client-cert.pem
-rw-----. 1 27 27 1680 Jun 13 07:50 client-key.pem
```

```
-rw-r----- 1 27 27 2 Jun 13 07:51 e89494e64d5b.pid
-rw-r----- 1 27 27 349 Jun 13 07:51 ib_buffer_pool
-rw-r----- 1 27 27 12582912 Jun 13 07:51 ibdata1
-rw-r----- 1 27 27 8388608 Jun 13 07:51 ib_logfile0
-rw-r----- 1 27 27 8388608 Jun 13 07:50 ib_logfile1
-rw-r----- 1 27 27 12582912 Jun 13 07:51 ibtmp1
drwxr-x--- 2 27 27 20 Jun 13 07:50 items
drwxr-x--- 2 27 27 4096 Jun 13 07:50 mysql
drwxr-x--- 2 27 27 8192 Jun 13 07:50 performance_schema
-rw----- 1 27 27 1680 Jun 13 07:50 private_key.pem
-rw-r--r-- 1 27 27 452 Jun 13 07:50 public_key.pem
-rw-r--r-- 1 27 27 1079 Jun 13 07:50 server-cert.pem
-rw----- 1 27 27 1676 Jun 13 07:50 server-key.pem
drwxr-x--- 2 27 27 8192 Jun 13 07:50 sys
```

This directory stores data related to the `items` database that was created by this container. If this directory is not available, the mount point was not defined correctly in the container creation.

Finish

On workstation, run the `lab manage-storage finish` script to complete this lab.

```
[student@workstation ~]$ lab manage-storage finish
```

This concludes the exercise.

ACCESSING CONTAINERS

OBJECTIVES

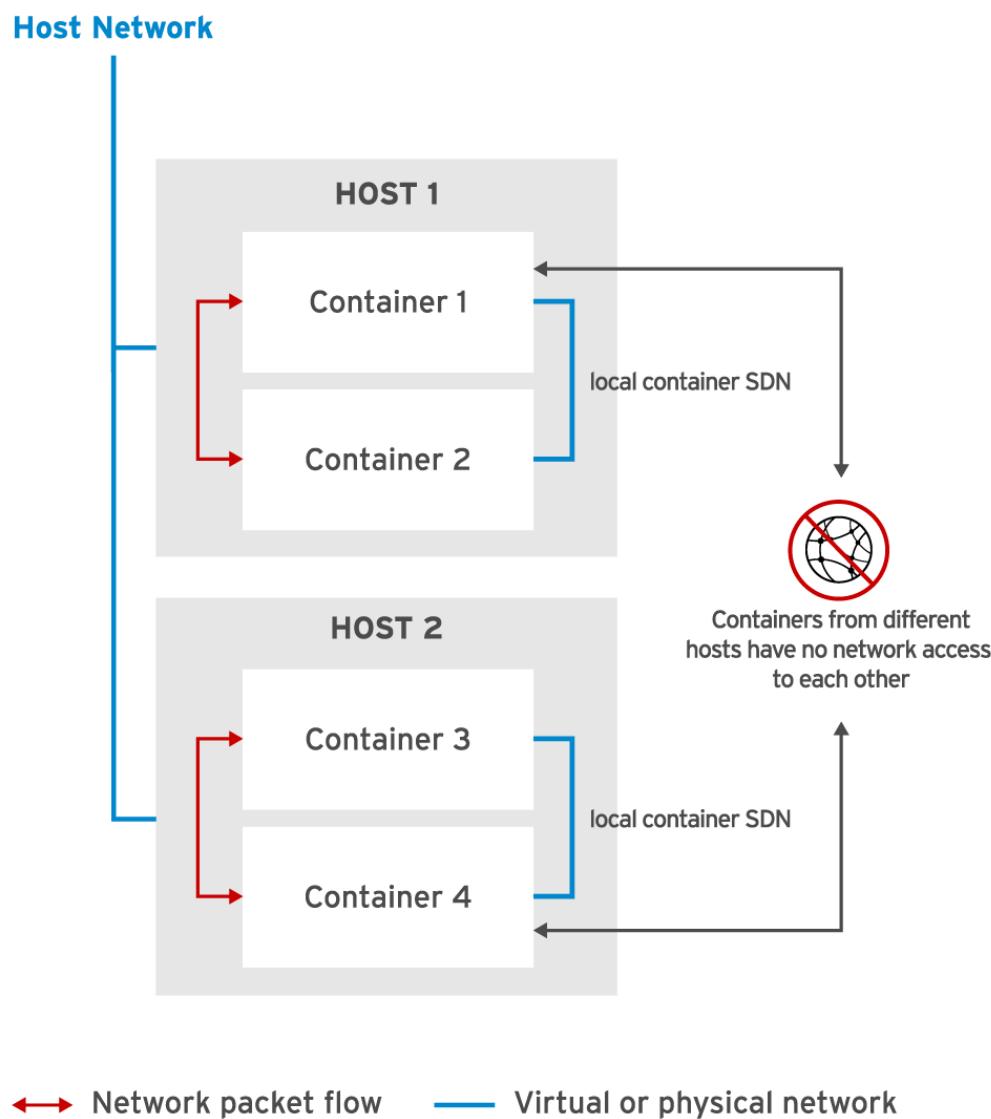
After completing this section, students should be able to:

- Describe the basics of networking with containers.
- Remotely connect to services within a container.

INTRODUCING NETWORKING WITH CONTAINERS

The Cloud Native Computing Foundation (CNCF) sponsors the *Container Networking Interface (CNI)* open source project. The CNI project aims to standardize the network interface for containers in cloud native environments, such as Kubernetes and Red Hat OpenShift Container Platform.

Podman uses the CNI project to implement a *software-defined network (SDN)* for containers on each host. Podman attaches each container to a virtual bridge and assigns each container a private IP address. The configuration file that specifies CNI settings for Podman is **/etc/cni/net.d/87-podman-bridge.conflist**.

**Figure 3.4: Basic Linux container networking**

When Podman creates containers on the same host, it assigns each container a unique IP address and connects them all to the same software-defined network. These containers can communicate freely with each other by IP address.

Containers created with Podman running on different hosts belong to different software-defined networks. Each SDN is isolated, which prevents a container in one network from communicating with a container in a different network. Because of network isolation, a container in one SDN can have the same IP address as a container in a different SDN.

It is also important to note that, by default, all container networks are hidden from the host network. That is, containers typically can access the host network, but without explicit configuration, there is no access back into the container network.

MAPPING NETWORK PORTS

Accessing a container from the host network can be a challenge. A container is assigned an IP address from a pool of available addresses. When a container is destroyed, the container's address is released back to the pool of available addresses. Another problem is that the container software-defined network is only accessible from the container host.

To solve these problems, define port forwarding rules to allow external access to a container service. Use the `-p [<IP address>:][:<host port>:<container port>]` option with the `podman run` command to create an externally accessible container. Consider the following example:

```
[student@workstation ~]$ sudo podman run -d --name apache1 -p 8080:80 httpd:2.4
```

The value **8080:80** specifies that any requests to port 8080 on the host are forwarded to port 80 within the container.

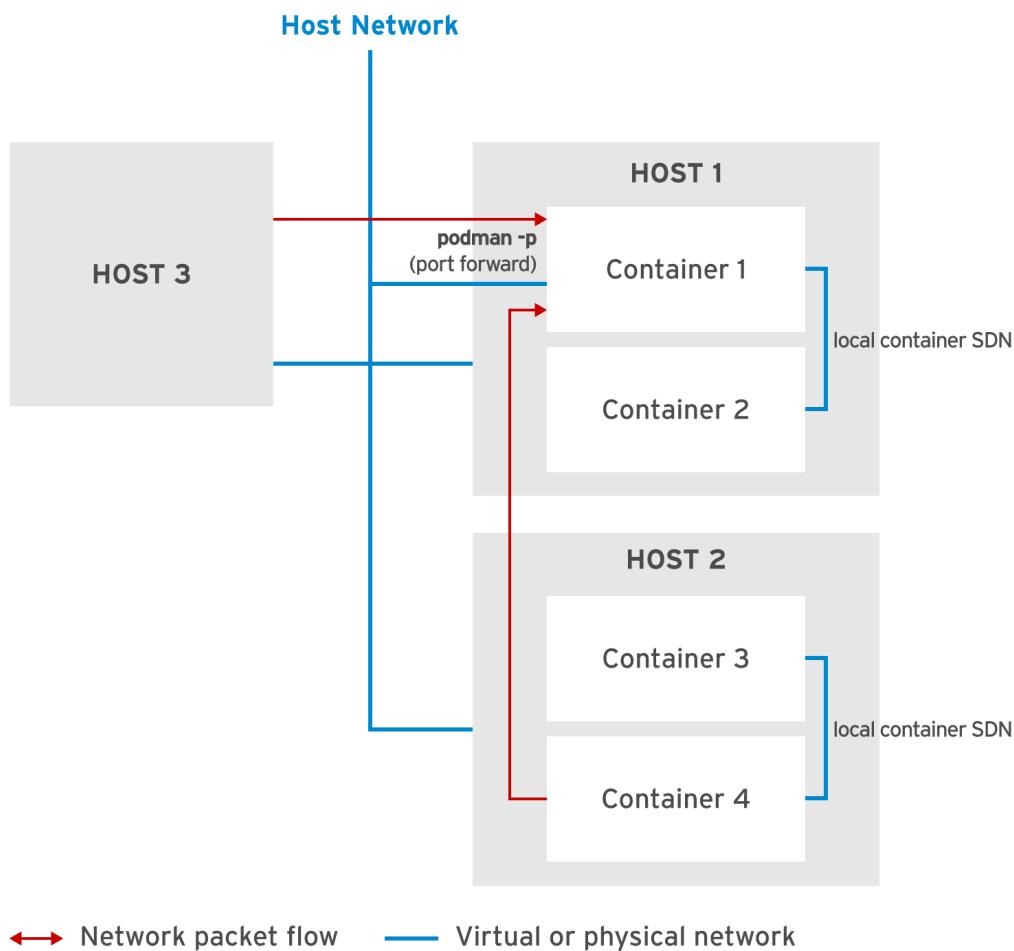


Figure 3.5: Allowing external accesses to Linux containers

You can also use the `-p` option to only forward requests to a container if those requests originate from a specified IP address:

```
[student@workstation ~]$ sudo podman run -d --name apache2 \
```

```
> -p 127.0.0.1:8081:80 httpd:2.4
```

The example above limits external access to the **apache2** container to requests from `localhost` to host port 8081. These requests are forwarded to port 80 in the **apache2** container.

If a port is not specified for the host port, Podman assigns a random available host port for the container:

```
[student@workstation ~]$ sudo podman run -d --name apache3 -p 127.0.0.1::80  
httpd:2.4
```

To see the port assigned by Podman, use the **podman port <container name>** command:

```
[student@workstation ~]$ sudo podman port apache3  
80/tcp -> 127.0.0.1:35134  
[student@workstation ~]$ curl 127.0.0.1:35134  
<html><body><h1>It works!</h1></body></html>
```

If only a container port is specified with the **-p** option, a random available host port is assigned to container. Requests to this assigned host port from any IP address are forwarded to the container port.

```
[student@workstation ~]$ sudo podman run -d --name apache4 -p 80 httpd:2.4  
[student@workstation ~]$ sudo podman port apache4  
80/tcp -> 0.0.0.0:37068
```

In the above example, any routable request to host port 37068 is forwarded to the port 80 in the container.



REFERENCES

Container Network Interface - networking for Linux containers

<https://github.com/containernetworking/cni>

Cloud Native Computing Foundation

<https://www.cncf.io/>

► GUIDED EXERCISE

LOADING THE DATABASE

In this exercise, you will create a MySQL database container. You will forward ports from the container to the host in order to load the database with a SQL script.

OUTCOMES

You should be able to deploy a database container and load a SQL script.

BEFORE YOU BEGIN

Open a terminal on `workstation` as the `student` user and run the following command:

```
[student@workstation ~]$ lab manage-networking start
```

This ensures the `/var/local/mysql` directory exists and is configured with the correct permissions to enable persistent storage for the MySQL container.

- 1. Create a MySQL container instance with persistent storage and port forwarding:

```
[student@workstation ~]$ sudo podman run --name mysqlDb-port \
> -d -v /var/local/mysql:/var/lib/mysql/data -p 13306:3306 \
> -e MYSQL_USER=user1 -e MYSQL_PASSWORD=mpa55 \
> -e MYSQL_DATABASE=items -e MYSQL_ROOT_PASSWORD=r00tpa55 \
> rhel7/mysql-5.7-rhel7
Trying to pull ...output omitted...
Copying blob sha256:e373541...output omitted...
  69.66 MB / 69.66 MB [=====] 8s
Copying blob sha256:c5d2e94...output omitted...
  1.20 KB / 1.20 KB [=====] 0s
Copying blob sha256:b3949ae...output omitted...
  62.03 MB / 62.03 MB [=====] 8s
Writing manifest to image destination
Storing signatures
9941da2936a5a15e1cab15e2790cf0b4e9ae4db07a18dc6b52a23621d5917d89
```

The last line of your output will differ from that shown above, as well as the time needed to download each image layer.

The `-p` option configures port forwarding. In this case, every connection on the host IP address to the port 13306 is forwarded to the container port 3306.



NOTE

The `/var/local/mysql` directory is created and configured by the start script to have the permissions required by the containerized database.

- 2. Run the following command:

```
[student@workstation ~]$ sudo podman ps  
CONTAINER ID ...output omitted... PORTS NAMES  
9941da2936a5 ...output omitted... 0.0.0.0:13306->3306/tcp mysqldb-port
```

Inspect the **PORTS** column to see the port forwarding rule.

- 3. Load the database:

```
[student@workstation ~]$ mysql -uuser1 -h 127.0.0.1 -pmypa55 \  
> -P13306 items < /home/student/D0180/labs/manage-networking/db.sql
```

If there are no errors, the above command does not return any output.

- 4. Verify that the database was successfully loaded by one of the three following alternatives:

- Using the port forwarded from the host and the local database client:

```
[student@workstation ~]$ mysql -uuser1 -h 127.0.0.1 -pmypa55 \  
> -P13306 items -e "SELECT * FROM Item"  
+-----+-----+  
| id | description | done |  
+-----+-----+  
| 1 | Pick up newspaper | 0 |  
| 2 | Buy groceries | 1 |  
+-----+-----+
```

- Run an interactive terminal and the database client from inside the container:

1. Open a Bash shell inside the container.

```
[student@workstation ~]$ sudo podman exec -it mysqldb-port /bin/bash  
bash-4.2$
```

2. Verify that the **mysql** command is installed in the container:

```
bash-4.2$ which mysql  
/opt/rh/rh-mysql57/root/usr/bin/mysql
```

3. Verify that the database contains data:

```
bash-4.2$ mysql -uroot items -e "SELECT * FROM Item"  
+-----+-----+  
| id | description | done |  
+-----+-----+  
| 1 | Pick up newspaper | 0 |  
| 2 | Buy groceries | 1 |  
+-----+-----+
```

4. Exit from the Bash shell inside the container:

```
bash-4.2$ exit  
[student@workstation ~]$
```

- Inject the **mysql** process inside the container.

```
[student@workstation ~]$ sudo podman exec -it mysql5db-port \  
> /opt/rh/rh-mysql57/root/usr/bin/mysql -uroot items -e "SELECT * FROM Item"  
+-----+  
| id | description | done |  
+-----+  
| 1 | Pick up newspaper | 0 |  
| 2 | Buy groceries | 1 |  
+-----+
```

**NOTE**

The **mysql** command is not in the PATH variable and, for this reason, you must use an absolute path.

Finish

On workstation, run the **lab manage-networking finish** script to complete this lab.

```
[student@workstation ~]$ lab manage-networking finish
```

This concludes the exercise.

► LAB

MANAGING CONTAINERS

PERFORMANCE CHECKLIST

In this lab, you will deploy a container that saves the MySQL database data into a host folder, loads the database, and manages the container.

OUTCOMES

You should be able to deploy and manage a persistent database using a shared volume. You should also be able to start a second database using the same shared volume and observe that the data is consistent between the two containers because they are using the same directory on the host to store the MySQL data.

BEFORE YOU BEGIN

Open a terminal on **workstation** as the **student** user and run the following command:

```
[student@workstation ~]$ lab manage-review start
```

1. Create the **/var/local/mysql** directory with the correct SELinux context and permissions.
 - 1.1. Create the **/var/local/mysql** directory.
 - 1.2. Add the appropriate SELinux context for the **/var/local/mysql** directory and its contents. With the correct context, you can mount this directory in a running container.
 - 1.3. Apply the SELinux policy to the newly created directory.
 - 1.4. Change the owner of the **/var/local/mysql** directory to match the **mysql** user and **mysql** group for the **rhscl/mysql-57-rhel7** container image:
2. Deploy a MySQL container instance using the following characteristics:
 - **Name:** **mysql-1**
 - **Run as daemon:** yes
 - **Volume:** from **/var/local/mysql** host folder to **/var/lib/mysql/data** container folder
 - **Container image:** **rhscl/mysql-57-rhel7**
 - **Port forward:** no
 - **Environment variables:**
 - **MYSQL_USER:** **user1**
 - **MYSQL_PASSWORD:** **mypa55**

- MYSQL_DATABASE: **items**
 - MYSQL_ROOT_PASSWORD: **r00tpa55**
3. Load the **items** database using the **/home/student/D0180/labs/manage-review/db.sql** script.
- 3.1. Get the container IP address.
 - 3.2. Load the database using the SQL commands in **/home/student/D0180/labs/manage-review/db.sql**. Use the IP address you find in the previous step as the database server's host IP.

**NOTE**

You can import all the commands in the above file using the less-than operator (<) after the **mysql** command, instead of typing them. Also, you need to add the **-h CONTAINER_IP** parameter to the **mysql** command to connect to the correct container.

- 3.3. Use an SQL **SELECT** statement to output all rows of the Item table to verify that the Items database is loaded.

**NOTE**

You can add the **-e SQL** parameter to the **mysql** command to execute an SQL instruction.

4. Stop the container gracefully.

**IMPORTANT**

This step is very important because a new container will be created sharing the same volume for database data. Having two containers using the same volume can corrupt the database. Do not restart the **mysql-1** container.

5. Create a new container with the following characteristics:
 - **Name:** mysql-2
 - **Run as a daemon:** yes
 - **Volume:** from `/var/local/mysql` host folder to `/var/lib/mysql/data` container folder
 - **Container image:** rhsc1/mysql-57-rhel7
 - **Port forward:** yes, from host port 13306 to container port 3306
 - **Environment variables:**
 - `MYSQL_USER: user1`
 - `MYSQL_PASSWORD: mypa55`
 - `MYSQL_DATABASE: items`
 - `MYSQL_ROOT_PASSWORD: r00tpa55`
6. Save the list of all containers (including stopped ones) to the `/tmp/my-containers` file.
7. Access the Bash shell inside the container and verify that the `items` database and the `Item` table are still available. Confirm also that the table contains data.
 - 7.1. Access the Bash shell inside the container.
 - 7.2. Connect to the MySQL server.
 - 7.3. List all databases and confirm that the `items` database is available.
 - 7.4. List all tables from the `items` database and verify that the `Item` table is available.
 - 7.5. View the data from the table.
 - 7.6. Exit from the MySQL client and from the container shell.
8. Using port forwarding, insert a new row into the `Item` table. The row should have a `description` value of **Finished lab**, and a `done` value of **1**.
 - 8.1. Connect to the MySQL database.
 - 8.2. Insert the new row.
 - 8.3. Exit from the MySQL client.
9. Because the first container is not required any more, remove it to release resources.

Evaluation

Grade your work by running the `lab manage-review grade` command from your workstation machine. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab manage-review grade
```

Finish

On workstation, run the **lab manage-review finish** command to complete this lab.

```
[student@workstation ~]$ lab manage-review finish
```

This concludes the lab.

► SOLUTION

MANAGING CONTAINERS

PERFORMANCE CHECKLIST

In this lab, you will deploy a container that saves the MySQL database data into a host folder, loads the database, and manages the container.

OUTCOMES

You should be able to deploy and manage a persistent database using a shared volume. You should also be able to start a second database using the same shared volume and observe that the data is consistent between the two containers because they are using the same directory on the host to store the MySQL data.

BEFORE YOU BEGIN

Open a terminal on `workstation` as the `student` user and run the following command:

```
[student@workstation ~]$ lab manage-review start
```

1. Create the `/var/local/mysql` directory with the correct SELinux context and permissions.
 - 1.1. Create the `/var/local/mysql` directory.

```
[student@workstation ~]$ sudo mkdir -pv /var/local/mysql
mkdir: created directory '/var/local/mysql'
```

- 1.2. Add the appropriate SELinux context for the `/var/local/mysql` directory and its contents. With the correct context, you can mount this directory in a running container.

```
[student@workstation ~]$ sudo semanage fcontext -a \
> -t container_file_t '/var/local/mysql(/.*)?'
```

- 1.3. Apply the SELinux policy to the newly created directory.

```
[student@workstation ~]$ sudo restorecon -R /var/local/mysql
```

- 1.4. Change the owner of the `/var/local/mysql` directory to match the `mysql` user and `mysql` group for the `rhsc1/mysql-57-rhel7` container image:

```
[student@workstation ~]$ sudo chown -Rv 27:27 /var/local/mysql
changed ownership of '/var/local/mysql' from root:root to 27:27
```

2. Deploy a MySQL container instance using the following characteristics:

- **Name:** `mysql-1`

- **Run as daemon:** yes
- **Volume:** from `/var/local/mysql` host folder to `/var/lib/mysql/data` container folder
- **Container image:** `rhscl/mysql-57-rhel7`
- **Port forward:** no
- **Environment variables:**
 - `MYSQL_USER: user1`
 - `MYSQL_PASSWORD: mypa55`
 - `MYSQL_DATABASE: items`
 - `MYSQL_ROOT_PASSWORD: r00tpa55`

2.1. Create and start the container.

```
[student@workstation ~]$ sudo podman run --name mysql-1 \
> -d -v /var/local/mysql:/var/lib/mysql/data \
> -e MYSQL_USER=user1 -e MYSQL_PASSWORD=mypa55 \
> -e MYSQL_DATABASE=items -e MYSQL_ROOT_PASSWORD=r00tpa55 \
> rhscl/mysql-57-rhel7
Trying to pull ...output omitted...
...output omitted...
Writing manifest to image destination
Storing signatures
616azfaa55x866e7eb18b1fd0423a5461d8f24c147b1ad8668b76e6167587cd
```

2.2. Verify that the container was started correctly.

```
[student@workstation ~]$ sudo podman ps
CONTAINER ID      ...output omitted...    NAMES
616azfaa55x8      ...output omitted...    mysql-1
```

3. Load the `items` database using the `/home/student/D0180/labs/manage-review/db.sql` script.

3.1. Get the container IP address.

```
[student@workstation ~]$ sudo podman inspect \
> -f '{{ .NetworkSettings.IPAddress }}' mysql-1
```

10.88.0.6

- 3.2. Load the database using the SQL commands in `/home/student/D0180/labs/manage-review/db.sql`. Use the IP address you find in the previous step as the database server's host IP.

**NOTE**

You can import all the commands in the above file using the less-than operator (<) after the `mysql` command, instead of typing them. Also, you need to add the `-h CONTAINER_IP` parameter to the `mysql` command to connect to the correct container.

```
[student@workstation ~]$ mysql -uuser1 -h CONTAINER_IP \
> -pmypa55 items < /home/student/D0180/labs/manage-review/db.sql
```

- 3.3. Use an SQL **SELECT** statement to output all rows of the Item table to verify that the Items database is loaded.

**NOTE**

You can add the `-e SQL` parameter to the `mysql` command to execute an SQL instruction.

```
[student@workstation ~]$ mysql -uuser1 -h CONTAINER_IP -pmypa55 items \
> -e "SELECT * FROM Item"
+---+-----+----+
| id | description | done |
+---+-----+----+
| 1 | Pick up newspaper | 0 |
| 2 | Buy groceries | 1 |
+---+-----+----+
```

4. Stop the container gracefully.

**IMPORTANT**

This step is very important because a new container will be created sharing the same volume for database data. Having two containers using the same volume can corrupt the database. Do not restart the `mysql-1` container.

Use the following command to stop the container:

```
[student@workstation ~]$ sudo podman stop mysql-1
616azfaa55x866e7eb18b1fd0423a5461d8f24c147b1ad8668b76e6167587cdd
```

5. Create a new container with the following characteristics:

- **Name:** mysql-2
- **Run as a daemon:** yes
- **Volume:** from **/var/local/mysql** host folder to **/var/lib/mysql/data** container folder
- **Container image:** rhsc1/mysql-57-rhel7
- **Port forward:** yes, from host port 13306 to container port 3306
- **Environment variables:**
 - MySQL_USER: user1
 - MySQL_PASSWORD: mypa55
 - MySQL_DATABASE: items
 - MySQL_ROOT_PASSWORD: r00tpa55

- 5.1. Create and start the container.

```
[student@workstation ~]$ sudo podman run --name mysql-2 \
> -d -v /var/local/mysql:/var/lib/mysql/data \
> -p 13306:3306 \
> -e MYSQL_USER=user1 -e MYSQL_PASSWORD=mpa55 \
> -e MYSQL_DATABASE=items -e MYSQL_ROOT_PASSWORD=r00tpa55 \
> rhsc1/mysql-57-rhel7
281c0e2790e54cd5a0b8e2a8cb6e3969981b85cde8ac611bf7ea98ff78bdffbb
```

- 5.2. Verify that the container was started correctly.

```
[student@workstation ~]$ sudo podman ps
CONTAINER ID      ...output omitted...    NAMES
281c0e2790e5      ...output omitted...    mysql-2
```

6. Save the list of all containers (including stopped ones) to the **/tmp/my-containers** file.

Save the information with the following command:

```
[student@workstation ~]$ sudo podman ps -a > /tmp/my-containers
```

7. Access the Bash shell inside the container and verify that the `items` database and the `Item` table are still available. Confirm also that the table contains data.

7.1. Access the Bash shell inside the container.

```
[student@workstation ~]$ sudo podman exec -it mysql-2 /bin/bash
```

7.2. Connect to the MySQL server.

```
bash-4.2$ mysql -uroot
```

7.3. List all databases and confirm that the `items` database is available.

```
mysql> show databases;
+-----+
| Database      |
+-----+
| information_schema |
| items          |
| mysql          |
| performance_schema |
| sys            |
+-----+
5 rows in set (0.03 sec)
```

7.4. List all tables from the `items` database and verify that the `Item` table is available.

```
mysql> use items;
Database changed
mysql> show tables;
+-----+
| Tables_in_items |
+-----+
| Item           |
+-----+
1 row in set (0.01 sec)
```

7.5. View the data from the table.

```
mysql> SELECT * FROM Item;
+---+-----+---+
| id | description      | done |
+---+-----+---+
| 1  | Pick up newspaper | 0   |
| 2  | Buy groceries     | 1   |
+---+-----+---+
```

7.6. Exit from the MySQL client and from the container shell.

```
mysql> exit
Bye
```

```
bash-4.2$ exit
```

8. Using port forwarding, insert a new row into the `Item` table. The row should have a `description` value of **Finished lab**, and a `done` value of **1**.

8.1. Connect to the MySQL database.

```
[student@workstation ~]$ mysql -uuser1 -h workstation.lab.example.com \
> -pmypa55 -P13306 items
...output omitted...

Welcome to the MariaDB monitor. Commands end with ; or \g.
...output omitted...

MySQL [items]>
```

8.2. Insert the new row.

```
MySQL[items]> insert into Item (description, done) values ('Finished lab', 1);
Query OK, 1 row affected (0.00 sec)
```

8.3. Exit from the MySQL client.

```
MySQL[items]> exit
Bye
```

9. Because the first container is not required any more, remove it to release resources.
Use the following command to remove the container:

```
[student@workstation ~]$ sudo podman rm mysql-1
616azfaa55x866e7eb18b1fd0423a5461d8f24c147b1ad8668b76e6167587cdd
```

Evaluation

Grade your work by running the `lab manage-review grade` command from your workstation machine. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab manage-review grade
```

Finish

On workstation, run the `lab manage-review finish` command to complete this lab.

```
[student@workstation ~]$ lab manage-review finish
```

This concludes the lab.

SUMMARY

In this chapter, you learned:

- Podman has subcommands to: create a new container (**run**), delete a container (**rm**), list containers (**ps**), stop a container (**stop**), and start a process in a container (**exec**).
- Default container storage is ephemeral, meaning its contents are not present after the container restarts or is removed.
- Containers can use a folder from the host file system to work with persistent data.
- Podman mounts volumes in a container with the **-v** option in the **podman run** command.
- The **podman exec** command starts an additional process inside a running container.
- Podman maps local ports to container ports by using the **-p** option in the **run** subcommand.

CHAPTER 4

MANAGING CONTAINER IMAGES

GOAL

Manage the life cycle of a container image from creation to deletion.

OBJECTIVES

- Search for and pull images from remote registries.
- Export, import, and manage container images locally and in a registry.

SECTIONS

- Accessing Registries (and Quiz)
- Manipulating Container Images (and Guided Exercise)

LAB

- Managing Container Images

ACCESSING REGISTRIES

OBJECTIVES

After completing this section, students should be able to:

- Search for and pull images from remote registries using Podman commands and the registry REST API.
- List the advantages of using a certified public registry to download secure images.
- Customize the configuration of Podman to access alternative container image registries.
- List images downloaded from a registry to the local file system.
- Manage tags to pull tagged images.

PUBLIC REGISTRIES

Image registries are services offering container images to download. They allow image creators and maintainers to store and distribute container images to public or private audiences.

Podman searches for and downloads container images from public and private registries. Red Hat Container Catalog is the public image registry managed by Red Hat. It hosts a large set of container images, including those provided by major open source projects, such as Apache, MySQL, and Jenkins. All images in the Container Catalog are vetted by the Red Hat internal security team, meaning they are trustworthy and secured against security flaws.

Red Hat container images provide the following benefits:

- *Trusted source*: All container images comprise sources known and trusted by Red Hat.
- *Original dependencies*: None of the container packages have been tampered with, and only include known libraries.
- *Vulnerability-free*: Container images are free of known vulnerabilities in the platform components or layers.
- *Runtime protection*: All applications in container images run as non-root users, minimizing the exposure surface to malicious or faulty applications.
- *Red Hat Enterprise Linux (RHEL) compatible*: Container images are compatible with all RHEL platforms, from bare metal to cloud.
- *Red Hat support*: Red Hat commercially supports the complete stack.

Quay.io is another public image repository sponsored by Red Hat. Quay.io introduces several exciting features, such as server-side image building, fine-grained access controls, and automatic scanning of images for known vulnerabilities.

While Red Hat Container Catalog images are trusted and verified, Quay.io offers live images regularly updated by creators. Quay.io users can create their namespaces, with fine-grained access control, and publish the images they create to that namespace. Container Catalog users rarely or never push new images, but consume trusted images generated by the Red Hat team.

PRIVATE REGISTRIES

Some image creators or maintainers want to make their images public available. Other image creators, however, prefer to keep their images private due to:

- Company privacy and secret protection.
- Legal restrictions and laws.
- Avoidance of publishing images in development.

Private registries give image creators the control about their images placement, distribution and usage.

CONFIGURING REGISTRIES IN PODMAN

To configure registries for the **podman** command, you need to update the **/etc/containers/registries.conf** file. Edit the **registries** entry in the **[registries.search]** section, adding an entry to the values list.

```
[registries.search]
registries = ["registry.redhat.io", "quay.io"]
```



NOTE

Use an FQDN and port number to identify a registry. A registry that does not include a port number has a default port number of 5000. If the registry uses a different port, it must be specified. Indicate port numbers by appending a colon (:) and the port number after the FQDN.

Secure connections to a registry require a trusted certificate. To support insecure connections, add the registry name to the **registries** entry in **[registries.insecure]** section of **/etc/containers/registries.conf** file:

```
[registries.insecure]
registries = ['localhost:5000']
```

ACCESSING REGISTRIES

Searching for Images in Registries

The **podman search** command finds images by image name, user name, or description from all the registries listed in the **/etc/containers/registries.conf** configuration file. The syntax for the **podman search** command is shown below:

```
[student@workstation ~]$ sudo podman search [OPTIONS] <term>
```

The following table shows some useful options available for the **search** subcommand:

OPTION	DESCRIPTION
--limit <number>	Limits the number of listed images per registry.

OPTION	DESCRIPTION
<code>--filter <filter=value></code>	Filter output based on conditions provided. Supported filters are: <ul style="list-style-type: none"> • stars=<number>: Show only images with at least this number of stars. • is-automated=<true false>: Show only images automatically built. • is-official=<true false>: Show only images flagged as official.
<code>--tls-verify <true false></code>	Enables or disables HTTPS certificate validation for all used registries. true

Registry HTTP API

A remote registry exposes web services that provide an application programming interface (API) to the registry. Podman uses these interfaces to access and interact with remote repositories.

Many registries conform to the **Docker Registry HTTP API v2** specification, which exposes a standardized REST interface for registry interaction. You can use this REST interface to directly interact with a registry, instead of using Podman.

Some samples using this API with **curl** commands are shown below:

To list all repositories available in a registry, use the **/v2/_catalog** endpoint. The **n** parameter is used to limit the number of repositories to return.

```
[student@workstation ~]$ curl -Ls https://registry.lab.example.com/v2/_catalog?n=3
{"repositories": ["centos/httpd", "do180/custom-httpd", "hello-openshift"]}
```



NOTE

If Python is available, use it to format the JSON response:

```
[student@workstation ~]$ curl \
> -Ls https://registry.lab.example.com/v2/_catalog?n=3 \
> | python -m json.tool
{
  "repositories": [
    "centos/httpd",
    "do180/custom-httpd",
    "hello-openshift"
  ]
}
```

The **/v2/<name>/tags/list** endpoint provides the list of tags available for a single image:

```
[student@workstation ~]$ curl -Ls https://quay.io/v2/bitnami/nginx/tags/list \
> | python -m json.tool
```

```
{  
  "name": "bitnami/nginx",  
  "tags": [  
    "1.10.3-r1",  
    "1.10.3-r2",  
    "1.12.0-r0",  
    "1.12.0-r1",  
    "1.12.0-r2",  
    "1.12.0-r3",  
    "1.13.2-r0",  
    "1",  
    "1.12.0-r4",  
    ...output omitted...
```

Registry Authentication

Some container image registries require access authorization. The **podman login** command allows username and password authentication to a registry:

```
[student@workstation ~]$ sudo podman login -u username \  
> -p password registry.connect.redhat.com  
Login Succeeded!
```

The registry HTTP API requires authentication credentials. First, use the Red Hat Single Sign On (SSO) service to obtain an access token:

```
[student@workstation ~]$ curl -u username:password \  
> -Ls "https://sso.redhat.com/auth/realms/rhcc/protocol/redhat-docker-v2/auth?<br>  
service=docker-registry"  
{"token":"eyJh...o5G8",  
"access_token":"eyJh...mgL4",  
"expires_in":...output omitted...}[student@workstation ~]$
```

Then, include this token in a **Bearer** authorization header in subsequent requests:

```
[student@workstation ~]$ curl -H "Authorization: Bearer eyJh...mgL4 \  
> -Ls https://registry.redhat.io/v2/rhscl/mysql-57-rhel7/tags/list \  
> | python -mjson.tool  
{  
  "name": "rhscl/mysql-57-rhel7",  
  "tags": [  
    "5.7-3.9",  
    "5.7-3.8",  
    "5.7-3.4",  
    "5.7-3.7",  
    ...output omitted...
```



NOTE

Other registries may require different steps to provide credentials. If a registry adheres to the **Docker Registry HTTP v2 API**, authentication conforms to the RFC7235 scheme.

Pulling Images

To pull container images from a registry, use the **podman pull** command:

```
[student@workstation ~]$ sudo podman pull [OPTIONS] [REGISTRY[:PORT]/]NAME[:TAG]
```

The **podman pull** command uses the image name obtained from the **search** subcommand to pull an image from a registry. The **pull** subcommand allows adding the registry name to the image. This variant supports having the same image in multiple registries.

For example, to pull an NGINX container from the quay.io registry, use the following command:

```
[student@workstation ~]$ sudo podman pull quay.io/nginx
```



NOTE

If the image name does not include a registry name, Podman searches for a matching container image using the registries listed in the **/etc/containers/registries.conf** configuration file. Podman search for images in registries in the same order they appear in the configuration file.

Listing Local Copies of Images

Any container image downloaded from a registry is stored locally on the same host where the **podman** command is executed. This behavior avoids repeating image downloads and minimizes the deployment time for a container. Podman also stores any custom container images you build in the same local storage.



NOTE

By default, Podman stores container images in the **/var/lib/containers/storage/overlay-images** directory.

Podman provides an **images** subcommand to list all the container images stored locally.

```
[student@workstation ~]$ sudo podman images
REPOSITORY           TAG      IMAGE ID      CREATED     SIZE
registry.redhat.io/rhscl/mysql-57-rhel7  latest   c07bf25398f4  13 days ago  444MB
```

Image Tags

An image tag is a mechanism to support multiple releases of the same image. This feature is useful when multiple versions of the same software are provided, such as a production-ready container or the latest updates of the same software developed for community evaluation. Any Podman subcommand that requires a container image name accepts a tag parameter to differentiate between multiple tags. If an image name does not contain a tag, then the tag value defaults to **latest**. For example, to pull an image with the tag **5.6** from **mysql**, use the following command:

```
[student@workstation ~]$ sudo podman pull mysql:5.6
```

To start a new container based on the **mysql:5.6** image, use the following command:

```
[student@workstation ~]$ sudo podman run mysql:5.6
```



REFERENCES

Red Hat Container Catalog

<https://registry.redhat.io>

Quay.io

<https://quay.io>

Docker Registry HTTP API V2

<https://github.com/docker/distribution/blob/master/docs/spec/api.md>

RFC7235 - HTTP/1.1: Authentication

<https://tools.ietf.org/html/rfc7235>

► QUIZ

WORKING WITH REGISTRIES

Choose the correct answers to the following questions, based on the following information:

Podman is available on a RHEL host with the following entry in **/etc/containers/registries.conf** file:

```
[registries.search]
registries = ["registry.redhat.io", "quay.io"]
```

The `registry.redhat.io` and `quay.io` hosts have a registry running, both have valid certificates, and use the version 1 registry. The following images are available for each host:

Image names/tags per registry

REGISTRY	IMAGE
<code>registry.redhat.io</code>	<ul style="list-style-type: none"> • <code>nginx/1.0</code> • <code>mysql/5.6</code> • <code>httpd/2.2</code>
<code>quay.io</code>	<ul style="list-style-type: none"> • <code>mysql/5.5</code> • <code>httpd/2.4</code>

No images are locally available.

► 1. Which two commands display mysql images available for download from `registry.redhat.io`? (Choose two.)

- `podman search registry.redhat.io/mysql`
- `podman images`
- `podman pull mysql`
- `podman search mysql`

► 2. Which command is used to list all available image tags for the httpd container image?

- `podman search httpd`
- `podman images httpd`
- `podman pull --all-tags=true httpd`
- There is no podman command available to search for tags.

► 3. Which two commands pull the httpd image with the 2.2 tag? (Choose two.)

- a. `podman pull httpd:2.2`
- b. `podman pull httpd:latest`
- c. `podman pull quay.io/httpd`
- d. `podman pull registry.redhat.io/httpd:2.2`

► 4. When running the following commands, what container images will be downloaded?

```
podman pull registry.redhat.io/httpd:2.2  
podman pull quay.io/mysql:5.6
```

- a. `quay.io/httpd:2.2`
`registry.redhat.io/mysql:5.6`
- b. `registry.redhat.io/httpd:2.2`
`registry.redhat.io/mysql:5.6`
- c. `registry.redhat.io/httpd:2.2`
No image will be downloaded for mysql.
- d. `quay.io/httpd:2.2`
No image will be downloaded for mysql.

► SOLUTION

WORKING WITH REGISTRIES

Choose the correct answers to the following questions, based on the following information:

Podman is available on a RHEL host with the following entry in **/etc/containers/registries.conf** file:

```
[registries.search]
registries = ["registry.redhat.io", "quay.io"]
```

The `registry.redhat.io` and `quay.io` hosts have a registry running, both have valid certificates, and use the version 1 registry. The following images are available for each host:

Image names/tags per registry

REGISTRY	IMAGE
<code>registry.redhat.io</code>	<ul style="list-style-type: none"> <code>nginx/1.0</code> <code>mysql/5.6</code> <code>httpd/2.2</code>
<code>quay.io</code>	<ul style="list-style-type: none"> <code>mysql/5.5</code> <code>httpd/2.4</code>

No images are locally available.

► 1. Which two commands display mysql images available for download from `registry.redhat.io`? (Choose two.)

- `podman search registry.redhat.io/mysql`
- `podman images`
- `podman pull mysql`
- `podman search mysql`

► 2. Which command is used to list all available image tags for the httpd container image?

- `podman search httpd`
- `podman images httpd`
- `podman pull --all-tags=true httpd`
- There is no podman command available to search for tags.

► 3. Which two commands pull the httpd image with the 2.2 tag? (Choose two.)

- a. `podman pull httpd:2.2`
- b. `podman pull httpd:latest`
- c. `podman pull quay.io/httpd`
- d. `podman pull registry.redhat.io/httpd:2.2`

► 4. When running the following commands, what container images will be downloaded?

```
podman pull registry.redhat.io/httpd:2.2  
podman pull quay.io/mysql:5.6
```

- a. `quay.io/httpd:2.2`
`registry.redhat.io/mysql:5.6`
- b. `registry.redhat.io/httpd:2.2`
`registry.redhat.io/mysql:5.6`
- c. `registry.redhat.io/httpd:2.2`
No image will be downloaded for mysql.
- d. `quay.io/httpd:2.2`
No image will be downloaded for mysql.

MANIPULATING CONTAINER IMAGES

OBJECTIVES

After completing this section, students should be able to:

- Save and load container images to local files.
- Delete images from the local storage.
- Create new container images from containers and update image metadata.
- Manage image tags for distribution purposes.

INTRODUCTION

There are various ways to manage image containers while adhering to DevOps principles. For example, a developer finishes testing a custom container in a machine and needs to transfer this container image to another host for another developer, or to a production server. There are two ways to do this:

1. Save the container image to a **.tar** file.
2. Publish (*push*) the container image to an image registry.



NOTE

One of the ways a developer could have created this custom container is discussed later in this chapter (**podman commit**). However, in the following chapters we discuss the recommended way to do so using **Dockerfiles**.

SAVING AND LOADING IMAGES

Existing images from the Podman local storage can be saved to a **.tar** file using the **podman save** command. The generated file is not a regular TAR archive; it contains image metadata and preserves the original image layers. Using this file, Podman can recreate the original image exactly as it was.

The general syntax of the **save** subcommand is as follows:

```
[student@workstation ~]$ sudo podman save [-o FILE_NAME] IMAGE_NAME[:TAG]
```

Podman sends the generated image to the standard output as binary data. To avoid that, use the **-o** option.

The following example saves the previously downloaded MySQL container image from the Red Hat Container Catalog to the **mysql.tar** file:

```
[student@workstation ~]$ sudo podman save \
> -o mysql.tar registry.access.redhat.com/rhscl/mysql-57-rhel7
```

Use the **.tar** files generated by the **save** subcommand for backup purposes. To restore the container image, use the **podman load** command. The general syntax of the command is as follows:

```
[student@workstation ~]$ sudo podman load [-i FILE_NAME]
```

For example, this command would load an image saved in a file named **mysql.tar**.

```
[student@workstation ~]$ sudo podman load -i mysql.tar
```

If the **.tar** file given as an argument is not a container image with metadata, the **podman load** command fails.



NOTE

To save disk space, compress the file generated by the **save** subcommand with Gzip using the **--compress** parameter. The **load** subcommand uses the **gunzip** command before importing the file to the local storage.

DELETING IMAGES

Podman keeps any image downloaded in its local storage, even the ones currently unused by any container. However, images can become outdated, and should be subsequently replaced.



NOTE

Any updates to images in a registry are not automatically updated. The image must be removed and then pulled again to guarantee that the local storage has the latest version of an image.

To delete an image from the local storage, run the **podman rmi** command. The syntax for this command is as follows:

```
[student@workstation ~]$ sudo podman rmi [OPTIONS] IMAGE [IMAGE...]
```

An image can be referenced using its name or its ID for removal purposes. Podman cannot delete images while containers are using that image. You must stop and remove all containers using that image before deleting it.

To avoid this, the **rmi** subcommand has the **--force** option. This option forces the removal of an image even if that the image is used by several containers or these containers are running. Podman stops and removes all containers using the forcefully removed image before removing it.

DELETING ALL IMAGES

To delete all images that are not used by any container, use the following command:

```
[student@workstation ~]$ sudo podman rmi -a
```

The command returns all the image IDs available in the local storage and passes them as parameters to the **podman rmi** command for removal. Images that are in use are not deleted. However, this does not prevent any unused images from being removed.

MODIFYING IMAGES

Ideally, all container images should be built using a **Dockerfile**, in order to create a clean, lightweight set of image layers without log files, temporary files, or other artifacts created by the container customization. However, some users may provide container images as they are, without a **Dockerfile**. As an alternative approach to creating new images, change a running container in place and save its layers to create a new container image. The **podman commit** command provides this feature.



WARNING

Even though the **podman commit** command is the most straightforward approach to creating new images, it is not recommended because of the image size (**commit** keeps logs and process ID files in the captured layers), and the lack of change traceability. A **Dockerfile** provides a robust mechanism to customize and implement changes to a container using a human-readable set of commands, without the set of files that are generated by the operating system.

The syntax for the **podman commit** command is as follows:

```
[student@workstation ~]$ sudo podman commit [OPTIONS] CONTAINER \
> [REPOSITORY[:PORT]/]IMAGE_NAME[:TAG]
```

The following table shows the most important options available for the **podman commit** command:

OPTION	DESCRIPTION
--author ""	Identifies who created the container image.
--message ""	Includes a commit message to the registry.
--format	Selects the format of the image. Valid options are oci and docker .



NOTE

The **--message** option is not available in the default OCI container format.

To find the ID of a running container in Podman, run the **podman ps** command:

```
[student@workstation ~]$ sudo podman ps
CONTAINER ID IMAGE ... NAMES
87bdfcc7c656 mysql ...output omitted... mysql-basic
```

Eventually, administrators might customize the image and set the container to the desired state. To identify which files were changed, created, or deleted since the container was started, use the **diff** subcommand. This subcommand only requires the container name or container ID:

```
[student@workstation ~]$ sudo podman diff mysql-basic
C /run
C /run/mysql
A /run/mysql/mysql.pid
A /run/mysql/mysql.sock
A /run/mysql/mysql.sock.lock
A /run/secrets
```

The **diff** subcommand tags any added file with an **A**, any changed ones with a **C**, and any deleted file with a **D**.



NOTE

The **diff** command only reports added, changed, or deleted files to the container file system. Files that are mounted to a running container are not considered part of the container file system.

To retrieve the list of mounted files and directories for a running container, use the **podman inspect** command:

```
[student@workstation ~]$ sudo podman inspect \
> -f "{{range .Mounts}}{{println .Destination}}{{end}}" CONTAINER_NAME/ID
```

Any file in this list, or file under a directory in this list, is not shown in the output of the **podman diff** command.

To commit the changes to another image, run the following command:

```
[student@workstation ~]$ sudo podman commit mysql-basic mysql-custom
```

TAGGING IMAGES

A project with multiple images based on the same software could be distributed, creating individual projects for each image, but this approach requires more maintenance for managing and deploying the images to the correct locations.

Container image registries support tags to distinguish multiple releases of the same project. For example, a customer might use a container image to run with a MySQL or PostgreSQL database, using a tag as a way to differentiate which database is to be used by a container image.



NOTE

Usually, the tags are used by container developers to distinguish between multiple versions of the same software. Multiple tags are provided to identify a release easily. The official MySQL container image website uses the version as the tag's name (**5.5.16**). Also, the same image has a second tag with the minor version, such as 5.5, to minimize the need to get the latest release for a specific version.

To tag an image, use the **podman tag** command:

```
[student@workstation ~]$ sudo podman tag [OPTIONS] IMAGE[:TAG] \
```

```
> [REGISTRYHOST/] [USERNAME/] NAME[:TAG]
```

The **IMAGE** argument is the image name with an optional tag, which is managed by Podman. The following argument refers to the new alternative name for the image. Podman assumes the latest version, as indicated by the **latest** tag, if the tag value is absent. For example, to tag an image, use the following command:

```
[student@workstation ~]$ sudo podman tag mysql-custom devops/mysql
```

The **mysql-custom** option corresponds to the image name in the container registry.

To use a different tag name, use the following command instead:

```
[student@workstation ~]$ sudo podman tag mysql-custom devops/mysql:snapshot
```

Removing Tags from Images

A single image can have multiple tags assigned using the **podman tag** command. To remove them, use the **podman rmi** command, as mentioned earlier:

```
[student@workstation ~]$ sudo podman rmi devops/mysql:snapshot
```



NOTE

Because multiple tags can point to the same image, to remove an image referred to by multiple tags, first remove each tag individually.

BEST PRACTICES FOR TAGGING IMAGES

Podman automatically adds the **latest** tag if you do not specify any tag, because Podman considers the image to be the latest build. However, this may not be true depending on how each project uses tags. For example, many open source projects consider the **latest** tag to match the most recent release, but not the latest build.

Moreover, multiple tags are provided to minimize the need to remember the latest release of a particular version of a project. Thus, if there is a project version release, for example, **2.1.10**, another tag called **2.1** can be created pointing to the same image from the **2.1.10** release. This simplifies pulling images from the registry.

PUBLISHING IMAGES TO A REGISTRY

To publish an image to a registry, it must reside in the Podman's local storage and be tagged for identification purposes. To push the image to the registry the syntax of the **push** subcommand is:

```
[student@workstation ~]$ sudo podman push [OPTIONS] IMAGE [DESTINATION]
```

For example, to push the **nginx** image to its repository, use the following command:

```
[student@workstation ~]$ sudo podman push nginx
```



REFERENCES

Podman site

<https://podman.io/>

► GUIDED EXERCISE

CREATING A CUSTOM APACHE CONTAINER IMAGE

In this guided exercise, you will create a custom Apache container image using the **podman commit** command.

OUTCOMES

You should be able to create a custom container image.

BEFORE YOU BEGIN

Open a terminal on **workstation** as the **student** user and run the following command:

```
[student@workstation ~]$ lab image-operations start
```

- 1. Open a terminal on **workstation** (Applications → System Tools → Terminal) and start a container by using the image available at **centos/httpd**. The **-p** option allows you to specify a redirect port. In this case, Podman forwards incoming requests on TCP port 8180 of the host to TCP port 80 of the container.

```
[student@workstation ~]$ sudo podman run -d --name official-httdp \
> -p 8180:80 centos/httpd
...output omitted...
Writing manifest to image destination
Storing signatures
3a6baecaff2b4e8c53b026e04847dda5976b773ade1a3a712b1431d60ac5915d
```

Your last line of output is different from the last line shown above. Remember the first few characters

- 2. Create an HTML page on the **official-httdp** container.

- 2.1. Access the shell of the container by using the **exec** subcommand and create an HTML page.

```
[student@workstation ~]$ sudo podman exec -it official-httdp /bin/bash
[root@3a6baecaff2b /]# echo "DO180 Page" > /var/www/html/do180.html
```

- 2.2. Exit the container.

```
[root@3a6baecaff2b /]# exit
```

- 2.3. Ensure that the HTML file is reachable from the **workstation** VM by using the **curl** command.

```
[student@workstation ~]$ curl 127.0.0.1:8180/do180.html
```

You should see the following output:

D0180 Page

- 3. Use the **diff** subcommand to examine the differences in the container between the image and the new layer created by the container.

- 3.1. Retrieve the list of external files and directories that Podman mounts to the running container:

```
[student@workstation ~]$ sudo podman inspect -f \
> "{{range .Mounts}}{{println .Destination}}{{end}}" official-httdp
/proc
/dev
/dev/pts
/dev/shm
/dev/mqueue
/sys
/sys/fs/cgroup
```

- 3.2. Execute the **podman diff** command to see a list of modified files in the container file system:

```
[student@workstation ~]$ sudo podman diff official-httdp
C /etc
C /root
A /root/.bash_history
...output omitted...
C /tmp
C /var
C /var/log
C /var/log/httpd
A /var/log/httpd/access_log
A /var/log/httpd/error_log
C /var/www
C /var/www/html
```

```
A /var/www/html/do180.html
```

Notice that none of the files or directories in the output are in the list of externally mounted files in the previous step.



NOTE

Often, web server container images label the `/var/www/html` directory as a volume. In these cases, any files added to this directory are not considered part of the container file system, and would not show in the output of the `git diff` command.

The `centos/httpd` container image does not label the `/var/www/html` directory as a volume. As a result, the change to the `/var/www/html/do180.html` file is considered a change to the underlying container file system.

- ▶ 4. Create a new image with the changes created by the running container.

- 4.1. Stop the `official-httpd` container.

```
[student@workstation ~]$ sudo podman stop official-httpd
3a6baecaff2b4e8c53b026e04847dda5976b773ade1a3a712b1431d60ac5915d
```

- 4.2. Commit the changes to a new container image with a new name. Use your name as the author of the changes.

```
[student@workstation ~]$ sudo podman commit \
> -a 'Your Name' official-httpd do180/custom-httpd
Getting image source signatures
Skipping fetch of repeat blob sha256:071d8bd765171080d01682844524be57ac9883e...
...output omitted...
Copying blob sha256:1e19be875ce6f5b9dece378755eb9df96ee205abfb4f165c797f59a9...
15.00 KB / 15.00 KB [=====] 0s
Copying config sha256:8049dc2e7d0a0b1a70fc0268ad236399d9f5fb686ad4e31c7482cc...
2.99 KB / 2.99 KB [=====] 0s
Writing manifest to image destination
Storing signatures
31c3ac78e9d4137c928da23762e7d32b00c428eb1036cab1caeeb399befef2a23
```

- 4.3. List the available container images.

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

The expected output is similar to the following:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
localhost/do180/custom-httpd	latest	31c3ac78e9d4
registry.lab.example.com/centos/httpd	latest	2cc07fbb5000

The image ID matches the first 12 characters of the hash. The most recent images are listed at the top.

► 5. Publish the saved container image to the container registry.

- 5.1. To tag the image with the registry host name and tag, run the following command.

```
[student@workstation ~]$ sudo podman tag do180/custom-httdp \
> registry.lab.example.com/do180/custom-httdp:v1.0
```

- 5.2. Run the **podman images** command to ensure that the new name has been added to the cache.

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

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
localhost/do180/custom-httdp	latest	31c3ac78e9d4
registry.lab.example.com/do180/custom-httdp	v1.0	31c3ac78e9d4
registry.lab.example.com/centos/httdp	latest	2cc07fbb5000

- 5.3. Publish the image to the private registry, accessible at `registry.lab.example.com`.

```
[student@workstation ~]$ sudo podman push \
> registry.lab.example.com/do180/custom-httdp:v1.0
Getting image source signatures
Copying blob sha256:071d8bd765171080d01682844524be57ac9883e53079b6ac66707e19...
 200.44 MB / 200.44 MB [=====] 1m38s
...output omitted...
Copying config sha256:31c3ac78e9d4137c928da23762e7d32b00c428eb1036cab1caeab3...
 2.99 KB / 2.99 KB [=====] 0s
Writing manifest to image destination
Copying config sha256:31c3ac78e9d4137c928da23762e7d32b00c428eb1036cab1caeab3...
 0 B / 2.99 KB [-----] 0s
Writing manifest to image destination
Storing signatures
```



NOTE

Each student only has access to their own private registry; it is therefore impossible that they will interfere with each other. Even though these private registries do not require authentication, most public registries require that you authenticate before pushing any image.

- 5.4. Verify that the image is returned by the **podman search** command.

```
[student@workstation ~]$ podman search custom-httdp
INDEX          NAME                           DESCRIPTION STARS ...
example.com    registry.lab.example.com/do180/custom-httdp      0
```

- 6. Create a container from the newly published image.

Use the **podman run** command to start a new container. Use **do180/custom-
httpd:v1.0** as the base image.

```
[student@workstation ~]$ sudo podman run -d --name test-httpd -p 8280:80 \
> do180/custom-httpd:v1.0
c0f04e906bb12bd0e514cbd0e581d2746e04e44a468dfbc85bc29ffcc5acd16c
```

- 7. Use the **curl** command to access the HTML page. Make sure you use port 8280.

This should display the HTML page created in the previous step.

```
[student@workstation ~]$ curl http://localhost:8280/do180.html
DO180 Page
```

Finish

On workstation, run the **lab image-operations finish** script to complete this lab.

```
[student@workstation ~]$ lab image-operations finish
```

This concludes the guided exercise.

► LAB

MANAGING IMAGES

PERFORMANCE CHECKLIST

In this lab, you will create and manage container images.

OUTCOMES

You should be able to create a custom container image and manage container images.

BEFORE YOU BEGIN

Open a terminal on workstation as the student user and run the following command:

```
[student@workstation ~]$ lab image-review start
```

1. Use the **podman search** command to locate the **nginx** image and pull it into your local file system.



NOTE

There are three Nginx images in the repository, and two of them belong to the Red Hat Software Collection Library. Use the other image for this lab.

2. Ensure that the image has been successfully retrieved.
2. Start a new container using the Nginx image, according to the specifications listed in the following list.
 - **Name:** **official-nginx**
 - **Run as daemon:** yes
 - **Container image:** **nginx**
 - **Port forward:** from host port 8080 to container port 80.
3. Log in to the container using the **exec** subcommand. Replace the contents of the **index.html** file with **DO180**. The web server directory is located at **/usr/share/nginx/html**.
After the file has been updated, exit the container and use the **curl** command to access the web page.
4. Stop the running container and commit your changes to create a new container image. Give the new image a name of **do180/mynginx** and a tag of **v1.0-SNAPSHOT**. Use the following specifications:
 - Image name: **do180/mynginx**
 - Image tag: **v1.0-SNAPSHOT**

- Author name: *your name*
5. Start a new container using the updated Nginx image, according to the specifications listed in the following list.
- **Name:** **official-nginx-dev**
 - **Run as daemon:** yes
 - **Container image:** **do180/mynginx:v1.0-SNAPSHOT**
 - **Port forward:** from host port 8080 to container port 80.
6. Log in to the container using the **exec** subcommand to introduce a final change. Replace the contents of the file **/usr/share/nginx/html/index.html** file with **DO180 Page**. After the file has been updated, exit the container and use the **curl** command to verify the changes.
7. Stop the running container and commit your changes to create the final container image. Give the new image a name of **do180/mynginx** and a tag of **v1.0**. Use the following specifications:
- Image name: **do180/mynginx**
 - Image tag: **v1.0**
 - Author name: *your name*
8. Remove the development image **do180/mynginx:v1.0-SNAPSHOT** from local image storage.
9. Use the image tagged **do180/mynginx:v1.0** to create a new container with the following specifications:
- Container name: **my-nginx**
 - Run as daemon: yes
 - Container image: **do180/mynginx:v1.0**
 - Port forward: from host port 8280 to container port 80

On workstation, use the **curl** command to access the web server, accessible from the port 8280.

Evaluation

Grade your work by running the **lab image-review grade** command on your workstation machine. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab image-review grade
```

Finish

On workstation, run the **lab image-review finish** command to complete this lab.

```
[student@workstation ~]$ lab image-review finish
```

This concludes the lab.

► SOLUTION

MANAGING IMAGES

PERFORMANCE CHECKLIST

In this lab, you will create and manage container images.

OUTCOMES

You should be able to create a custom container image and manage container images.

BEFORE YOU BEGIN

Open a terminal on workstation as the `student` user and run the following command:

```
[student@workstation ~]$ lab image-review start
```

1. Use the `podman search` command to locate the `nginx` image and pull it into your local file system.

NOTE

There are three Nginx images in the repository, and two of them belong to the Red Hat Software Collection Library. Use the other image for this lab.

Ensure that the image has been successfully retrieved.

- 1.1. Use the `podman search` command to search for the Nginx container image `nginx`.

```
[student@workstation ~]$ sudo podman search nginx
INDEX          NAME                            DESCRIPTION      ...
example.com    registry.lab.example.com/nginx ...
example.com    registry.lab.example.com/rhscl/nginx-16-rhel7 ...
example.com    registry.lab.example.com/rhscl/nginx-18-rhel7 ...
```

- 1.2. Use the `podman pull` command to pull the Nginx container image.

```
[student@workstation ~]$ sudo podman pull registry.lab.example.com/nginx
Trying to pull registry.lab.example.com/nginx:latest ...output omitted...
...output omitted...
Storing signatures
42b4762643dcc9bf492b08064b55fef64942f055f0da91289a8abf93c6d6b43c
```

- 1.3. Ensure that the container image is available locally by running the `podman images` command.

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

This command produces output similar to the following:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
registry.lab.example.com/nginx	latest	42b4762643dc	8 days ago	113MB

2. Start a new container using the Nginx image, according to the specifications listed in the following list.

- **Name:** **official-nginx**
- **Run as daemon:** yes
- **Container image:** **nginx**
- **Port forward:** from host port 8080 to container port 80.

- 2.1. On workstation, use the **podman run** command to create a container named **official-nginx**.

```
[student@workstation ~]$ sudo podman run --name official-nginx \
> -d -p 8080:80 registry.lab.example.com/nginx
02dbc348c7dcf8560604a44b11926712f018b0ac44063d34b05704fb8447316f
```

3. Log in to the container using the **exec** subcommand. Replace the contents of the **index.html** file with **D0180**. The web server directory is located at **/usr/share/nginx/html**.

After the file has been updated, exit the container and use the **curl** command to access the web page.

- 3.1. Log in to the container by using the **podman exec** command.

```
[student@workstation ~]$ sudo podman exec -it official-nginx /bin/bash
root@f22c60d901fa:/#
```

- 3.2. Update the **index.html** file located at **/usr/share/nginx/html**. The file should read **D0180**.

```
root@f22c60d901fa:/# echo 'D0180' > /usr/share/nginx/html/index.html
```

- 3.3. Exit the container.

```
root@f22c60d901fa:/# exit
```

- 3.4. Use the **curl** command to ensure that the **index.html** file is updated.

```
[student@workstation ~]$ curl 127.0.0.1:8080
D0180
```

4. Stop the running container and commit your changes to create a new container image. Give the new image a name of **do180/mynginx** and a tag of **v1.0-SNAPSHOT**. Use the following specifications:

- Image name: **do180/mynginx**
- Image tag: **v1.0-SNAPSHOT**
- Author name: *your name*

4.1. Use the **sudo podman stop** command to stop the **official-nginx** container.

```
[student@workstation ~]$ sudo podman stop official-nginx  
2dbc348c7dcf8560604a44b11926712f018b0ac44063d34b05704fb8447316f
```

- 4.2. Commit your changes to a new container image. Use your name as the author of the changes.

```
[student@workstation ~]$ sudo podman commit -a 'Your Name' \  
> official-nginx do180/mynginx:v1.0-SNAPSHOT  
Getting image source signatures  
...output omitted...  
Storing signatures  
4a13dd08d175a6095e6462e52431be1577ca931fc1aea139b71346bfc7f9c76
```

- 4.3. List the available container images to locate your newly created image.

```
[student@workstation ~]$ sudo podman images  
REPOSITORY          TAG      IMAGE ID   CREATED       SIZE  
localhost/do180/mynginx    v1.0-SNAPSHOT 4a13dd08d175  5 minutes ago  113MB  
registry.lab.example.com/nginx  latest     42b4762643dc  8 days ago   113MB
```

5. Start a new container using the updated Nginx image, according to the specifications listed in the following list.

- **Name:** **official-nginx-dev**
- **Run as daemon:** yes
- **Container image:** **do180/mynginx:v1.0-SNAPSHOT**
- **Port forward:** from host port 8080 to container port 80.

5.1. On workstation, use the **podman run** command to create a container named **official-nginx-dev**.

```
[student@workstation ~]$ sudo podman run --name official-nginx-dev \  
> -d -p 8080:80 do180/mynginx:v1.0-SNAPSHOT  
02dbc348c7dcf8560604a44b11926712f018b0ac44063d34b05704fb8447316f
```

6. Log in to the container using the **exec** subcommand to introduce a final change. Replace the contents of the file **/usr/share/nginx/html/index.html** file with **D0180 Page**.

After the file has been updated, exit the container and use the **curl** command to verify the changes.

- 6.1. Log in to the container by using the **podman exec** command.

```
[student@workstation ~]$ sudo podman exec -it official-nginx-dev /bin/bash  
root@f22c60d901fa:/#
```

- 6.2. Update the **index.html** file located at **/usr/share/nginx/html**. The file should read **D0180 Page**.

```
root@f22c60d901fa:/# echo 'D0180 Page' > /usr/share/nginx/html/index.html
```

- 6.3. Exit the container.

```
root@f22c60d901fa:/# exit
```

- 6.4. Use the **curl** command to ensure that the **index.html** file is updated.

```
[student@workstation ~]$ curl 127.0.0.1:8080  
D0180 Page
```

7. Stop the running container and commit your changes to create the final container image. Give the new image a name of **do180/mynginx** and a tag of **v1.0**. Use the following specifications:

- Image name: **do180/mynginx**
- Image tag: **v1.0**
- Author name: *your name*

- 7.1. Use the **sudo podman stop** command to stop the **official-nginx-dev** container.

```
[student@workstation ~]$ sudo podman stop official-nginx-dev  
2dbc348c7dcf8560604a44b11926712f018b0ac44063d34b05704fb8447316f
```

- 7.2. Commit your changes to a new container image. Use your name as the author of the changes.

```
[student@workstation ~]$ sudo podman commit -a 'Your Name' \  
> official-nginx-dev do180/mynginx:v1.0  
Getting image source signatures  
...output omitted...  
Storing signatures
```

```
4a13dd08d175a6095e6462e52431be1577ca931fcd1aea139b71346bfc7f9c76
```

- 7.3. List the available container images in order to locate your newly created image.

```
[student@workstation ~]$ sudo podman images
REPOSITORY          TAG      IMAGE ID   CREATED     SIZE
localhost/do180/mynginx    v1.0      892569a87e3f  7 seconds ago  113MB
localhost/do180/mynginx    v1.0-SNAPSHOT 0857d81f5a4b  4 minutes ago  113MB
registry.lab.example.com/nginx  latest      f09fe80eb0e7  5 weeks ago  113MB
```

8. Remove the development image **do180/mynginx:v1.0-SNAPSHOT** from local image storage.

- 8.1. Despite being stopped, the **official-nginx-dev** is still present. Display the container with the **podman ps** command with the **-a** flag.

```
[student@workstation ~]$ sudo podman ps -a \
> --format="{{.ID}} {{.Names}} {{.Status}}"
e169c5fc8c3e  official-nginx-dev  Exited (0) 9 minutes ago
ccf046c2f87d  official-nginx      Exited (0) 12 minutes ago
```

- 8.2. Remove the container with the **podman rm** command.

```
[student@workstation ~]$ sudo podman rm official-nginx-dev
e169c5fc8c3ed5c024af94aec752fa565650f9d07b95bb009329874801d859a10
```

- 8.3. Verify that the container is deleted by resubmitting the same **podman ps** command.

```
[student@workstation ~]$ sudo podman ps -a \
> --format="{{.ID}} {{.Names}} {{.Status}}"
ccf046c2f87d  official-nginx      Exited (0) 12 minutes ago
```

- 8.4. Use the **sudo podman rmi** command to remove the **do180/mynginx:v1.0-SNAPSHOT** image.

```
[student@workstation ~]$ sudo podman rmi do180/mynginx:v1.0-SNAPSHOT
Untagged: localhost/do180/mynginx:v1.0-SNAPSHOT
```

- 8.5. Verify that the image is no longer present by listing all images using the **podman images** command.

```
[student@workstation ~]$ sudo podman images
REPOSITORY          TAG      IMAGE ID   CREATED     SIZE
localhost/do180/mynginx    v1.0      892569a87e3f  13 minutes ago  113MB
registry.lab.example.com/nginx  latest      f09fe80eb0e7  5 weeks ago  113MB
```

9. Use the image tagged **do180/mynginx:v1.0** to create a new container with the following specifications:

- Container name: **my-nginx**
- Run as daemon: yes
- Container image: **do180/mynginx:v1.0**
- Port forward: from host port 8280 to container port 80

On workstation, use the **curl** command to access the web server, accessible from the port 8280.

- 9.1. Use the **sudo podman run** command to create the **my-nginx** container, according to the specifications.

```
[student@workstation ~]$ sudo podman run -d --name my-nginx \
> -p 8280:80 do180/mynginx:v1.0
c1cba44fa67bf532d6e661fc5e1918314b35a8d46424e502c151c48fb5fe6923
```

- 9.2. Use the **curl** command to ensure that the **index.html** page is available and returns the custom content.

```
[student@workstation ~]$ curl 127.0.0.1:8280
DO108 Page
```

Evaluation

Grade your work by running the **lab image-review grade** command on your workstation machine. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab image-review grade
```

Finish

On workstation, run the **lab image-review finish** command to complete this lab.

```
[student@workstation ~]$ lab image-review finish
```

This concludes the lab.

SUMMARY

In this chapter, you learned:

- The Red Hat Container Catalog provides tested and certified images at `registry.redhat.io`.
- Podman can interact with remote container registries to search, pull, and push container images.
- Image tags are a mechanism to support multiple releases of a container image.
- Podman provides commands to manage container images both in local storage and as compressed files.
- Use the **podman commit** to create an image from a container.

CHAPTER 5

CREATING CUSTOM CONTAINER IMAGES

GOAL

Design and code a Dockerfile to build a custom container image.

OBJECTIVES

- Describe the approaches for creating custom container images.
- Create a container image using common Dockerfile commands.

SECTIONS

- Designing Custom Container Images (and Quiz)
- Building Custom Container Images with Dockerfiles (and Guided Exercise)

LAB

- Creating Custom Container Images

DESIGNING CUSTOM CONTAINER IMAGES

OBJECTIVES

After completing this section, students should be able to:

- Describe the approaches for creating custom container images.
- Find existing Dockerfiles to use as a starting point for creating a custom container image.
- Define the role played by the Red Hat Software Collections Library (RHSCl) in designing container images from the Red Hat registry.
- Describe the Source-to-Image (S2I) alternative to Dockerfiles.

REUSING EXISTING DOCKERFILES

One method of creating container images has been covered so far: create a container, modify it to meet the requirements of the application to run in it, and then commit the changes to an image. This option, although straightforward, is only suitable for using or testing very specific changes. It does not follow best software practices, like maintainability, automation of building, and repeatability.

Addressing these limitations are Dockerfiles, another option for creating container images. Dockerfiles are easy to share, version control, reuse, and extend.

Dockerfiles also make it easy to extend one image, called a *child image*, from another image, called a *parent image*. A child image incorporates everything in the parent image and all changes and additions made to create it.

To share and reuse images, many popular applications, languages, and frameworks are already available in public image registries such as Quay . io. It is not trivial to customize an application configuration to follow recommended practices for containers, and so starting from a proven parent image usually saves a lot of work.

Using a high-quality parent image enhances maintainability, especially if the parent image is kept updated by its author to account for bug fixes and security issues.

Typical scenarios in creating a Dockerfile for building a child image from an existing container image include:

- Add new runtime libraries, such as database connectors.
- Include organization-wide customization such as SSL certificates and authentication providers.
- Add internal libraries to be shared as a single image layer by multiple container images for different applications.

Changing an existing Dockerfile to create a new image can also be a sensible approach in other scenarios. For example:

- Trim the container image by removing unused material (such as man pages, or documentation found in `/usr/share/doc`).

- Lock either the parent image or some included software package to a specific release to lower risk related to future software updates.

Two sources of container images to use either as parent images or for changing their Dockerfiles are Docker Hub and the Red Hat Software Collections Library (RHSCl).

WORKING WITH THE RED HAT SOFTWARE COLLECTIONS LIBRARY

Red Hat Software Collections Library (RHSCl), or simply Software Collections, is Red Hat's solution for developers who need to use the latest development tools that usually do not fit the standard RHEL release schedule.

Red Hat Enterprise Linux (RHEL) provides a stable environment for enterprise applications. This requires RHEL to keep the major releases of upstream packages at the same level to prevent API and configuration file format changes. Security and performance fixes are back-ported from later upstream releases, but new features that would break backward-compatibility are not back-ported.

RHSCl allows software developers to use the latest version without impacting RHEL, because RHSCl packages do not replace or conflict with default RHEL packages. Default RHEL packages and RHSCl packages are installed side-by-side.



NOTE

All RHEL subscribers have access to the RHSCl. To enable a particular *software collection* for a specific user or application environment (for example, MySQL 5.7, which is named `rh-mysql157`), enable the RHSCl software Yum repositories and follow a few simple steps.

FINDING DOCKERFILES FROM THE RED HAT SOFTWARE COLLECTIONS LIBRARY

RHSCl is the source of most container images provided by the Red Hat image registry for use by RHEL Atomic Host and OpenShift Container Platform customers.

Red Hat provides RHSCl Dockerfiles and related sources in the `rhscl-dockerfiles` package available from the RHSCl repository. Community users can get Dockerfiles for CentOS-based equivalent container images from <https://github.com/scloorg?q=-container>.



NOTE

Many RHSCl container images include support for *Source-to-Image (S2I)*, best known as an OpenShift Container Platform feature. Having support for S2I does not affect the use of these container images with Docker.

CONTAINER IMAGES IN RED HAT CONTAINER CATALOG (RHCC)

Mission-critical applications require trusted containers. The *Red Hat Container Catalog* is a repository of reliable, tested, certified, and curated collection of container images built on versions of Red Hat Enterprise Linux (RHEL) and related systems. Container images available through RHCC have undergone a quality-assurance process. All components have been rebuilt by Red Hat to avoid known security vulnerabilities. They are upgraded on a regular basis so that they contain

the required version of software even when a new image is not yet available. Using RHCC, you can browse and search for images, and you can access information about each image, such as its version, contents, and usage.

SEARCHING FOR IMAGES USING QUAY.IO

Quay.io is an advanced container repository from CoreOS optimized for team collaboration. You can search for container images using <https://quay.io/search>.

Clicking on an image's name provides access to the image information page, including access to all existing tags for the image, and the command to pull the image.

FINDING DOCKERFILES ON DOCKER HUB

Anyone can create a Docker Hub account and publish container images there. There are no general assurances about quality and security; images on Docker Hub range from professionally supported to one-time experiments. Each image has to be evaluated individually.

After searching for an image, the documentation page might provide a link to its Dockerfile. For example, the first result when searching for **mysql** is the documentation page for the MySQL official image at https://hub.docker.com/_/mysql.

On that page, the link for the 5.6/Dockerfile image points to the **docker-library** GitHub project, which hosts **Dockerfiles** for images built by the Docker community automatic build system.

The direct URL for the Docker Hub MySQL 5.6 **Dockerfile** tree is <https://github.com/docker-library/mysql/blob/master/5.6>.

DESCRIBING HOW TO USE THE OPENSHIFT SOURCE-TO-IMAGE TOOL

Source-to-Image (S2I) provides an alternative to using Dockerfiles to create new container images and can be used either as a feature from OpenShift or as the standalone **s2i** utility. S2I allows developers to work using their usual tools, instead of learning Dockerfile syntax and using operating system commands such as **yum**, and usually creates slimmer images, with fewer layers.

S2I uses the following process to build a custom container image for an application:

1. Start a container from a base container image called the *builder image*, which includes a programming language runtime and essential development tools such as compilers and package managers.
2. Fetch the application source code, usually from a Git server, and send it to the container.
3. Build the application binary files inside the container.
4. Save the container, after some clean up, as a new container image, which includes the programming language runtime and the application binaries.

The builder image is a regular container image following a standard directory structure and providing scripts that are called during the S2I process. Most of these builder images can also be used as base images for Dockerfiles, outside the S2I process.

The **s2i** command is used to run the S2I process outside of OpenShift, in a Docker-only environment. It can be installed on a RHEL system from the *source-to-image* RPM package, and on

other platforms, including Windows and Mac OS, from the installers available in the S2I project on GitHub.



REFERENCES

Red Hat Software Collections Library (RHSCl)

<https://access.redhat.com/documentation/en/red-hat-software-collections/>

Red Hat Container Catalog (RHCC)

<https://access.redhat.com/containers/>

RHSCl Dockerfiles on GitHub

<https://github.com/sclorg?q=-container>

Using Red Hat Software Collections Container Images

<https://access.redhat.com/articles/1752723>

Quay.io

<https://quay.io/search>

Docker Hub

<https://hub.docker.com/>

Docker Library GitHub project

<https://github.com/docker-library>

The S2I GitHub project

<https://github.com/openshift/source-to-image>

► QUIZ

APPROACHES TO CONTAINER IMAGE DESIGN

Choose the correct answers to the following questions:

► 1. Which method for creating container images is recommended by the containers community? (Choose one.)

- a. Run commands inside a basic OS container, commit the container, and save or export it as a new container image.
- b. Run commands from a Dockerfile and push the generated container image to an image registry.
- c. Create the container image layers manually from tar files.
- d. Run the **podman build** command to process a container image description in YAML format.

► 2. What are two advantages of using the standalone S2I process as an alternative to Dockerfiles? (Choose two.)

- a. Requires no additional tools apart from a basic Podman setup.
- b. Creates smaller container images, having fewer layers.
- c. Reuses high-quality builder images.
- d. Automatically updates the child image as the parent image changes (for example, with security fixes).
- e. Creates images compatible with OpenShift, unlike container images created from Docker tools.

► 3. What are two typical scenarios for creating a Dockerfile to build a child image from an existing image? (Choose two.)

- a. Adding new runtime libraries.
- b. Setting constraints to a container's access to the host machine's CPU.
- c. Adding internal libraries to be shared as a single image layer by multiple container images for different applications.

► SOLUTION

APPROACHES TO CONTAINER IMAGE DESIGN

Choose the correct answers to the following questions:

► 1. **Which method for creating container images is recommended by the containers community? (Choose one.)**

- a. Run commands inside a basic OS container, commit the container, and save or export it as a new container image.
- b. Run commands from a Dockerfile and push the generated container image to an image registry.
- c. Create the container image layers manually from tar files.
- d. Run the `podman build` command to process a container image description in YAML format.

► 2. **What are two advantages of using the standalone S2I process as an alternative to Dockerfiles? (Choose two.)**

- a. Requires no additional tools apart from a basic Podman setup.
- b. Creates smaller container images, having fewer layers.
- c. Reuses high-quality builder images.
- d. Automatically updates the child image as the parent image changes (for example, with security fixes).
- e. Creates images compatible with OpenShift, unlike container images created from Docker tools.

► 3. **What are two typical scenarios for creating a Dockerfile to build a child image from an existing image? (Choose two.)**

- a. Adding new runtime libraries.
- b. Setting constraints to a container's access to the host machine's CPU.
- c. Adding internal libraries to be shared as a single image layer by multiple container images for different applications.

BUILDING CUSTOM CONTAINER IMAGES WITH DOCKERFILES

OBJECTIVES

After completing this section, students should be able to create a container image using common Dockerfile commands.

BUILDING BASE CONTAINERS

A **Dockerfile** is a mechanism to automate the building of container images. Building an image from a Dockerfile is a three-step process:

1. Create a working directory
2. Write the **Dockerfile**
3. Build the image with Podman

Create a Working Directory

The *working directory* is the directory containing all files needed to build the image. Creating an empty working directory is good practice to avoid incorporating unnecessary files into the image. For security reasons, the root directory, `/`, should never be used as a working directory for image builds.

Write the Dockerfile Specification

A **Dockerfile** is a text file that must exist in the working directory. This file contains the instructions needed to build the image. The basic syntax of a **Dockerfile** follows:

```
# Comment  
INSTRUCTION arguments
```

Lines that begin with a hash, or pound, symbol (#) are comments. *INSTRUCTION* states for any Dockerfile instruction keyword. Instructions are not case-sensitive, but the convention is to make instructions all uppercase to improve visibility.

The first non-comment instruction must be a **FROM** instruction to specify the base image. Dockerfile instructions are executed into a new container using this image and then committed to a new image. The next instruction (if any) executes into that new image. The execution order of instructions is the order of their appearance in the Dockerfile.



NOTE

The **ARG** instruction can appear before the **FROM** instruction, but **ARG** instructions are outside the objectives for this section.

Each Dockerfile instruction runs in an independent container using an intermediate image built from every previous command. This means each instruction is independent from other instructions in the Dockerfile.

The following is an example Dockerfile for building a simple Apache web server container:

```
# This is a comment line ①
FROM rhel7:7.5 ②
LABEL description="This is a custom httpd container image" ③
MAINTAINER John Doe <jdoe@xyz.com> ④
RUN yum install -y httpd ⑤
EXPOSE 80 ⑥
ENV LogLevel "info" ⑦
ADD http://someserver.com/filename.pdf /var/www/html ⑧
COPY ./src/ /var/www/html/ ⑨
USER apache ⑩
ENTRYPOINT ["/usr/sbin/httpd"] ⑪
CMD ["-D", "FOREGROUND"] ⑫
```

- ① Lines that begin with a hash, or pound, sign (#) are comments.
- ② The **FROM** instruction declares that the new container image extends `rhel7:7.5` container base image. Dockerfiles can use any other container image as a base image, not only images from operating system distributions. Red Hat provides a set of container images that are certified and tested and highly recommends using these container images as a base.
- ③ The **LABEL** is responsible for adding generic metadata to an image. A **LABEL** is a simple key-value pair.
- ④ **MAINTAINER** indicates the **Author** field of the generated container image's metadata. You can use the **podman inspect** command to view image metadata.
- ⑤ **RUN** executes commands in a new layer on top of the current image. The shell that is used to execute commands is **/bin/sh**.
- ⑥ **EXPOSE** indicates that the container listens on the specified network port at runtime. The **EXPOSE** instruction defines metadata only; it does not make ports accessible from the host. The **-p** option in the **podman run** command exposes container ports from the host.
- ⑦ **ENV** is responsible for defining environment variables that are available in the container. You can declare multiple **ENV** instructions within the **Dockerfile**. You can use the **env** command inside the container to view each of the environment variables.
- ⑧ **ADD** instruction copies files or folders from a local or remote source and adds them to the container's file system. If used to copy local files, those must be in the working directory. **ADD** instruction unpacks local **.tar** files to the destination image directory.
- ⑨ **COPY** copies files from the working directory and adds them to the container's file system. It is not possible to copy a remote file using its URL with this Dockerfile instruction.
- ⑩ **USER** specifies the username or the UID to use when running the container image for the **RUN**, **CMD**, and **ENTRYPOINT** instructions. It is a good practice to define a different user other than root for security reasons.
- ⑪ **ENTRYPOINT** specifies the default command to execute when the image runs in a container. If omitted, the default **ENTRYPOINT** is **/bin/sh -c**.
- ⑫ **CMD** provides the default arguments for the **ENTRYPOINT** instruction. If the default **ENTRYPOINT** applies (**/bin/sh -c**), then **CMD** forms an executable command and parameters that run at container start.

CMD AND ENTRYPOINT

ENTRYPOINT and **CMD** instructions have two formats:

- Exec form (using a JSON array):

```
ENTRYPOINT ["command", "param1", "param2"]
CMD ["param1","param2"]
```

- Shell form:

```
ENTRYPOINT command param1 param2
CMD param1 param2
```

Exec form is the preferred form. Shell form wraps the commands in a **/bin/sh -c** shell, creating a sometimes unnecessary shell process. Also, some combinations are not allowed, or may not work as expected. For example, if **ENTRYPOINT** is `["ping"]` (exec form) and **CMD** is `localhost` (shell form), then the expected executed command is `ping localhost`, but the container tries `ping /bin/sh -c localhost`, which is a malformed command.

The **Dockerfile** should contain at most one **ENTRYPOINT** and one **CMD** instruction. If more than one of each is present, then only the last instruction takes effect. **CMD** can be present without specifying an **ENTRYPOINT**. In this case, the base image's **ENTRYPOINT** applies, or the default **ENTRYPOINT** if none is defined.

Podman can override the **CMD** instruction when starting a container. If present, all parameters for the **podman run** command after the image name form the **CMD** instruction. For example, the following instruction causes the running container to display the current time:

```
ENTRYPOINT ["/bin/date", "+%H:%M"]
```

The **ENTRYPOINT** defines both the command to be executed and the parameters. So the **CMD** instruction cannot be used. The following example provides the same functionality, with the added benefit of the **CMD** instruction being overwritable when a container starts:

```
ENTRYPOINT ["/bin/date"]
CMD ["+%H:%M"]
```

In both cases, when a container starts without providing a parameter, the current time is displayed:

```
[student@workstation ~]$ sudo podman run -it do180/rhel
11:41
```

In the second case, if a parameter appears after the image name in the **podman run** command, it overwrites the **CMD** instruction. The following command displays the current day of the week instead of the time:

```
[student@workstation demo-basic]$ sudo podman run -it do180/rhel +%
Tuesday
```

Another approach is using the default **ENTRYPOINT** and the **CMD** instruction to define the initial command. The following instruction displays the current time, with the added benefit of being able to be overridden at run time.

```
CMD ["date", "+%H:%M"]
```

ADD AND COPY

The **ADD** and **COPY** instructions have two forms:

- The Shell form:

```
ADD <source>... <destination>
COPY <source>... <destination>
```

- The Exec form:

```
ADD [<source>, ... "<destination>"]
COPY [<source>, ... "<destination>"]
```

If the **source** is a file system path, it must be inside the working directory.

The **ADD** instruction also allows you to specify a resource using a URL:

```
ADD http://someserver.com/filename.pdf /var/www/html
```

If the **source** is a compressed file, then the **ADD** instruction decompresses the file to the **destination** folder. The **COPY** instruction does not have this functionality.



WARNING

Both the **ADD** and **COPY** instructions copy the files, retaining permissions, with root as the owner, even if the **USER** instruction is specified. Red Hat recommends using a **RUN** instruction after the copy to change the owner and avoid “permission denied” errors.

LAYERING IMAGE

Each instruction in a **Dockerfile** creates a new image layer. Having too many instructions in a **Dockerfile** causes too many layers, resulting in large images. For example, consider the following **RUN** instructions in a **Dockerfile**:

```
RUN yum --disablerepo=* --enablerepo=rhel-7-server-rpms
RUN yum update -y
RUN yum install -y httpd
```

The previous example is not a good practice when creating container images. It creates three layers (one for each **RUN** instruction) while only the last is meaningful. Red Hat recommends minimizing the number of layers. You can achieve the same objective while creating a single layer by using the **&&** conjunction:

```
RUN yum --disablerepo=* --enablerepo=rhel-7-server-rpms && yum update -y && yum
install -y httpd
```

The problem with this approach is that the readability of the **Dockerfile** decays. Use the \ escape code to insert line breaks and improve readability. You can also indent lines to align the commands:

```
RUN yum --disablerepo=* --enablerepo="rhel-7-server-rpms" && \
    yum update -y && \
    yum install -y httpd
```

This example creates only one layer, and the readability improves. **RUN**, **COPY**, and **ADD** instructions create new image layers, but **RUN** can be improved this way.

Red Hat recommends applying similar formatting rules to other instructions accepting multiple parameters, such as **LABEL** and **ENV**:

```
LABEL version="2.0" \
      description="This is an example container image" \
      creationDate="01-09-2017"
```

```
ENV MYSQL_ROOT_PASSWORD="my_password" \
      MYSQL_DATABASE "my_database"
```

BUILDING IMAGES WITH PODMAN

The **podman build** command processes the **Dockerfile** and builds a new image based on the instructions it contains. The syntax for this command is as follows:

```
$ podman build -t NAME:TAG DIR
```

DIR is the path to the working directory, which must include the **Dockerfile**. It can be the current directory as designated by a dot (.) if the working directory is the current directory. *NAME:TAG* is a name with a tag given to the new image. If *TAG* is not specified, then the image is automatically tagged as **latest**.



REFERENCES

Dockerfile Reference Guide

<https://docs.docker.com/engine/reference/builder/>

Creating base images

<https://docs.docker.com/engine/userguide/eng-image/baseimages/>

► GUIDED EXERCISE

CREATING A BASIC APACHE CONTAINER IMAGE

In this exercise, you will create a basic Apache container image.

OUTCOMES

You should be able to create a custom Apache container image built on a Red Hat Enterprise Linux 7.5 image.

BEFORE YOU BEGIN

Run the following command to download the relevant lab files and to verify that Docker is running:

```
[student@workstation ~]$ lab dockerfile-create start
```

► 1. Create the Apache Dockerfile

- 1.1. Open a terminal on **workstation**. Use your preferred editor and create a new Dockerfile:

```
[student@workstation ~]$ vim /home/student/D0180/labs/dockerfile-create/Dockerfile
```

- 1.2. Use RHEL 7.5 as a base image by adding the following **FROM** instruction at the top of the new Dockerfile:

```
FROM rhel7:7.5
```

- 1.3. Below the **FROM** instruction, include the **MAINTAINER** instruction to set the **Author** field in the new image. Replace the values to include your name and email address:

```
MAINTAINER Your Name <youremail>
```

- 1.4. Below the **MAINTAINER** instruction, add the following **LABEL** instruction to add description metadata to the new image:

```
LABEL description="A custom Apache container based on RHEL 7"
```

- 1.5. Insert an **ADD** instruction to copy the local **training.repo** file to the **/etc/yum.repos.d** directory in the container image, which configures a new local repository in the container image:

```
ADD training.repo /etc/yum.repos.d/training.repo
```

- 1.6. Add a **RUN** instruction with a **yum install** command to install Apache on the new container:

```
RUN yum install -y httpd && \
    yum clean all
```

- 1.7. Add a **RUN** instruction to replace contents of the default HTTPD home page:

```
RUN echo "Hello from Dockerfile" > /usr/share/httpd/noindex/index.html
```

- 1.8. Use the **EXPOSE** instruction below the **RUN** instruction to document the port that the container listens to at runtime. In this instance, set the port to 80, because it is the default for an Apache server:

```
EXPOSE 80
```

**NOTE**

The **EXPOSE** instruction does not actually make the specified port available to the host; rather, the instruction serves as metadata about which ports the container is listening on.

- 1.9. At the end of the file, use the following **ENTRYPOINT** instruction to set **httpd** as the default entry point:

```
ENTRYPOINT ["httpd", "-D", "FOREGROUND"]
```

- 1.10. Verify that your Dockerfile matches the following before saving and proceeding with the next steps:

```
FROM rhel7:7.5

MAINTAINER Your Name <youremail>

LABEL description="A basic Apache container on RHEL 7"

ADD training.repo /etc/yum.repos.d/training.repo

RUN yum install -y httpd && \
    yum clean all

RUN echo "Hello from Dockerfile" > /usr/share/httpd/noindex/index.html

EXPOSE 80

ENTRYPOINT ["httpd", "-D", "FOREGROUND"]
```

► 2. Build and verify the Apache container image.

- 2.1. Use the following commands to create a basic Apache container image using the newly created Dockerfile:

```
[student@workstation ~]$ cd /home/student/D0180/labs/dockerfile-create
[student@workstation dockerfile-create]$ sudo podman build --layers=false \
> -t do180/apache .
STEP 1: FROM rhel7:7.5
Getting image source signatures ①
Copying blob sha256:...output omitted...
71.46 MB / 71.46 MB [=====] 18s
...output omitted...
Storing signatures
STEP 2: MAINTAINER username <username@example.com>
STEP 3: LABEL description="A basic Apache container on RHEL 7"
STEP 4: ADD training.repo /etc/yum.repos.d/training.repo
STEP 5: RUN yum install -y && httpd yum clean all
Loaded plugins: ovl, product-id, search-disabled-repos, subscription-manager
...output omitted...
Complete!
STEP 6: RUN echo "Hello from Dockerfile" > /usr/share/httpd/noindex/index.html
STEP 7: EXPOSE 80
STEP 8: ENTRYPOINT ["httpd", "-D", "FOREGROUND"]

ERRO[0109] HOSTNAME is not supported for OCI image format...output omitted... ②
STEP 9: COMMIT ...output omitted... localhost/do180/apache:latest
Getting image source signatures
...output omitted...
Storing signatures
--> 190a86564a35bbc4fc7be79a4a740fe6f0edf0eb7cf363bd1ff6c246096895c5
```

- ① The container image listed in the **FROM** instruction is only downloaded if not already present in local storage.
- ② This error is benign, and can be ignored. It is a known issue of Podman (see Bug 1634806 [https://bugzilla.redhat.com/show_bug.cgi?id=1634806]) and will disappear in later versions.

- 2.2. After the build process has finished, run **podman images** to see the new image in the image repository:

```
[student@workstation dockerfile-create]$ sudo podman images
REPOSITORY          TAG      IMAGE ID      CREATED        SIZE
localhost/do180/apache    latest   190a86564a35  About a minute ago  271MB
registry.lab.example.com/rhel7  7.5     e64297b706b7  7 months ago   211MB
```



NOTE

Podman creates many anonymous intermediate images during the build process. They are not be listed unless **-a** is used. Use the **--layers=false** option of **build** subcommand to instruct Podman to delete intermediate images.

► 3. Run the Apache Container

3.1. Use the following command to run a container using the Apache image:

```
[student@workstation dockerfile-create]$ sudo podman run --name lab-apache \
> -d -p 10080:80 do180/apache
fa1d1c450e8892ae085dd8bbf763edac92c41e6ffaa7ad6ec6388466809bb391
```

3.2. Run the **podman ps** command to see the running container:

```
[student@workstation dockerfile-create]$ sudo podman ps
CONTAINER ID IMAGE COMMAND ...output omitted...
fa1d1c450e88 localhost/do180/apache:latest httpd -D FOREGROU...output omitted...
```

3.3. Use the **curl** command to verify that the server is running:

```
[student@workstation dockerfile-create]$ curl 127.0.0.1:10080
Hello from Dockerfile
```

Finish

On workstation, run the **lab dockerfile-create finish** script to complete this lab.

```
[student@workstation ~]$ lab dockerfile-create finish
```

This concludes the guided exercise.

► LAB

CREATING CUSTOM CONTAINER IMAGES

PERFORMANCE CHECKLIST

In this lab, you will create a Dockerfile to build a custom Apache Web Server container image. The custom image will be based on a RHEL 7.5 image and serve a custom **index.html** page.

OUTCOMES

You should be able to create a custom Apache Web Server container that hosts static HTML files.

BEFORE YOU BEGIN

Open a terminal on workstation as the student user and run the following command:

```
[student@workstation ~]$ lab dockerfile-review start
```

1. Review the provided Dockerfile stub in the **/home/student/D0180/labs/dockerfile-review/** folder. Edit the **Dockerfile** and ensure that it meets the following specifications:
 - The base image is **rhel7:7.5**
 - Sets the desired author name and email ID with the **MAINTAINER** instruction
 - Sets the environment variable **PORT** to 8080
 - Install Apache (**httpd** package) using the classroom Yum repository. Also ensure that you run the **yum clean all** command as a best practice.
 - Change the Apache configuration file **/etc/httpd/conf/httpd.conf** to listen to port 8080 instead of the default port 80.
 - Change ownership of the **/etc/httpd/logs** and **/run/httpd** folders to user and group apache (UID and GID are 48).
 - So that container users know how to access the Apache Web Server, expose the value set in the **PORT** environment variable.
 - Copy the contents of the **src/** folder in the lab directory to the Apache **DocumentRoot** file (**/var/www/html/**) inside the container.

The **src** folder contains a single **index.html** file that prints a **Hello World!** message.

- Start the Apache **httpd** daemon in the foreground using the following command:

```
httpd -D FOREGROUND
```

2. Build the custom Apache image with the name **d0180/custom-apache**.

3. Create a new container in detached mode with the following characteristics:

- Name: **dockerfile**
- Container image: **do180/custom-apache**
- Port forward: from host port 20080 to container port 8080
- Run as a daemon: yes

Verify that the container is ready and running.

4. Verify that the server is serving the HTML file.

Evaluation

Grade your work by running the **lab dockerfile-review grade** command from your workstation machine. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab dockerfile-review grade
```

Finish

From workstation, run the **lab dockerfile-review finish** command to complete this lab.

```
[student@workstation ~]$ lab dockerfile-review finish
```

This concludes the lab.

► SOLUTION

CREATING CUSTOM CONTAINER IMAGES

PERFORMANCE CHECKLIST

In this lab, you will create a Dockerfile to build a custom Apache Web Server container image. The custom image will be based on a RHEL 7.5 image and serve a custom **index.html** page.

OUTCOMES

You should be able to create a custom Apache Web Server container that hosts static HTML files.

BEFORE YOU BEGIN

Open a terminal on workstation as the student user and run the following command:

```
[student@workstation ~]$ lab dockerfile-review start
```

1. Review the provided Dockerfile stub in the **/home/student/D0180/labs/dockerfile-review/** folder. Edit the **Dockerfile** and ensure that it meets the following specifications:
 - The base image is **rhel7:7.5**
 - Sets the desired author name and email ID with the **MAINTAINER** instruction
 - Sets the environment variable **PORT** to 8080
 - Install Apache (**httpd** package) using the classroom Yum repository. Also ensure that you run the **yum clean all** command as a best practice.
 - Change the Apache configuration file **/etc/httpd/conf/httpd.conf** to listen to port 8080 instead of the default port 80.
 - Change ownership of the **/etc/httpd/logs** and **/run/httpd** folders to user and group apache (UID and GID are 48).
 - So that container users know how to access the Apache Web Server, expose the value set in the **PORT** environment variable.
 - Copy the contents of the **src/** folder in the lab directory to the Apache **DocumentRoot** file (**/var/www/html/**) inside the container.

The **src** folder contains a single **index.html** file that prints a **Hello World!** message.

- Start the Apache **httpd** daemon in the foreground using the following command:

```
httpd -D FOREGROUND
```

- 1.1. Open a terminal (Applications → System Tools → Terminal) and use your preferred editor to modify the Dockerfile located in the **/home/student/D0180/labs/dockerfile-review/** folder.

```
[student@workstation ~]$ cd /home/student/D0180/labs/dockerfile-review/  
[student@workstation dockerfile-review]$ vim Dockerfile
```

- 1.2. Set the base image for the Dockerfile to **rhel7:7.5**.

```
FROM rhel7:7.5
```

- 1.3. Set your name and email with a **MAINTAINER** instruction.

```
MAINTAINER Your Name <youremail>
```

- 1.4. Create an environment variable called **PORT** and set it to 8080.

```
ENV PORT 8080
```

- 1.5. Add the classroom Yum repositories configuration file. Install Apache and clean any cached packages or metadata with a single **RUN** instruction.

```
ADD training.repo /etc/yum.repos.d/training.repo  
  
RUN yum install -y httpd && \  
    yum clean all
```

- 1.6. Change the Apache HTTP Server configuration file to listen to port 8080 and change ownership of the server working folders with a single **RUN** instruction.

```
RUN sed -ri -e "/^Listen 80/c\Listen ${PORT}" /etc/httpd/conf/httpd.conf && \  
    chown -R apache:apache /etc/httpd/logs/ && \  
    chown -R apache:apache /run/httpd/
```

- 1.7. Use the **USER** instruction to run the container as the apache user. Use the **EXPOSE** instruction to document the port that the container listens to at runtime. In this instance, set the port to the **PORT** environment variable, which is the default for an Apache server.

```
USER apache
```

```
# Expose the custom port that you provided in the ENV var  
EXPOSE ${PORT}
```

- 1.8. Copy all the files from the **src** folder to the Apache **DocumentRoot** path at **/var/www/html**.

```
# Copy all files under src/ folder to Apache DocumentRoot (/var/www/html)
```

```
COPY ./src/ /var/www/html/
```

- 1.9. Finally, insert a **CMD** instruction to run **httpd** in the foreground, and then save the Dockerfile.

```
# Start Apache in the foreground
CMD ["httpd", "-D", "FOREGROUND"]
```

2. Build the custom Apache image with the name **do180/custom-apache**.

- 2.1. Verify the Dockerfile for the custom Apache image.

The Dockerfile for the custom Apache image should look like the following:

```
FROM rhel7:7.5

MAINTAINER Your Name <youremail>

ENV PORT 8080

ADD training.repo /etc/yum.repos.d/training.repo

RUN yum install -y httpd && \
    yum clean all

RUN sed -ri -e "/^Listen 80/c\Listen ${PORT}" /etc/httpd/conf/httpd.conf && \
    chown -R apache:apache /etc/httpd/logs/ && \
    chown -R apache:apache /run/httpd/

USER apache

# Expose the custom port that you provided in the ENV var
EXPOSE ${PORT}

# Copy all files under src/ folder to Apache DocumentRoot (/var/www/html)
COPY ./src/ /var/www/html/

# Start Apache in the foreground
CMD ["httpd", "-D", "FOREGROUND"]
```

- 2.2. Run a **sudo podman build** command to build the custom Apache image and name it **do180/custom-apache**.

```
[student@workstation dockerfile-review]$ sudo podman build \
> -t do180/custom-apache .
STEP 1: FROM rhel7:7.5
...output omitted...
STEP 2: MAINTAINER username <username@example.com>
...output omitted...
--> b94c4cb26414292787a67c53da1ba4dac1387e57b51e937243c69534f69107b3
STEP 3: FROM b94c4cb26414292787a67c53da1ba4dac1387e57b51e937243c69534f69107b3
STEP 4: ENV PORT 8080
--> cee771456b94af02bbd752452a18852b9570873c2e6eb0d2dcba59cd6a62...
STEP 5: FROM cee771456b94af02bbd752452a18852b9570873c2e6eb0d2dcba59cd6a6215eb
```

```
STEP 6: ADD training.repo /etc/yum.repos.d/training.repo
--> 7fb63e83428db0ec63621b0bd12c9c20602d6594e17dc1e6c7b5fa283d677c46
STEP 7: FROM 7fb63e83428db0ec63621b0bd12c9c20602d6594e17dc1e6c7b5fa283d677c46
STEP 8: RUN yum install -y httpd && yum clean all
...output omitted...

Installed:
httpd.x86_64 0:2.4.6-88.el7

...output omitted...
STEP 16: COPY ./src/ /var/www/html/
--> 589e6f6e82e8506e17872f7464b726fd0f7e79c0b703337b4a6c3b41bc03705b
STEP 17: FROM 589e6f6e82e8506e17872f7464b726fd0f7e79c0b703337b4a6c3b41bc03705b
STEP 18: CMD ["httpd", "-D", "FOREGROUND"]
--> a9c102b521f1872116bcc18149b7157aab6a169ed18952909e97b852da7cb2e1
STEP 19: COMMIT do180/custom-apache
```

- 2.3. Run the **podman images** command to verify that the custom image is built successfully.

```
[student@workstation dockerfile-review]$ sudo podman images
REPOSITORY                                TAG      IMAGE ID      ...
localhost/do180/custom-apache              latest   a9c102b521f1 ...
registry.lab.example.com/rhel7             7.5     e64297b706b7 ...
```

3. Create a new container in detached mode with the following characteristics:

- Name: **dockerfile**
- Container image: **do180/custom-apache**
- Port forward: from host port 20080 to container port 8080
- Run as a daemon: yes

Verify that the container is ready and running.

- 3.1. Create and run the container.

```
[student@workstation dockerfile-review]$ sudo podman run -d \
> --name dockerfile -p 20080:8080 do180/custom-apache
367823e35c4a...
```

- 3.2. Verify that the container is ready and running.

```
[student@workstation dockerfile-review]$ sudo podman ps
... IMAGE          COMMAND          ... PORTS          NAMES
... do180/custom... "httpd -D ..." ... 0.0.0.0:20080->8080/tcp  dockerfile
```

- Verify that the server is serving the HTML file.

Run a **curl** command on **127.0.0.1:20080**

```
[student@workstation dockerfile-review]$ curl 127.0.0.1:20080
```

The output should be as follows:

```
<html>
<header><title>D0180 Hello!</title></header>
<body>
  Hello World! The dockerfile-review lab works!
</body>
</html>
```

Evaluation

Grade your work by running the **lab dockerfile-review grade** command from your workstation machine. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab dockerfile-review grade
```

Finish

From workstation, run the **lab dockerfile-review finish** command to complete this lab.

```
[student@workstation ~]$ lab dockerfile-review finish
```

This concludes the lab.

SUMMARY

In this chapter, you learned:

- A **Dockerfile** contains instructions that specify how to construct a container image.
- Container images provided by Red Hat Container Catalog or Quay.io are a good starting point for creating custom images for a specific language or technology.
- Building an image from a Dockerfile is a three-step process:
 1. Create a working directory.
 2. Specify the build instructions in a **Dockerfile** file.
 3. Build the image with the **podman build** command.
- The Source-to-Image (S2I) process provides an alternative to Dockerfiles. S2I implements a standardized container image build process for common technologies from application source code. This allows developers to focus on application development and not Dockerfile development.

CHAPTER 6

DEPLOYING CONTAINERIZED APPLICATIONS ON OPENSHIFT

GOAL

Deploy single container applications on OpenShift Container Platform.

OBJECTIVES

- Describe the architecture of Kubernetes and Red Hat OpenShift Container Platform.
- Create standard Kubernetes resources.
- Create a route to a service.
- Build an application using the Source-to-Image facility of OpenShift Container Platform.
- Create an application using the OpenShift web console.

SECTIONS

- Describing Kubernetes and OpenShift Architecture (and Quiz)
- Creating Kubernetes Resources (and Guided Exercise)
- Creating Routes (and Guided Exercise)
- Creating Applications with the Source-to-Image Facility (and Guided Exercise)
- Creating Applications with the OpenShift Web Console (and Guided Exercise)

LAB

- Deploying Containerized Applications on OpenShift

DESCRIBING KUBERNETES AND OPENSHIFT ARCHITECTURE

OBJECTIVES

After completing this section, students should be able to:

- Describe the architecture of a Kubernetes cluster running on the Red Hat OpenShift Container Platform (RHOCP).
- List the main resource types provided by Kubernetes and RHOCP.
- Identify the network characteristics of containers, Kubernetes, and RHOCP.
- List mechanisms to make a pod externally available.

KUBERNETES AND OPENSIFT

In previous chapters we saw that Kubernetes is an orchestration service that simplifies the deployment, management, and scaling of containerized applications. One of the main advantages of using Kubernetes is that it uses several nodes to ensure the resiliency and scalability of its managed applications. Kubernetes forms a cluster of node servers that run containers and are centrally managed by a set of master servers. A server can act as both a server and a node, but those roles are usually segregated for increased stability.

Kubernetes Terminology

TERM	DEFINITION
Node	A server that hosts applications in a Kubernetes cluster.
Master Node	A node server that manages the control plane in a Kubernetes cluster. Master nodes provide basic cluster services such as APIs or controllers.
Worker Node	Also named Compute Node , worker nodes execute workloads for the cluster. Application pods are scheduled onto worker nodes.
Resource	Resources are any kind of component definition managed by Kubernetes. Resources contain the configuration of the managed component (for example, the role assigned to a node), and the current state of the component (for example, if the node is available).
Controller	A controller is a Kubernetes process that watches resources and makes changes attempting to move the current state towards the desired state.
Label	A key-value pair that can be assigned to any Kubernetes resource. Selectors use labels to filter eligible resources for scheduling and other operations.
Namespace	A scope for Kubernetes resources and processes, so that resources with the same name can be used in different boundaries.

**NOTE**

The latest Kubernetes versions implement many controllers as *Operators*. Operators are Kubernetes plug-in components that can react to cluster events and control the state of resources. Operators and CoreOS Operator Framework are outside the scope of this document.

Red Hat OpenShift Container Platform is a set of modular components and services built on top of Red Hat CoreOS and Kubernetes. RHOP adds PaaS capabilities such as remote management, increased security, monitoring and auditing, application life-cycle management, and self-service interfaces for developers.

An OpenShift cluster is a Kubernetes cluster that can be managed the same way, but using the management tools provided by OpenShift, such as the command-line interface or the web console. This allows for more productive workflows and makes common tasks much easier.

OpenShift Terminology

TERM	DEFINITION
Infra Node	A node server containing infrastructure services like monitoring, logging, or external routing.
Console	A web UI provided by the RHOP cluster that allows developers and administrators to interact with cluster resources.
Project	OpenShift's extension of Kubernetes' namespaces. Allows the definition of user access control (UAC) to resources.

The following schema illustrates the OpenShift Container Platform stack.

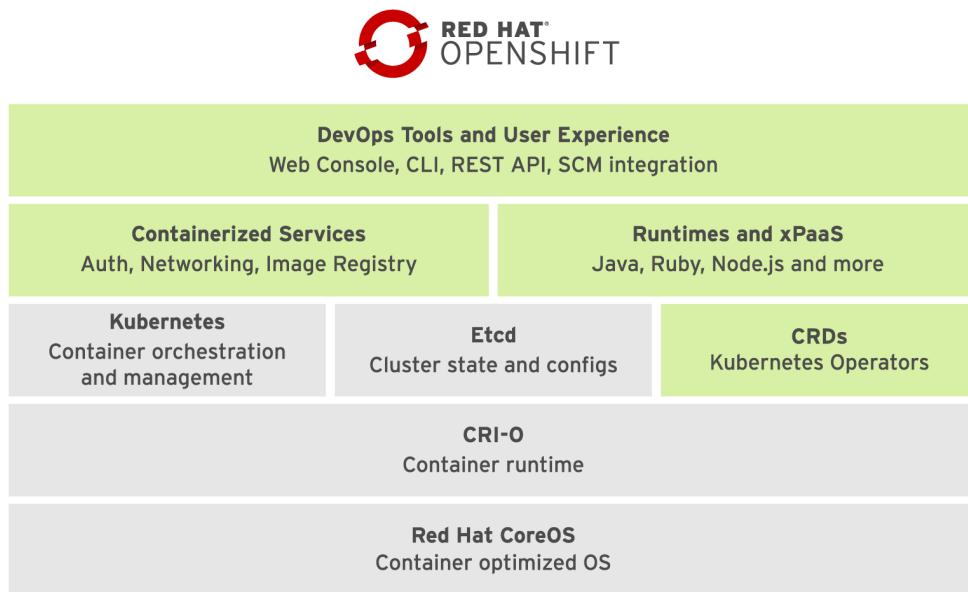


Figure 6.1: OpenShift component stack

From bottom to top, and from left to right, this shows the basic container infrastructure, integrated and enhanced by Red Hat:

- The base OS is Red Hat CoreOS. Red Hat CoreOS is a Linux distribution focused on providing an immutable operating system for container execution.
- CRI-O is an implementation of the Kubernetes CRI (Container Runtime Interface) to enable using OCI (Open Container Initiative) compatible runtimes. CRI-O can use any container runtime that satisfies CRI: **runc** (used by the Docker service), **libpod** (used by Podman) or **rkt** (from CoreOS).
- Kubernetes manages a cluster of hosts, physical or virtual, that run containers. It uses *resources* that describe multicontainer applications composed of multiple resources, and how they interconnect.
- *Etcd* is a distributed key-value store, used by Kubernetes to store configuration and state information about the containers and other resources inside the Kubernetes cluster.
- *Custom Resource Definitions (CRDs)* are resource types stored in Etcd and managed by Kubernetes. These resource types form the state and configuration of all resources managed by OpenShift.
- Containerized services fulfill many PaaS infrastructure functions, such as networking and authorization. RHOCP uses the basic container infrastructure from Kubernetes and the underlying container runtime for most internal functions. That is, most RHOCP internal services run as containers orchestrated by Kubernetes.
- Runtimes and xPaaS are base container images ready for use by developers, each preconfigured with a particular runtime language or database. The xPaaS offering is a set of base images for Red Hat middleware products such as JBoss EAP and ActiveMQ. *Red Hat OpenShift Application Runtimes (RHOAR)* are a set runtimes optimized for cloud native applications in OpenShift. The application runtimes available are Red Hat JBoss EAP, OpenJDK, Thorntail, Eclipse Vert.x, Spring Boot, and Node.js.
- DevOps tools and user experience: RHOCP provides web UI and CLI management tools for managing user applications and RHOCP services. The OpenShift web UI and CLI tools are built from REST APIs which can be used by external tools such as IDEs and CI platforms.

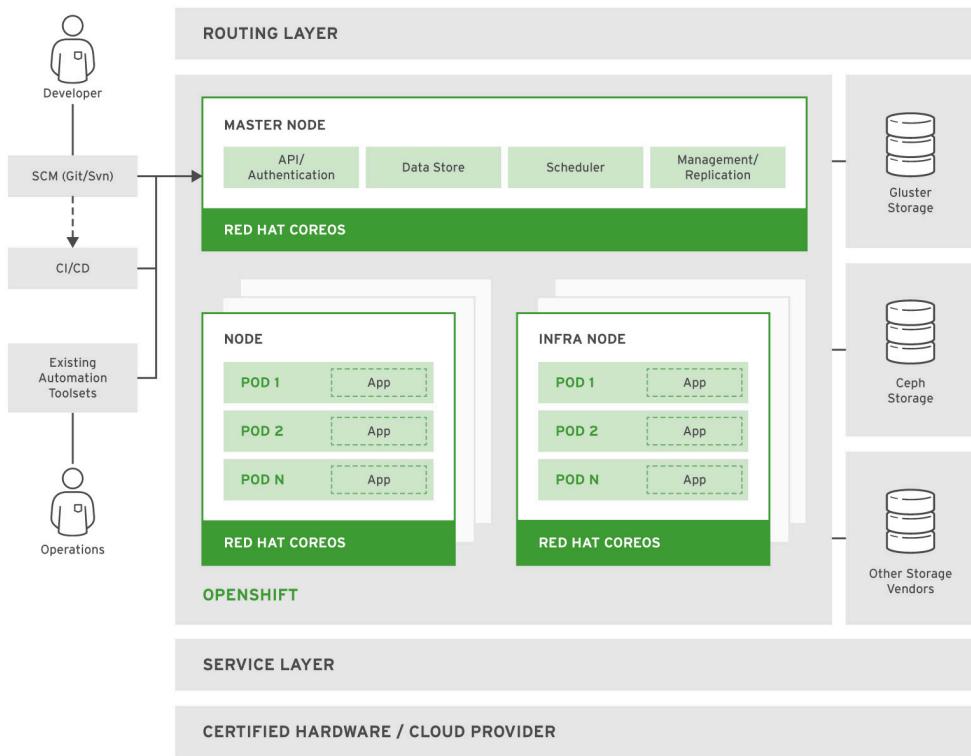


Figure 6.2: OpenShift and Kubernetes architecture

NEW FEATURES IN RHOC 4

RHOC 4 is a massive change from previous versions. As well as keeping backwards compatibility with previous releases, it includes new features, such as:

- CoreOS as the mandatory operating system for all nodes, offering an immutable infrastructure optimized for containers.
- A brand new cluster installer which guides the process of installation and update.
- A self-managing platform, able to automatically apply cluster updates and recoveries without disruption.
- A redesigned application life-cycle management.
- An Operator SDK to build, test, and package Operators.

DESCRIBING KUBERNETES RESOURCE TYPES

Kubernetes has six main resource types that can be created and configured using a YAML or a JSON file, or using OpenShift management tools:

Pods (po)

Represent a collection of containers that share resources, such as IP addresses and persistent storage volumes. It is the basic unit of work for Kubernetes.

Services (svc)

Define a single IP/port combination that provides access to a pool of pods. By default, services connect clients to pods in a round-robin fashion.

Replication Controllers (rc)

A Kubernetes resource that defines how pods are replicated (horizontally scaled) into different nodes. Replication controllers are a basic Kubernetes service to provide high availability for pods and containers.

Persistent Volumes (pv)

Define storage areas to be used by Kubernetes pods.

Persistent Volume Claims (pvc)

Represent a request for storage by a pod. PVCs links a PV to a pod so its containers can make use of it, usually by mounting the storage into the container's file system.

ConfigMaps (cm) and Secrets

Contains a set of keys and values that can be used by other resources. ConfigMaps and Secrets are usually used to centralize configuration values used by several resources. Secrets differ from ConfigMaps maps in that Secrets' values are always encoded (not encrypted) and their access is restricted to fewer authorized users.



NOTE

For the purpose of this course, the PVs are provisioned on local storage, not on networked storage. This is a valid approach for development purposes, but it is not a recommended approach for a production environment.

Although Kubernetes pods can be created standalone, they are usually created by high-level resources such as replication controllers.

OPENSIFT RESOURCE TYPES

The main resource types added by OpenShift Container Platform to Kubernetes are as follows:

Deployment config (dc)

Represents the set of containers included in a pod, and the deployment strategies to be used. A **dc** also provides a basic but extensible continuous delivery workflow.

Build config (bc)

Defines a process to be executed in the OpenShift project. Used by the OpenShift *Source-to-Image (S2I)* feature to build a container image from application source code stored in a Git repository. A **bc** works together with a **dc** to provide a basic but extensible continuous integration and continuous delivery workflows.

Routes

Represent a DNS host name recognized by the OpenShift router as an ingress point for applications and microservices.



NOTE

To obtain a list of all the resources available in a RHOCP cluster and their abbreviations, use the **oc api-resources** or **kubectl api-resources** commands.

Although Kubernetes replication controllers can be created standalone in OpenShift, they are usually created by higher-level resources such as deployment controllers.

NETWORKING

Each container deployed in a Kubernetes cluster has an IP address assigned from an internal network that is accessible only from the node running the container. Because of the container's ephemeral nature, IP addresses are constantly assigned and released.

Kubernetes provides a *software-defined network (SDN)* that spawns the internal container networks from multiple nodes and allows containers from any pod, inside any host, to access pods from other hosts. Access to the SDN only works from inside the same Kubernetes cluster.

Containers inside Kubernetes pods should not connect to each other's dynamic IP address directly. Services resolves this problem by linking more stable IP addresses from the SDN to the pods. If pods are restarted, replicated, or rescheduled to different nodes, services are updated, providing scalability and fault tolerance.

External access to containers is more complicated. Kubernetes services can specify a **NodePort** attribute, which is a network port redirected by all the cluster nodes to the SDN. Then, the containers in the node can redirect a port to the node's port. Unfortunately, none of these approaches scale well.

OpenShift makes external access to containers both scalable and simpler by defining *route* resources. A route defines external-facing DNS names and ports for a service. A router (ingress controller) forwards HTTP and TLS requests to the service addresses inside the Kubernetes SDN. The only requirement is that the desired DNS names are mapped to the IP addresses of the RHOPC router nodes.



REFERENCES

Kubernetes documentation website

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

OpenShift documentation website

<https://docs.openshift.com/>

CoreOS Operators and Operator Framework

<https://coreos.com/operators/>

► QUIZ

DESCRIBING KUBERNETES AND OPENSHIFT

Choose the correct answers to the following questions:

► 1. Which two sentences are correct regarding Kubernetes architecture? (Choose two.)

- a. Kubernetes nodes can be managed without a master.
- b. Kubernetes masters manage pod scaling.
- c. Kubernetes masters schedule pods to specific nodes.
- d. Kubernetes tools cannot be used to manage resources in an OpenShift cluster.
- e. Containers created from Kubernetes pods cannot be managed using standalone tools such as Podman.

► 2. Which two sentences are correct regarding Kubernetes and OpenShift resource types? (Choose two.)

- a. A pod is responsible for provisioning its own persistent storage.
- b. All pods generated from the same replication controller have to run in the same node.
- c. A route is responsible for providing IP addresses for external access to pods.
- d. A replication controller is responsible for monitoring and maintaining the number of pods for a particular application.

► 3. Which two statements are true regarding Kubernetes and OpenShift networking? (Choose two.)

- a. A Kubernetes service can provide an IP address to access a set of pods.
- b. Kubernetes is responsible for providing internal IP addresses for each container.
- c. Kubernetes is responsible for providing a fully qualified domain name for a pod.
- d. A replication controller is responsible for routing external requests to the pods.
- e. A route is responsible for providing DNS names for external access.

► 4. Which statement is correct regarding persistent storage in OpenShift and Kubernetes?

- a. A PVC represents a storage area that a pod can use to store data and is provisioned by the application developer.
- b. A PVC represents a storage area that can be requested by a pod to store data but is provisioned by the cluster administrator.
- c. A PVC represents the amount of memory that can be allocated to a node, so that a developer can state how much memory he requires for his application to run.
- d. A PVC represents the number of CPU processing units that can be allocated to an application pod, subject to a limit managed by the cluster administrator.

► **5. Which statement is correct regarding OpenShift additions to Kubernetes?**

- a. OpenShift adds features to simplify Kubernetes configuration of many real-world use cases.
- b. Container images created for OpenShift cannot be used with plain Kubernetes.
- c. Red Hat maintains forked versions of Kubernetes internal to the RHOCUP product.
- d. Doing continuous integration and continuous deployment with RHOCUP requires external tools.

► SOLUTION

DESCRIBING KUBERNETES AND OPENSHIFT

Choose the correct answers to the following questions:

► 1. Which two sentences are correct regarding Kubernetes architecture? (Choose two.)

- a. Kubernetes nodes can be managed without a master.
- b. Kubernetes masters manage pod scaling.
- c. Kubernetes masters schedule pods to specific nodes.
- d. Kubernetes tools cannot be used to manage resources in an OpenShift cluster.
- e. Containers created from Kubernetes pods cannot be managed using standalone tools such as Podman.

► 2. Which two sentences are correct regarding Kubernetes and OpenShift resource types? (Choose two.)

- a. A pod is responsible for provisioning its own persistent storage.
- b. All pods generated from the same replication controller have to run in the same node.
- c. A route is responsible for providing IP addresses for external access to pods.
- d. A replication controller is responsible for monitoring and maintaining the number of pods for a particular application.

► 3. Which two statements are true regarding Kubernetes and OpenShift networking? (Choose two.)

- a. A Kubernetes service can provide an IP address to access a set of pods.
- b. Kubernetes is responsible for providing internal IP addresses for each container.
- c. Kubernetes is responsible for providing a fully qualified domain name for a pod.
- d. A replication controller is responsible for routing external requests to the pods.
- e. A route is responsible for providing DNS names for external access.

► 4. Which statement is correct regarding persistent storage in OpenShift and Kubernetes?

- a. A PVC represents a storage area that a pod can use to store data and is provisioned by the application developer.
- b. A PVC represents a storage area that can be requested by a pod to store data but is provisioned by the cluster administrator.
- c. A PVC represents the amount of memory that can be allocated to a node, so that a developer can state how much memory he requires for his application to run.
- d. A PVC represents the number of CPU processing units that can be allocated to an application pod, subject to a limit managed by the cluster administrator.

► **5. Which statement is correct regarding OpenShift additions to Kubernetes?**

- a. OpenShift adds features to simplify Kubernetes configuration of many real-world use cases.
- b. Container images created for OpenShift cannot be used with plain Kubernetes.
- c. Red Hat maintains forked versions of Kubernetes internal to the RHOC product.
- d. Doing continuous integration and continuous deployment with RHOC requires external tools.

CREATING KUBERNETES RESOURCES

OBJECTIVES

After completing this section, students should be able to create standard Kubernetes resources.

THE RED HAT OPENSHIFT CONTAINER PLATFORM (RHOCP) COMMAND-LINE TOOL

The main method of interacting with an RHOCP cluster is using the **oc** command. The basic usage of the command is through its subcommands in the following syntax:

```
$> oc <command>
```

Before interacting with a cluster, most operations require a logged-in user. The syntax to log in is shown below:

```
$> oc login <clusterUrl>
```

DESCRIBING POD RESOURCE DEFINITION SYNTAX

RHOCP runs containers inside Kubernetes pods, and to create a pod from a container image, OpenShift needs a *pod resource definition*. This can be provided either as a JSON or YAML text file, or can be generated from defaults by the **oc new-app** command or the OpenShift web console.

A pod is a collection of containers and other resources. An example of a WildFly application server pod definition in YAML format is shown below:

```
apiVersion: v1
kind: Pod1
metadata:
  name: wildfly2
  labels:
    name: wildfly3
spec:
  containers:
    - resources:
        limits:
          cpu: 0.5
      image: do276/todojee
      name: wildfly
      ports:
        - containerPort: 80804
          name: wildfly
    env:5
      - name: MYSQL_ENV_MYSQL_DATABASE
```

```

    value: items
- name: MYSQL_ENV_MYSQL_USER
  value: user1
- name: MYSQL_ENV_MYSQL_PASSWORD
  value: mypa55

```

- ➊ Declares a Kubernetes pod resource type.
- ➋ A unique name for a pod in Kubernetes that allows administrators to run commands on it.
- ➌ Creates a label with a key named **name** that other resources in Kubernetes, usually as service, can use to find it.
- ➍ A container-dependent attribute identifying which port on the container is exposed.
- ➎ Defines a collection of environment variables.

Some pods may require environment variables that can be read by a container. Kubernetes transforms all the **name** and **value** pairs to environment variables. For instance, the **MYSQL_ENV_MYSQL_USER** variable is declared internally by the Kubernetes runtime with a value of **user1**, and is forwarded to the container image definition. Because the container uses the same variable name to get the user's login, the value is used by the WildFly container instance to set the username that accesses a MySQL database instance.

DESCRIBING SERVICE RESOURCE DEFINITION SYNTAX

Kubernetes provides a virtual network that allows pods from different workers to connect. But, Kubernetes provides no easy way for a pod to discover the IP addresses of other pods.

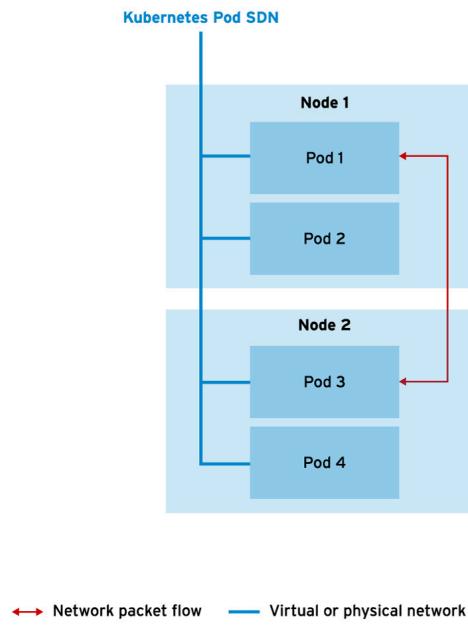


Figure 6.3: Basic Kubernetes networking

Services are essential resources to any OpenShift application. They allow containers in one pod to open network connections to containers in another pod. A pod can be restarted for many reasons, and it gets a different internal IP address each time. Instead of a pod having to discover the IP address of another pod after each restart, a service provides a stable IP address for other pods to use, no matter what worker node runs the pod after each restart.

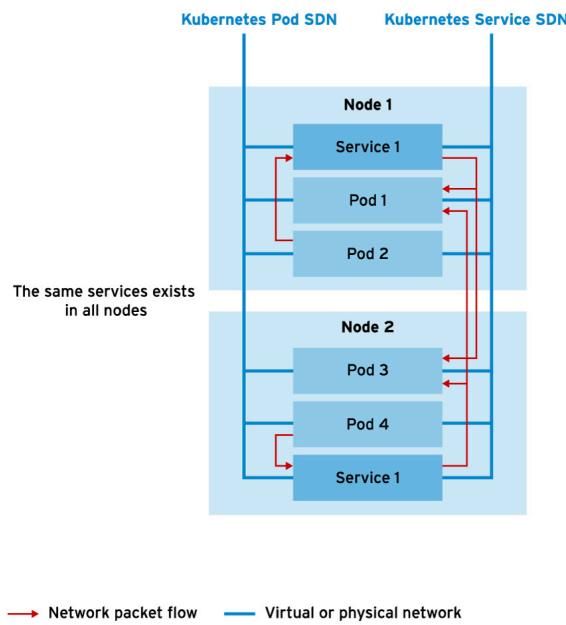


Figure 6.4: Kubernetes services networking

Most real-world applications do not run as a single pod. They need to scale horizontally, so many pods run the same containers from the same pod resource definition to meet growing user demand. A service is tied to a set of pods, providing a single IP address for the whole set, and a load-balancing client request among member pods.

The set of pods running behind a service is managed by a *DeploymentConfig* resource. A *DeploymentConfig* resource embeds a *ReplicationController* that manages how many pod copies (replicas) have to be created, and creates new ones if any of them fail. *DeploymentConfig* and *ReplicationController* resources are explained later in this chapter.

The following example shows a minimal service definition in JSON syntax:

```
{
  "kind": "Service", ①
  "apiVersion": "v1",
  "metadata": {
    "name": "quotedb" ②
  },
  "spec": {
    "ports": [ ③
      {
        "port": 3306,
        "targetPort": 3306
      }
    ],
    "selector": {
      "name": "mysql ldb" ④
    }
  }
}
```

- ➊ The kind of Kubernetes resource. In this case, a Service.
- ➋ A unique name for the service.
- ➌ `ports` is an array of objects that describes network ports exposed by the service. The **targetPort** attribute has to match a `containerPort` from a pod container definition, and the **port** attribute is the port that is exposed by the service. Clients connect to the service port and the service forwards packets to the pod **targetPort**.
- ➍ `selector` is how the service finds pods to forward packets to. The target pods need to have matching labels in their metadata attributes. If the service finds multiple pods with matching labels, it load balances network connections between them.

Each service is assigned a unique IP address for clients to connect to. This IP address comes from another internal OpenShift SDN, distinct from the pods' internal network, but visible only to pods. Each pod matching the `selector` is added to the service resource as an endpoint.

DISCOVERING SERVICES

An application typically finds a service IP address and port by using environment variables. For each service inside an OpenShift project, the following environment variables are automatically defined and injected into containers for all pods inside the same project:

- `SVC_NAME_SERVICE_HOST` is the service IP address.
- `SVC_NAME_SERVICE_PORT` is the service TCP port.



NOTE

The `SVC_NAME` part of the variable is changed to comply with DNS naming restrictions: letters are capitalized and underscores (`_`) are replaced by dashes (`-`).

Another way to discover a service from a pod is by using the OpenShift internal DNS server, which is visible only to pods. Each service is dynamically assigned an SRV record with an FQDN of the form:

`SVC_NAME.PROJECT_NAME.svc.cluster.local`

When discovering services using environment variables, a pod has to be created and started only after the service is created. If the application was written to discover services using DNS queries, however, it can find services created after the pod was started.

For applications that need access to the service outside the OpenShift cluster, there are two ways to achieve this objective:

1. **NodePort type:** This is an older Kubernetes-based approach, where the service is exposed to external clients by binding to available ports on the worker node host, which then proxies connections to the service IP address. Use the `oc edit svc` command to edit service attributes and specify **NodePort** as the value for `type`, and provide a port value for the `nodePort` attribute. OpenShift then proxies connections to the service via the public IP address of the worker node host and the port value set in `nodePort`.
2. **OpenShift Routes:** This is the preferred approach in OpenShift to expose services using a unique URL. Use the `oc expose` command to expose a service for external access or expose a service from the OpenShift web console.

Figure 6.5 illustrates how NodePort services allow external access to Kubernetes services. OpenShift routes are covered in more detail later in this course.

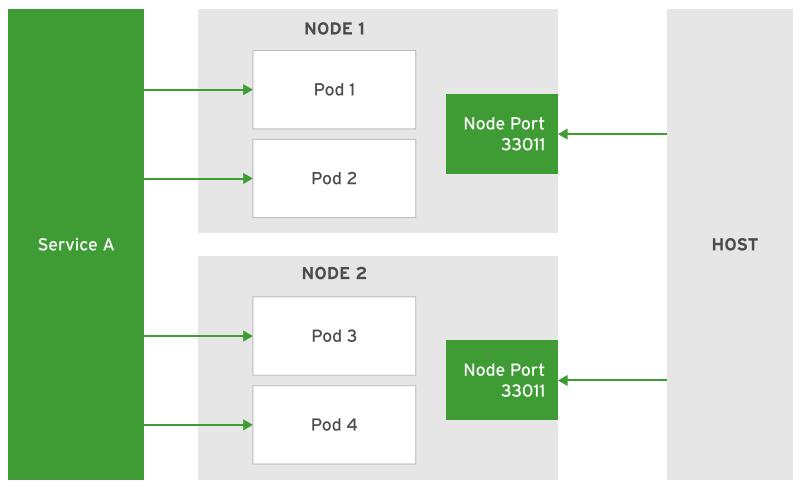


Figure 6.5: Alternative method for external access to a Kubernetes service

OpenShift provides the **oc port-forward** command for forwarding a local port to a pod port. This is different from having access to a pod through a service resource:

- The port-forwarding mapping exists only on the workstation where the **oc** client runs, while a service maps a port for all network users.
- A service load-balances connections to potentially multiple pods, whereas a port-forwarding mapping forwards connections to a single pod.



NOTE

Red Hat discourages the use of the **NodePort** approach to avoid exposing the service to direct connections. Mapping via port-forwarding in OpenShift is considered a more secure alternative.

The following example demonstrates the use of the **oc port-forward** command:

```
$ oc port-forward mysql-openshift-1-g1qrp 3306:3306
```

The previous command forwards port 3306 from the developer machine to port 3306 on the **db** pod, where a MySQL server (inside a container) accepts network connections.



NOTE

When running this command, make sure you leave the terminal window running. Closing the window or canceling the process stops the port mapping.

CREATING NEW APPLICATIONS

Simple applications, complex multtier applications, and microservice applications can be described by a single resource definition file. This single file would contain many pod definitions, service definitions to connect the pods, replication controllers or DeploymentConfigs to horizontally scale the application pods, PersistentVolumeClaims to persist application data, and anything else needed that can be managed by OpenShift.

The **oc new-app** command can be used with the **-o json** or **-o yaml** option to create a skeleton resource definition file in JSON or YAML format, respectively. This file can be customized and used to create an application using the **oc create -f <filename>** command, or merged with other resource definition files to create a composite application.

The **oc new-app** command can create application pods to run on OpenShift in many different ways. It can create pods from existing docker images, from Dockerfiles, and from raw source code using the Source-to-Image (S2I) process.

Run the **oc new-app -h** command to understand all the different options available for creating new applications on OpenShift.

The following command creates an application based on an image, **mysql**, from Docker Hub, with the label set to **db=mysql**:

```
oc new-app mysql MYSQL_USER=user MYSQL_PASSWORD=pass MYSQL_DATABASE=testdb -l db=mysql
```

The following figure shows the Kubernetes and OpenShift resources created by the **oc new-app** command when the argument is a container image:

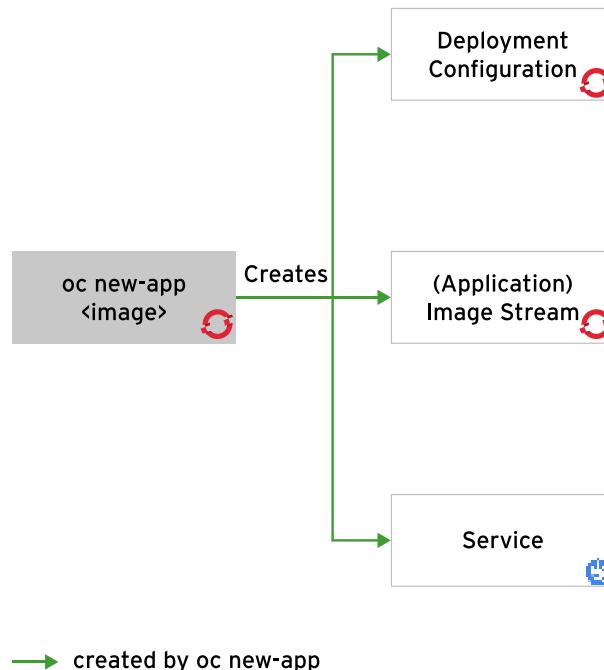


Figure 6.6: Resources created for a new application

The following command creates an application based on an image from a private Docker image registry:

```
oc new-app --docker-image=myregistry.com/mycompany/myapp --name=myapp
```

The following command creates an application based on source code stored in a Git repository:

```
oc new-app https://github.com/openshift/ruby-hello-world --name=ruby-hello
```

You will learn more about the Source-to-Image (S2I) process, its associated concepts, and more advanced ways to use `oc new-app` to build applications for OpenShift in the next section.

MANAGING OPENSHIFT RESOURCES AT THE COMMAND LINE

There are several essential commands used to manage OpenShift resources as described below.

Use the `oc get` command to retrieve information about resources in the cluster. Generally, this command outputs only the most important characteristics of the resources and omits more detailed information.

The `oc get RESOURCE_TYPE` command displays a summary of all resources of the specified type. The following illustrates example output of the `oc get pods` command.

NAME	READY	STATUS	RESTARTS	AGE
nginx-1-5r583	1/1	Running	0	1h
myapp-1-144m7	1/1	Running	0	1h

oc get all

Use the `oc get all` command to retrieve a summary of the most important components of a cluster. This command iterates through the major resource types for the current project and prints out a summary of their information.

NAME	DOCKER REPO	TAGS	UPDATED	
is/nginx	172.30.1.1:5000/basic-kubernetes/nginx	latest	About an hour ago	
<hr/>				
NAME	REVISION	DESIRED	CURRENT	
dc/nginx	1	1	1	
TRIGGERED BY config,image(nginx:latest)				
NAME	DESIRED	CURRENT	READY	AGE
rc/nginx-1	1	1	1	1h
NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
svc/nginx	172.30.72.75	<none>	80/TCP, 443/TCP	1h
NAME	READY	STATUS	RESTARTS	AGE
po/nginx-1-ypp8t	1/1	Running	0	1h

oc describe RESOURCE_TYPE RESOURCE_NAME

If the summaries provided by `oc get` are insufficient, use the `oc describe` command to retrieve additional information. Unlike the `oc get` command, there is no way to iterate through all the different resources by type. Although most major resources can be described, this functionality is not available across all resources. The following is an example output from describing a pod resource:

```
Name: mysql-openshift-1-glqrp
Namespace: mysql-openshift
Priority: 0
PriorityClassName: none
```

```
Node:           cluster-worker-1/172.25.250.52
Start Time:     Fri, 15 Feb 2019 02:14:34 +0000
Labels:         app=mysql-openshift
                deployment=mysql-openshift-1
                deploymentconfig=mysql-openshift
Annotations:   openshift.io/deployment-config.latest-version: 1
                openshift.io/deployment-config.name: mysql-openshift
                openshift.io/deployment.name: mysql-openshift-1
                openshift.io/generated-by: OpenShiftNewApp
                openshift.io/scc: restricted
Status:         Running
IP:             10.129.0.85
```

oc export

This command can be used to export a resource definition. Typical use cases include creating a backup, or to aid in the modification of a definition. By default, the **export** command prints out the object representation in YAML format, but this can be changed by providing a **-o** option.

oc create

This command creates resources from a resource definition. Typically, this is paired with the **oc export** command for editing definitions.

oc edit

This command allows the user to edit resources of a resource definition. By default, this command opens a **vi** buffer for editing the resource definition.

oc delete RESOURCE_TYPE name

The **oc delete** command removes a resource from an OpenShift cluster. Note that a fundamental understanding of the OpenShift architecture is needed here, because deleting managed resources such as pods results in new instances of those resources being automatically created. When a project is deleted, it deletes all of the resources and applications contained within it.

oc exec CONTAINER_ID options command

The **oc exec** command executes commands inside a container. You can use this command to run interactive and noninteractive batch commands as part of a script.

LABELLING RESOURCES

When working with many resources in the same project, it is often useful to group those resources by application, environment, or some other criteria. To establish these groups, you define labels for the resources in your project. Labels are part of the **metadata** section of a resource, and are defined as key/value pairs, as shown in the following example:

```
apiVersion: v1
kind: Service
metadata:
  ...contents omitted...
  labels:
    app: nexus
```

```
template: nexus-persistent-template
name: nexus
...contents omitted...
```

Many **oc** subcommands support a **-l** option to process resources from a label specification. For the **oc get** command, the **-l** option acts as a selector to only retrieve objects that have a matching label:

```
$ oc get svc,dc -l app=nexus
NAME          TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
service/nexus  ClusterIP  172.30.29.218  <none>           8081/TCP    4h

NAME                           REVISION      DESIRED      CURRENT      ...
deploymentconfig.apps.openshift.io/nexus  1            1            1            ...
```



NOTE

Although any label can appear in resources, both the **app** and **template** keys are common for labels. By convention, the **app** key indicates the application related to this resource. The **template** key labels all resources generated by the same template with the template's name.

When using templates to generate resources, labels are especially useful. A template resource has a **labels** section separated from the **metadata.labels** section. Labels defined in the **labels** section do not apply to the template itself, but are added to every resource generated by the template.

```
apiVersion: template.openshift.io/v1
kind: Template
labels:
  app: nexus
  template: nexus-persistent-template
metadata:
...contents omitted...
labels:
  maintainer: redhat
  name: nexus-persistent
...contents omitted...
objects:
- apiVersion: v1
  kind: Service
  metadata:
    name: nexus
  labels:
    version: 1
...contents omitted...
```

The previous example defines a template resource with a single label: **maintainer: redhat**. The template generates a service resource with three labels: **app: nexus**, **template: nexus-persistent-template**, and **version: 1**.



REFERENCES

Additional information about pods and services is available in the *Pods and Services* section of the OpenShift Container Platform documentation:

Architecture

https://access.redhat.com/documentation/en-us/openshift_container_platform/3.11/html-single/architecture/

Additional information about creating images is available in the OpenShift Container Platform documentation:

Creating Images

https://access.redhat.com/documentation/en-us/openshift_container_platform/3.11/html-single/creating_images/

Labels and label selectors details are available in *Working with Kubernetes Objects* section for the Kubernetes documentation:

Labels and Selectors

<https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/>

► GUIDED EXERCISE

DEPLOYING A DATABASE SERVER ON OPENSHIFT

In this exercise, you will create and deploy a MySQL database pod on OpenShift using the **oc new-app** command.

OUTCOMES

You should be able to create and deploy a MySQL database pod on OpenShift.

BEFORE YOU BEGIN

On workstation, run the following command to set up the environment:

```
[student@workstation ~]$ lab openshift-resources start
```

- 1. Log in to OpenShift as the `kubeadmin` user and create a new project for this exercise.

- 1.1. From the `workstation` VM, log in to OpenShift as the `kubeadmin` user with the password located in `/home/student/.kubeadmin`.

```
[student@workstation ~]$ oc login -u kubeadmin -p <password> \
> https://cluster-api.lab.example.com:6443
```

If the **oc login** command warns about using insecure connections, answer **y** (yes).

- 1.2. Create a new project for the resources you create during this exercise:

```
[student@workstation ~]$ oc new-project mysql-openshift
Now using project "mysql-openshift" on server "https://cluster-
api.lab.example.com:6443".
```

You can add applications to this project with the 'new-app' command. For example, try:

```
oc new-app centos/ruby-25-centos7~https://github.com/sclorg/ruby-ex.git
```

to build a new example application in Ruby.

- 2. Create a new application from the **rhscl/mysql-57-rhel7** container image using the **oc new-app** command.

This image requires that you use the **-e** option to set the `MYSQL_USER`, `MYSQL_PASSWORD`, `MYSQL_DATABASE`, and `MYSQL_ROOT_PASSWORD` environment variables.

Use the **--docker-image** option with the **oc new-app** command to specify the classroom private registry URI so that OpenShift does not try and pull the image from the internet:

```
[student@workstation ~]$ oc new-app \
> --docker-image=registry.lab.example.com/rhscl/mysql-57-rhel7:latest \
> --name=mysql-openshift \
> -e MYSQL_USER=user1 -e MYSQL_PASSWORD=mypa55 -e MYSQL_DATABASE=testdb \
> -e MYSQL_ROOT_PASSWORD=r00tpa55 --insecure-registry=true
--> Found Docker image e079d70 (17 months old) from registry.lab.example.com for
"registry.lab.example.com/rhscl/mysql-57-rhel7:latest"
...output omitted...
--> Creating resources ...
  imagestream.image.openshift.io "mysql-openshift" created
  deploymentconfig.apps.openshift.io "mysql-openshift" created
  service "mysql-openshift" created
--> Success
  Application is not exposed. You can expose services to the outside world by
executing one or more of the commands below:
  'oc expose svc/mysql-openshift'
  Run 'oc status' to view your app.
```

- ▶ 3. Verify that the MySQL pod was created successfully and view the details about the pod and its service.
 - 3.1. Run the **oc status** command to view the status of the new application and verify that the deployment of the MySQL image was successful:

```
[student@workstation ~]$ oc status
In project mysql-openshift on server https://cluster-api.lab.example.com:6443

svc/mysql-openshift - 172.30.114.39:3306
dc/mysql-openshift deploys istag/mysql-openshift:latest
  deployment #1 running for 11 seconds - 0/1 pods
...output omitted...
```

- 3.2. List the pods in this project to verify that the MySQL pod is ready and running:

NAME	READY	STATUS	RESTARTS	AGE	IP
NODE					

```
mysql-openshift-1-glqrp 1/1      Running  0          1m       10.129.0.85
cluster-worker-1
```

**NOTE**

Notice the worker on which the pod is running. You need this information to be able to log in to the MySQL database server later.

- 3.3. Use the **oc describe** command to view more details about the pod:

```
[student@workstation ~]$ oc describe pod mysql-openshift-1-glqrp
Name:           mysql-openshift-1-glqrp
Namespace:      mysql-openshift
Priority:       0
PriorityClassName: none
Node:           cluster-worker-1/172.25.250.52
Start Time:     Fri, 15 Feb 2019 02:14:34 +0000
Labels:         app=mysql-openshift
                deployment=mysql-openshift-1
                deploymentconfig=mysql-openshift
Annotations:   openshift.io/deployment-config.latest-version: 1
                openshift.io/deployment-config.name: mysql-openshift
                openshift.io/deployment.name: mysql-openshift-1
                openshift.io/generated-by: OpenShiftNewApp
                openshift.io/scc: restricted
Status:         Running
IP:             10.129.0.85
...output omitted...
```

- 3.4. List the services in this project and verify that the service to access the MySQL pod was created:

```
[student@workstation ~]$ oc get svc
NAME        TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
mysql-openshift  ClusterIP  172.30.114.39  <none>        3306/TCP   6m
```

- 3.5. Retrieve the details of the **mysql-openshift** service using the **oc describe** command and note that the Service type is **ClusterIP** by default:

```
[student@workstation ~]$ oc describe service mysql-openshift
Name:           mysql-openshift
Namespace:      mysql-openshift
Labels:         app=mysql-openshift
Annotations:   openshift.io/generated-by: OpenShiftNewApp
Selector:       app=mysql-openshift,deploymentconfig=mysql-openshift
Type:          ClusterIP
IP:            172.30.114.39
Port:          3306-tcp  3306/TCP
TargetPort:    3306/TCP
Endpoints:    10.129.0.85:3306
Session Affinity: None
```

```
Events: <none>
```

- 3.6. View details about the deployment configuration (**dc**) for this application:

```
[student@workstation ~]$ oc describe dc mysql-openshift
Name: mysql-openshift
Namespace: mysql-openshift
Created: 15 minutes ago
Labels: app=mysql-openshift
...output omitted...
Deployment #1 (latest):
  Name: mysql-openshift-1
  Created: 15 minutes ago
  Status: Complete
  Replicas: 1 current / 1 desired
  Selector: app=mysql-openshift,deployment=mysql-
  openshift-1,deploymentconfig=mysql-openshift
  Labels: app=mysql-openshift,openshift.io/deployment-config.name=mysql-openshift
  Pods Status: 1 Running / 0 Waiting / 0 Succeeded / 0 Failed
  ...output omitted...
```

- 3.7. Expose the service creating a route with a default name and a fully qualified domain name (FQDN):

```
[student@workstation ~]$ oc expose service mysql-openshift
route.route.openshift.io/mysql-openshift exposed
[student@workstation ~]$ oc get routes
NAME          HOST/PORT
PATH  SERVICES      PORT
mysql-openshift  mysql-openshift-mysql-openshift.apps.cluster.lab.example.com
mysql-openshift  3306-tcp
```

- ▶ 4. Connect to the MySQL database server and verify that the database was created successfully.
- 4.1. From the **workstation** machine, configure port forwarding between **workstation** and the database pod running on OpenShift using port 3306. The terminal will hang after executing the command.

```
[student@workstation ~]$ oc port-forward mysql-openshift-1-g1qrp 3306:3306
Forwarding from 127.0.0.1:3306 -> 3306
Forwarding from [::1]:3306 -> 3306
```

- 4.2. From the **workstation** machine open another terminal and connect to the MySQL server using the MySQL client.

```
[student@workstation ~]$ mysql -uuser1 -pmypa55 --protocol tcp -h localhost
Welcome to the MariaDB monitor. Commands end with ; or \g.
Your MySQL connection id is 1
Server version: 5.6.34 MySQL Community Server (GPL)
```

Copyright (c) 2000, 2016, Oracle, MariaDB Corporation Ab and others.

```
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.  
MySQL [(none)]>
```

4.3. Verify the creation of the testdb database.

```
MySQL [(none)]> show databases;  
+-----+  
| Database      |  
+-----+  
| information_schema |  
| testdb          |  
+-----+  
2 rows in set (0.00 sec)
```

4.4. Exit from the MySQL prompt:

```
MySQL [(none)]> exit  
Bye
```

► 5. Delete the project to remove all the resources within the project:

```
[student@workstation ~]$ oc delete project mysql-openshift
```

Finish

On workstation, run the **lab openshift-resources finish** script to complete this lab.

```
[student@workstation ~]$ lab openshift-resources finish
```

This concludes the exercise.

CREATING ROUTES

OBJECTIVES

After completing this section, students should be able to expose services using OpenShift routes.

WORKING WITH ROUTES

Services allow for network access between pods inside an OpenShift instance, and routes allow for network access to pods from users and applications outside the OpenShift instance.

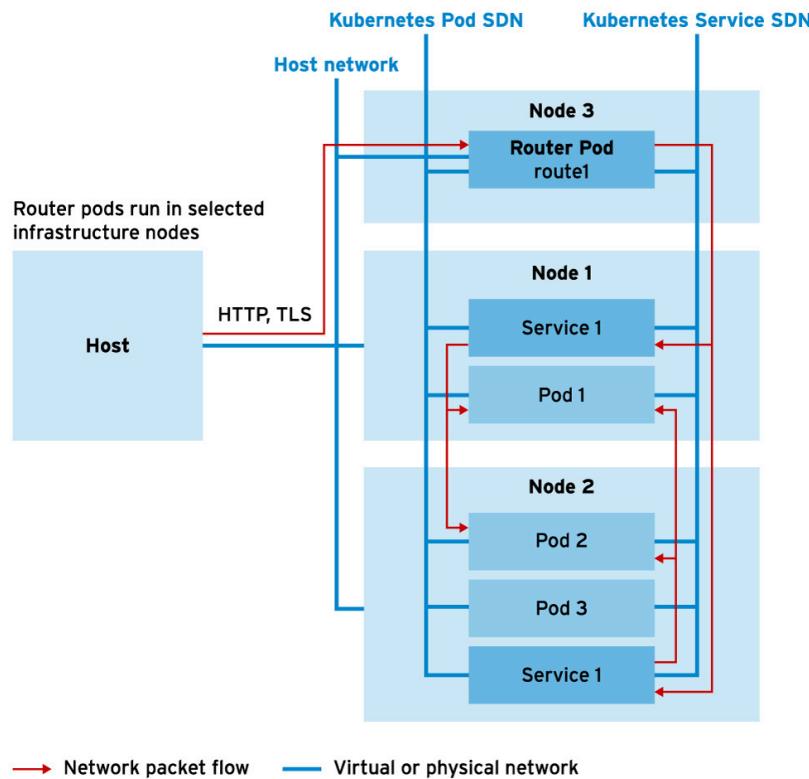


Figure 6.7: OpenShift routes and Kubernetes services

A route connects a public-facing IP address and DNS host name to an internal-facing service IP. It uses the service resource to find the endpoints; that is, the ports exposed by the service.

OpenShift routes are implemented by a cluster-wide router service, which runs as a containerized application in the OpenShift cluster. OpenShift scales and replicates router pods like any other OpenShift application.

NOTE

In practice, to improve performance and reduce latency, the OpenShift router connects directly to the pods using the internal pod software-defined network (SDN).

The router service uses *HAProxy* as the default implementation.

An important consideration for OpenShift administrators is that the public DNS host names configured for routes need to point to the public-facing IP addresses of the nodes running the router. Router pods, unlike regular application pods, bind to their nodes' public IP addresses instead of to the internal pod SDN.

The following example shows a minimal route defined using JSON syntax:

```
{  
    "apiVersion": "v1",  
    "kind": "Route",  
    "metadata": {  
        "name": "quoteapp"  
    },  
    "spec": {  
        "host": "quoteapp.apps.example.com",  
        "to": {  
            "kind": "Service",  
            "name": "quoteapp"  
        }  
    }  
}
```

The `apiVersion`, `kind`, and `metadata` attributes follow standard Kubernetes resource definition rules. The **Route** value for `kind` shows that this is a route resource, and the `metadata.name` attribute gives this particular route the identifier **quoteapp**.

As with pods and services, the main part is the `spec` attribute, which is an object containing the following attributes:

- `host` is a string containing the FQDN associated with the route. DNS must resolve this FQDN to the IP address of the OpenShift router. The details to modify DNS configuration are outside the scope of this course.
- `to` is an object stating the resource this route points to. In this case, the route points to an OpenShift Service with the name set to **quoteapp**.



NOTE

Names of different resource types do not collide. It is perfectly legal to have a route named **quoteapp** that points to a service also named **quoteapp**.



IMPORTANT

Unlike services, which use selectors to link to pod resources containing specific labels, a route links directly to the service resource name.

CREATING ROUTES

Use the **oc create** command to create route resources, just like any other OpenShift resource. You must provide a JSON or YAML resource definition file, which defines the route, to the **oc create** command.

The **oc new-app** command does not create a route resource when building a pod from container images, Dockerfiles, or application source code. After all, **oc new-app** does not know if the pod is intended to be accessible from outside the OpenShift instance or not.

Another way to create a route is to use the **oc expose service** command, passing a service resource name as the input. The **--name** option can be used to control the name of the route resource. For example:

```
$ oc expose service quotedb --name quote
```

By default, routes created by **oc expose** generate DNS names of the form:

route-name-project-name.default-domain

Where:

- *route-name* is the name assigned to the route. If no explicit name is set, OpenShift assigns the route the same name as the originating resource (for example, the service name).
- *project-name* is the name of the project containing the resource.
- *default-domain* is configured on the OpenShift master and corresponds to the wildcard DNS domain listed as a prerequisite for installing OpenShift.

For example, creating a route named **quote** in project named **test** from an OpenShift instance where the wildcard domain is **cloudapps.example.com** results in the FQDN **quote-test.cloudapps.example.com**.



NOTE

The DNS server that hosts the wildcard domain knows nothing about route host names. It merely resolves any name to the configured IP addresses. Only the OpenShift router knows about route host names, treating each one as an HTTP virtual host. The OpenShift router blocks invalid wildcard domain host names that do not correspond to any route and returns an HTTP 404 error.

Leveraging the Default Routing Service

The default routing service is implemented as an **HAProxy** pod. Router pods, containers, and their configuration can be inspected just like any other resource in an OpenShift cluster:

```
$ oc get pod --all-namespaces -l app=router
NAMESPACE      NAME           READY   STATUS    RESTARTS   AGE
openshift-ingress  router-default-746b5cfb65-f6sdm  1/1     Running   1          4d
```

By default, router is deployed in **openshift-ingress** project. Use **oc describe pod** command to get the routing configuration details:

```
$oc describe pod router-default-746b5cfb65-f6sdm
Name:            router-default-746b5cfb65-f6sdm
Namespace:       openshift-ingress
...output omitted...
Containers:
  router:
```

```
...output omitted...
Environment:
  STATS_PORT:          1936
  ROUTER_SERVICE_NAMESPACE:  openshift-ingress
  DEFAULT_CERTIFICATE_DIR: /etc/pki/tls/private
  ROUTER_SERVICE_NAME:    default
  ROUTER_CANONICAL_HOSTNAME: apps.cluster.lab.example.com
...output omitted...
```

The subdomain, or default domain to be used in all default routes, takes its value from the **ROUTER_CANONICAL_HOSTNAME** entry. It is also defined with the keyword **subdomain** in the **routingConfig** section of the OpenShift configuration file **master-config.yaml**. For example:

```
routingConfig:
  subdomain: 172.25.250.254.xip.io
```



REFERENCES

Additional information about the architecture of routes in OpenShift is available in

the *Architecture* and *Developer Guide* sections of the

OpenShift Container Platform documentation.

https://access.redhat.com/documentation/en-us/openshift_container_platform/

► GUIDED EXERCISE

EXPOSING A SERVICE AS A ROUTE

In this exercise, you will create, build, and deploy an application on an OpenShift cluster and expose its service as a route.

OUTCOMES

You should be able to expose a service as a route for a deployed OpenShift application.

BEFORE YOU BEGIN

Open a terminal on `workstation` as the `student` user and run the following command:

```
[student@workstation ~]$ lab openshift-routes start
```

- 1. Open a terminal on `workstation` and log in to OpenShift.

```
[student@workstation ~]$ oc login -u kubeadmin -p $(cat ~/.kubeadmin) \
> --insecure-skip-tls-verify=true https://cluster-api.lab.example.com:6443
Login successful.
```

You have access to the following projects and can switch between them with 'oc project <projectname>':

```
default
kube-public
kube-system
openshift
...output omitted...
Using project "default".
```

This command logs you in to the OpenShift cluster at `https://cluster-api.lab.example.com:6443` as the `kubeadmin` user. The password is populated from the content of the `~/.kubeadmin` file for ease of use. The output of the last line may differ from shown, because the `oc` command remembers last used project and uses that.

- 2. Create a new project named **route**:

```
[student@workstation ~]$ oc new-project route
Now using project "route" on server "https://cluster-api.lab.example.com:6443".
```

You can add applications to this project with the 'new-app' command. For example, try:

```
oc new-app centos/ruby-25-centos7-https://github.com/sclorg/ruby-ex.git
```

to build a new example application in Ruby.

- ▶ 3. Create a new PHP application using Source-to-Image from the Git repository at <http://services.lab.example.com/php-helloworld>

- 3.1. Use the **oc new-app** command to create the PHP application.



IMPORTANT

The following example uses a backslash (\) to indicate that the second line is a continuation of the first line. If you wish to ignore the backslash, you can type the entire command in one line.

```
[student@workstation ~]$ oc new-app \
> php:5.6~http://services.lab.example.com/php-helloworld
--> Found image 92ed8b3 (9 months old) in image stream "openshift/php" under tag
"5.6" for "php:5.6"
...output omitted...
--> Creating resources ...
...output omitted...
--> Success
Build scheduled, use 'oc logs -f bc/php-helloworld' to track its progress.
Application is not exposed. You can expose services to the outside world by
executing one or more of the commands below:
'oc expose svc/php-helloworld'
Run 'oc status' to view your app.
```

- 3.2. Wait until the application finishes building and deploying by monitoring the progress with the **oc get pods -w** command:

```
[student@workstation ~]$ oc get pods -w
NAME          READY   STATUS    RESTARTS   AGE
php-helloworld-1-build  0/1     Init:0/2   0          2s
php-helloworld-1-build  0/1     Init:0/2   0          4s
php-helloworld-1-build  0/1     Init:1/2   0          5s
php-helloworld-1-build  0/1     PodInitializing   0          6s
php-helloworld-1-build  1/1     Running   0          7s
php-helloworld-1-deploy  0/1     Pending   0          0s
php-helloworld-1-deploy  0/1     Pending   0          0s
php-helloworld-1-deploy  0/1     ContainerCreating 0          0s
php-helloworld-1-build  0/1     Completed   0          5m8s
php-helloworld-1-cnphm  0/1     Pending   0          0s
php-helloworld-1-cnphm  0/1     Pending   0          1s
php-helloworld-1-deploy  1/1     Running   0          4s
php-helloworld-1-cnphm  0/1     ContainerCreating 0          1s
php-helloworld-1-cnphm  1/1     Running   0          62s
php-helloworld-1-deploy  0/1     Completed   0          65s
php-helloworld-1-deploy  0/1     Terminating 0          66s
php-helloworld-1-deploy  0/1     Terminating 0          66s
```

^C

Your exact output may differ in names, status, timing, and order. Look for the **Completed** container with the **deploy** suffix: That means the application is deployed successfully. The container in **Running** status with a random suffix (**cnpfhm** in the example) contains the application and shows it is up and running.

Alternatively, monitor the build and deployment logs with the **oc logs -f bc/php-helloworld** and **oc logs -f dc/php-helloworld** commands, respectively.

```
[student@workstation ~]$ oc logs -f bc/php-helloworld
Cloning "http://services.lab.example.com/php-helloworld" ...
Commit: b2ceac129dcf3b9d03eef2545b90e884bba47d9e (Changed index page contents.)
Author: Student User <student@workstation.lab.example.com>
...output omitted...
STEP 7: USER 1001
STEP 8: RUN /usr/libexec/s2i/assemble
--> Installing application source...
...output omitted...
Push successful
[student@workstation ~]$ oc logs -f dc/php-helloworld
=> sourcing 50-mpm-tuning.conf ...
=> sourcing 40-ssl-certs.sh ...
...output omitted...
[core:notice] [pid 1] AH000094: Command line: 'httpd -D FOREGROUND'
```

Your exact output may differ.

3.3. Review the service for this application using the **oc describe** command:

```
[student@workstation ~]$ oc describe svc/php-helloworld
Name:           php-helloworld
Namespace:      route
Labels:          app=php-helloworld
Annotations:    openshift.io/generated-by=OpenShiftNewApp
Selector:        app=php-helloworld,deploymentconfig=php-helloworld
Type:           ClusterIP
IP:             172.30.200.65
Port:           8080-tcp  8080/TCP
TargetPort:     8080/TCP
Endpoints:      10.129.0.31:8080
Port:           8443-tcp  8443/TCP
TargetPort:     8443/TCP
Endpoints:      10.129.0.31:8443
Session Affinity: None
Events:         <none>
```

The IP address displayed in the output of the command may differ.

- ▶ 4. Expose the service, which creates a route. Use the default name and fully qualified domain name (FQDN) for the route:

```
[student@workstation ~]$ oc expose svc/php-helloworld
```

```
route.route.openshift.io/php-helloworld exposed
[student@workstation ~]$ oc describe route
Name:          php-helloworld
Namespace:     route
Created:       4 minutes ago
Labels:        app=php-helloworld
Annotations:   openshift.io/host.generated=true
Requested Host: php-helloworld-route.apps.cluster.lab.example.com
               exposed on router default (host apps.cluster.lab.example.com) 4 minutes ago
Path:          <none>
TLS Termination: <none>
Insecure Policy: <none>
Endpoint Port:  8080-tcp

Service:      php-helloworld
Weight:       100 (100%)
Endpoints:    10.130.0.48:8443, 10.130.0.48:8080
```

- 5. Access the service from a host external to the cluster to verify that the service and route are working.

```
[student@workstation ~]$ curl php-helloworld-route.apps.cluster.lab.example.com
Hello, World! php version is 5.6.25
```



NOTE

The output of the PHP application depends on the actual code in the Git repository. It may be different if you updated the code in previous sections.

Notice the FQDN is comprised of the application name and project name by default. The remainder of the FQDN, the subdomain, is defined when OpenShift is installed.

- 6. Replace this route with a route named **xyz**.

6.1. Delete the current route:

```
[student@workstation ~]$ oc delete route/php-helloworld
route.route.openshift.io "php-helloworld" deleted
```



NOTE

Deleting the route is optional. You can have multiple routes for the same service, provided they have different names.

6.2. Create a route for the service with a name of **xyz**.

```
[student@workstation ~]$ oc expose svc/php-helloworld --name=xyz
route.route.openshift.io/xyz exposed
[student@workstation ~]$ oc describe route
Name:          xyz
Namespace:     route
```

```
Created:          About a minute ago
Labels:           app=php-helloworld
Annotations:     openshift.io/host.generated=true
Requested Host: xyz-route.apps.cluster.lab.example.com
                  exposed on router default (host apps.cluster.lab.example.com) 2 minutes ago
Path:             <none>
TLS Termination: <none>
Insecure Policy: <none>
Endpoint Port:   8080-tcp

Service:         php-helloworld
Weight:          100 (100%)
Endpoints:       10.130.0.48:8443, 10.130.0.48:8080
```

Note the new FQDN that was generated based on the new route name.

- 6.3. Make an HTTP request using the FQDN on port 80:

```
[student@workstation ~]$ curl xyz-route.apps.cluster.lab.example.com
Hello, World! php version is 5.6.25
```

Finish

On workstation, run the **lab openshift-routes finish** script to complete this exercise.

```
[student@workstation ~]$ lab openshift-routes finish
```

This concludes the guided exercise.

CREATING APPLICATIONS WITH SOURCE-TO-IMAGE

OBJECTIVES

After completing this section, students should be able to deploy an application using the Source-to-Image (S2I) facility of OpenShift Container Platform.

THE SOURCE-TO-IMAGE (S2I) PROCESS

Source-to-Image (S2I) is a tool that makes it easy to build container images from application source code. This tool takes an application's source code from a Git repository, injects the source code into a base container based on the language and framework desired, and produces a new container image that runs the assembled application.

Figure 6.8 shows the resources created by the `oc new-app` command when the argument is an application source code repository. Notice that S2I also creates a Deployment Configuration and all its dependent resources.

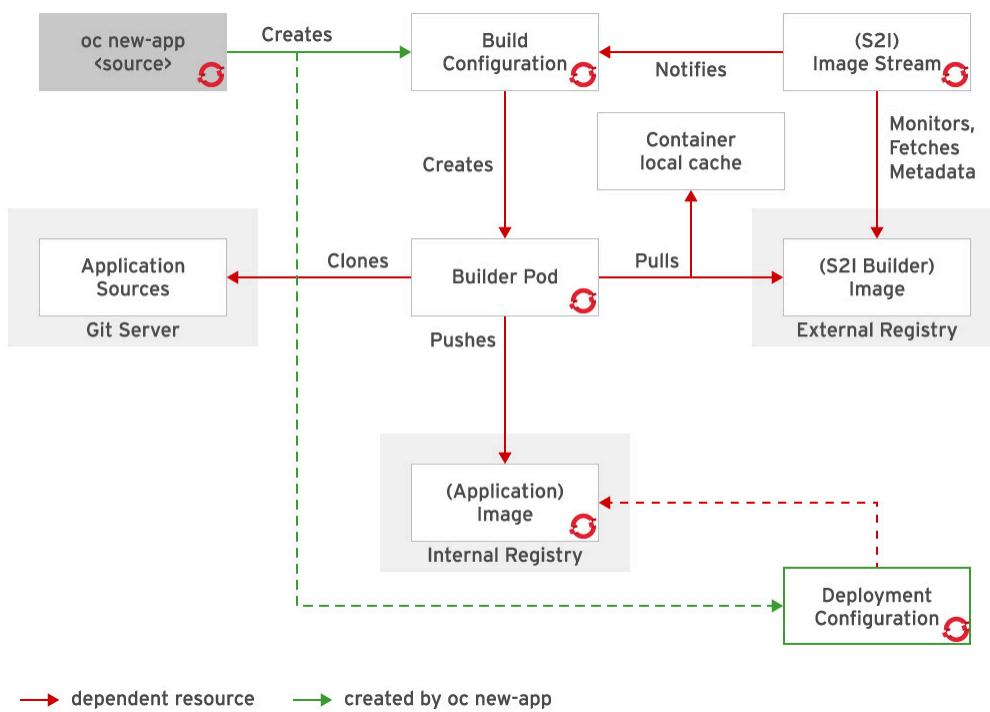


Figure 6.8: Deployment Configuration and dependent resources

S2I is the primary strategy used for building applications in OpenShift Container Platform. The main reasons for using source builds are:

- User efficiency: Developers do not need to understand Dockerfiles and operating system commands such as `yum install`. They work using their standard programming language tools.
- Patching: S2I allows for rebuilding all the applications consistently if a base image needs a patch due to a security issue. For example, if a security issue is found in a PHP base image, then updating this image with security patches updates all applications that use this image as a base.

- Speed: With S2I, the assembly process can perform a large number of complex operations without creating a new layer at each step, resulting in faster builds.
- Ecosystem: S2I encourages a shared ecosystem of images where base images and scripts can be customized and reused across multiple types of applications.

DESCRIBING IMAGE STREAMS

OpenShift deploys new versions of user applications into pods quickly. To create a new application, in addition to the application source code, a base image (the S2I builder image) is required. If either of these two components gets updated, OpenShift creates a new container image. Pods created using the older container image are replaced by pods using the new image.

Even though it is evident that the container image needs to be updated when application code changes, it may not be evident that the deployed pods also need to be updated should the builder image change.

The *image stream resource* is a configuration that names specific container images associated with *image stream tags*, an alias for these container images. OpenShift builds applications against an image stream. The OpenShift installer populates several image streams by default during installation. To determine available image streams, use the **oc get** command, as follows:

```
$ oc get is -n openshift
NAME          IMAGE REPOSITORY      TAGS
cli           ...svc:5000/openshift/cli    latest
dotnet        ...svc:5000/openshift/dotnet   2.0,2.1,latest
dotnet-runtime ...svc:5000/openshift/dotnet-runtime 2.0,2.1,latest
httpd         ...svc:5000/openshift/httpd    2.4,latest
jenkins       ...svc:5000/openshift/jenkins  1,2
mariadb       ...svc:5000/openshift/mariadb  10.1,10.2,latest
mongodb       ...svc:5000/openshift/mongodb  2.4,2.6,3.2,3.4,3.6,latest
mysql         ...svc:5000/openshift/mysql    5.5,5.6,5.7,latest
nginx         ...svc:5000/openshift/nginx    1.10,1.12,1.8,latest
nodejs        ...svc:5000/openshift/nodejs   0.10,10,11,4,6,8,latest
perl          ...svc:5000/openshift/perl     5.16,5.20,5.24,5.26,latest
php           ...svc:5000/openshift/php      5.5,5.6,7.0,7.1,latest
postgresql    ...svc:5000/openshift/postgresql 10,9.2,9.4,9.5,9.6,latest
python        ...svc:5000/openshift/python   2.7,3.3,3.4,3.5,3.6,latest
redis         ...svc:5000/openshift/redis    3.2,latest
ruby          ...svc:5000/openshift/ruby    2.0,2.2,2.3,2.4,2.5,latest
wildfly       ...svc:5000/openshift/wildfly  10.0,10.1,11.0,12.0,...
```



NOTE

Your OpenShift instance may have more or fewer image streams depending on local additions and OpenShift point releases.

OpenShift detects when an image stream changes and takes action based on that change. If a security issue arises in the **nodejs-010-rhel7** image, it can be updated in the image repository, and OpenShift can automatically trigger a new build of the application code.

It is likely that an organization chooses several supported base S2I images from Red Hat, but may also create their own base images.

BUILDING AN APPLICATION WITH S2I AND THE CLI

Building an application with S2I can be accomplished using the OpenShift CLI.

An application can be created using the S2I process with the **oc new-app** command from the CLI.

```
$ oc new-app ①php~http://services.lab.example.com/app② --name=myapp③
```

- ① The image stream used in the process appears to the left of the tilde (~).
- ② The URL after the tilde indicates the location of the source code's Git repository.
- ③ Sets the application name.



NOTE

Instead of using the tilde, you can set the image stream by using the **-i** option.

```
$ oc new-app -i php http://services.lab.example.com/app --name=myapp
```

The **oc new-app** command allows creating applications using source code from a local or remote Git repository. If only a source repository is specified, **oc new-app** tries to identify the correct image stream to use for building the application. In addition to application code, S2I can also identify and process Dockerfiles to create a new image.

The following example creates an application using the Git repository in the current directory.

```
$ oc new-app .
```



IMPORTANT

When using a local Git repository, the repository must have a remote origin that points to a URL accessible by the OpenShift instance.

It is also possible to create an application using a remote Git repository and a context subdirectory:

```
$ oc new-app https://github.com/openshift/sti-ruby.git \
--context-dir=2.0/test/puma-test-app
```

Finally, it is possible to create an application using a remote Git repository with a specific branch reference:

```
$ oc new-app https://github.com/openshift/ruby-hello-world.git#beta4
```

If an image stream is not specified in the command, **new-app** attempts to determine which language builder to use based on the presence of certain files in the root of the repository:

LANGUAGE	FILES
Ruby	Rakefile Gemfile , config.ru

LANGUAGE	FILES
Java EE	pom.xml
Node.js	app.json package.json
PHP	index.php composer.json
Python	requirements.txt config.py
Perl	index.pl cpanfile

After a language is detected, the **new-app** command searches for image stream tags that have support for the detected language, or an image stream that matches the name of the detected language.

A JSON resource definition file can be created using the **-o json** parameter and output redirection:

```
$ oc -o json new-app php~http://services.lab.example.com/app --name=myapp > s2i.json
```

This JSON definition file creates a list of resources. The first resource is the image stream:

```
...output omitted...
{
  "kind": "ImageStream", ❶
  "apiVersion": "image.openshift.io/v1",
  "metadata": {
    "name": "myapp", ❷
    "creationTimestamp": null
    "labels": {
      "app": "myapp"
    },
    "annotations": {
      "openshift.io/generated-by": "OpenShiftNewApp"
    }
  },
  "spec": {
    "lookupPolicy": {
      "local": false
    }
  },
  "status": {
    "dockerImageRepository": ""
  }
},
...output omitted...
```

- ❶ Define a resource type of image stream.
- ❷ Name the image stream **myapp**.

The build configuration (**bc**) is responsible for defining input parameters and triggers that are executed to transform the source code into a runnable image. The **BuildConfig** (BC) is the second resource, and the following example provides an overview of the parameters used by OpenShift to create a runnable image.

```
...output omitted...
{
  "kind": "BuildConfig", ①
  "apiVersion": "build.openshift.io/v1",
  "metadata": {
    "name": "myapp", ②
    "creationTimestamp": null,
    "labels": {
      "app": "myapp"
    },
    "annotations": {
      "openshift.io/generated-by": "OpenShiftNewApp"
    }
  },
  "spec": {
    "triggers": [
      {
        "type": "GitHub",
        "github": {
          "secret": "S5_4BZpPabM6KrIuPBvI"
        }
      },
      {
        "type": "Generic",
        "generic": {
          "secret": "3q8K8JNDoRzhjoz1KgMz"
        }
      },
      {
        "type": "ConfigChange"
      },
      {
        "type": "ImageChange",
        "imageChange": {}
      }
    ],
    "source": {
      "type": "Git",
      "git": {
        "uri": "http://services.lab.example.com/app" ③
      }
    },
    "strategy": {
      "type": "Source", ④
      "sourceStrategy": {
        "from": {
          "kind": "ImageStreamTag",
          "namespace": "openshift",
        }
      }
    }
  }
}
```

```

        "name": "php:7.1" ⑤
    }
}
},
"output": {
    "to": {
        "kind": "ImageStreamTag",
        "name": "myapp:latest" ⑥
    }
},
"resources": {},
"postCommit": {},
"nodeSelector": null
},
"status": {
    "lastVersion": 0
}
},
...output omitted...

```

- ① Define a resource type of **BuildConfig**.
- ② Name the **BuildConfig** myapp.
- ③ Define the address to the source code Git repository.
- ④ Define the strategy to use S2I.
- ⑤ Define the builder image as the php:7.1 image stream.
- ⑥ Name the output image stream **myapp:latest**.

The third resource is the deployment configuration that is responsible for customizing the deployment process in OpenShift. It may include parameters and triggers that are necessary to create new container instances, and are translated into a replication controller from Kubernetes. Some of the features provided by **DeploymentConfig** objects are:

- User customizable strategies to transition from the existing deployments to new deployments.
- Rollbacks to a previous deployment.
- Manual replication scaling.

```

...output omitted...
{
    "kind": "DeploymentConfig", ①
    "apiVersion": "apps.openshift.io/v1",
    "metadata": {
        "name": "myapp", ②
        "creationTimestamp": null,
        "labels": {
            "app": "myapp"
        },
        "annotations": {
            "openshift.io/generated-by": "OpenShiftNewApp"
        }
    },
}
```

```

"spec": {
    "strategy": {
        "resources": {}
    },
    "triggers": [
        {
            "type": "ConfigChange" 3
        },
        {
            "type": "ImageChange", 4
            "imageChangeParams": {
                "automatic": true,
                "containerNames": [
                    "myapp"
                ],
                "from": {
                    "kind": "ImageStreamTag",
                    "name": "myapp:latest"
                }
            }
        }
    ],
    "replicas": 1,
    "test": false,
    "selector": {
        "app": "myapp",
        "deploymentconfig": "myapp"
    },
    "template": {
        "metadata": {
            "creationTimestamp": null,
            "labels": {
                "app": "myapp",
                "deploymentconfig": "myapp"
            },
            "annotations": {
                "openshift.io/generated-by": "OpenShiftNewApp"
            }
        },
        "spec": {
            "containers": [
                {
                    "name": "myapp",
                    "image": "myapp:latest", 5
                    "ports": [ 6
                        {
                            "containerPort": 8080,
                            "protocol": "TCP"
                        },
                        {
                            "containerPort": 8443,
                            "protocol": "TCP"
                        }
                    ]
                }
            ]
        }
    }
}

```

```

        ],
        "resources": {}
    }
]
}
},
"status": {
    "latestVersion": 0,
    "observedGeneration": 0,
    "replicas": 0,
    "updatedReplicas": 0,
    "availableReplicas": 0,
    "unavailableReplicas": 0
}
},
...output omitted...

```

- ➊ Define a resource type of **DeploymentConfig**.
- ➋ Name the **DeploymentConfig** **myapp**.
- ➌ A configuration change trigger causes a new deployment to be created any time the replication controller template changes.
- ➍ An image change trigger causes the creation of a new deployment each time a new version of the **myapp:latest** image is available in the repository.
- ➎ Defines the container image to deploy: **myapp:latest**.
- ➏ Specifies the container ports.

The last item is the service, already covered in previous chapters:

```

...output omitted...
{
    "kind": "Service",
    "apiVersion": "v1",
    "metadata": {
        "name": "myapp",
        "creationTimestamp": null,
        "labels": {
            "app": "myapp"
        },
        "annotations": {
            "openshift.io/generated-by": "OpenShiftNewApp"
        }
    },
    "spec": {
        "ports": [
            {
                "name": "8080-tcp",
                "protocol": "TCP",
                "port": 8080,
                "targetPort": 8080
            },
            {
                "name": "8443-tcp",

```

```

        "protocol": "TCP",
        "port": 8443,
        "targetPort": 8443
    }
],
"selector": {
    "app": "myapp",
    "deploymentconfig": "myapp"
}
},
"status": {
    "loadBalancer": {}
}
}
}

```

**NOTE**

By default, the **oc new-app** command does not create a route. You can create a route after creating the application. However, a route is automatically created when using the web console because it uses a template.

After creating a new application, the build process starts. Use the **oc get builds** command to see a list of application builds:

```
$ oc get builds
NAME          TYPE      FROM      STATUS      STARTED      DURATION
php-helloworld-1  Source   Git@9e17db8  Running   13 seconds ago
```

OpenShift allows viewing of the build logs. The following command shows the last few lines of the build log:

```
$ oc logs build/myapp-1
```

**IMPORTANT**

If the build is not **Running** yet, or OpenShift has not deployed the **s2i-build** pod yet, the above command throws an error. Just wait a few moments and retry it.

Trigger a new build with the **oc start-build build_config_name** command:

```
$ oc get buildconfig
NAME          TYPE      FROM      LATEST
myapp        Source   Git        1
```

```
$ oc start-build myapp
build "myapp-2" started
```

RELATIONSHIP BETWEEN BUILD AND DEPLOYMENT CONFIGURATIONS

The **BuildConfig** pod is responsible for creating the images in OpenShift and pushing them to the internal container registry. Any source code or content update typically requires a new build to guarantee the image is updated.

The **DeploymentConfig** pod is responsible for deploying pods to OpenShift. The outcome of a **DeploymentConfig** pod execution is the creation of pods with the images deployed in the internal container registry. Any existing running pod may be destroyed, depending on how the **DeploymentConfig** resource is set.

The **BuildConfig** and **DeploymentConfig** resources do not interact directly. The **BuildConfig** resource creates or updates a container image. The **DeploymentConfig** reacts to this new image or updated image event and creates pods from the container image.

REFERENCES



Source-to-Image (S2I) Build

https://access.redhat.com/documentation/en-us/openshift_container_platform/3.11/html-single/architecture/#source-build

S2I Tool

https://access.redhat.com/documentation/en-us/openshift_container_platform/3.11/html-single/developer_guide/#dev-guide-s2i-tool

► GUIDED EXERCISE

CREATING A CONTAINERIZED APPLICATION WITH SOURCE-TO-IMAGE

In this exercise, you will explore a Source-to-Image container, build an application from source code, and deploy the application to an OpenShift cluster.

OUTCOMES

You should be able to:

- Describe the layout of a Source-to-Image container and the scripts used to build and run an application within the container.
- Build an application from source code using the OpenShift command-line interface.
- Verify the successful deployment of the application using the OpenShift command-line interface.

BEFORE YOU BEGIN

Run the following command to download the relevant lab files and configure the environment:

```
[student@workstation ~]$ lab openshift-s2i start
```

► 1. Examine the source code for the PHP version 5.6 Source-to-Image container.

- 1.1. Go to the lab directory.

```
[student@workstation ~]$ cd ~/DO180/labs/openshift-s2i
```

- 1.2. Use the **tree** command to review the files that make up the container image.

```
[student@workstation openshift-openshift-s2i]$ tree s2i-php-container
s2i-php-container/
├── 5.6
│   ├── cccp.yml
│   ├── contrib
│   └── etc
│       ├── conf.d
│       │   ├── 00-documentroot.conf.template
│       │   └── 50-mpm-tuning.conf.template
│       ├── httpdconf.sed
│       ├── php.d
│       │   └── 10-opcache.ini.template
│       └── php.ini.template
└── scl_enable
```

```

    └── Dockerfile
    └── Dockerfile.rhel7
    └── README.md
    └── s2i
        └── bin
            └── assemble
            └── run
            └── usage
        └── test
            └── run
            └── test-app
                └── composer.json
                └── index.php
    └── hack
        └── build.sh
    └── common.mk
    └── LICENSE
    └── Makefile
    └── README.md

```

- 1.3. Review the **s2i-php-container/5.6/s2i/bin/assemble** script. Note how it moves the PHP source code from the `/tmp/src/` directory to the container working directory near the top of the script. The OpenShift Source-to-Image process executes the `git clone` command on the Git repository provided at creation using the `oc new-app` command or the web console. The remainder of the script supports retrieving PHP packages that your application declares as requirements, if any.
- 1.4. Review the **s2i-php-container/5.6/s2i/bin/run** script. The PHP container built by the Source-to-Image process uses this script as the container's default command (The equivalent of the `CMD` instruction in a Dockerfile). This script is responsible for setting up and running the Apache HTTP service, which executes the PHP code in response to HTTP requests.
- 1.5. Review the **s2i-php-container/5.6/Dockerfile.rhel7** file. This Dockerfile builds the base PHP Source-to-Image container. It installs PHP and Apache HTTP Server from the Red Hat Software Collections Library, copies the Source-to-Image scripts you examined in earlier steps to their expected location, and modifies files and file permissions as needed to run on an OpenShift cluster.

► 2. Log in to OpenShift:

```

[student@workstation openshift-s2i]$ oc login -u kubeadmin \
> -p $(cat /home/student/.kubeadmin) \
> https://cluster-api.lab.example.com:6443
Login successful.

You have access to the following projects and can switch between them with ...

* default
...output omitted...

Using project "default".

```

- 3. Create a new project named **s2i**:

```
[student@workstation openshift-s2i]$ oc new-project s2i
Now using project "s2i" on server "https://cluster-api.lab.example.com:6443".
...output omitted...
```

- 4. Create a new PHP application using Source-to-Image from the Git repository at <http://services.lab.example.com/php-helloworld>

- 4.1. Use the **oc new-app** command to create the PHP application.



IMPORTANT

The following example uses the backslash (\) to indicate that the second line is a continuation of the first line. If you wish to ignore the backslash, you can type the entire command in one line.

```
[student@workstation openshift-s2i]$ oc new-app -i php:5.6 \
> --name=php-helloworld http://services.lab.example.com/php-helloworld
```

- 4.2. Wait for the build to complete and the application to deploy. Verify that the build process starts with the **oc get pods** command.

```
[student@workstation openshift-s2i]$ oc get pods
NAME                  READY   STATUS    RESTARTS   AGE
php-helloworld-1-build   1/1     Running   0          5s
```

- 4.3. Examine the logs for this build. Use the build pod name for this build, **php-helloworld-1-build**.

```
[student@workstation openshift-s2i]$ oc logs --all-containers \
> -f php-helloworld-1-build
Cloning "http://services.lab.example.com/php-helloworld" ...
Commit: 9e17db81a82fdce4ab59ebe8e50302134c039277 (Changed index page contents.)
Author: Student User <student@workstation.lab.example.com>
Date: Mon Feb 18 13:06:24 2019 +0000
Getting image source signatures
Copying blob sha256:469cfcc7a4b3947a4fa549c68cf4f8570be53779725f0c19f3d33d15...
...output omitted...

Writing manifest to image destination
Storing signatures
Generating dockerfile with builder image image-registry.openshift-image-...
php@sha256:f3c95020fa870fcfea7d1440d07a2b947834b87bdaf000588e84ef4a599c7546
STEP 1: FROM image-registry.openshift-image-registry.svc:5000/...
...output omitted...

Pushing image ...openshift-image-registry.svc:5000/s2i/php-helloworld:latest...
Getting image source signatures
```

```
...output omitted...
STEP 8: RUN /usr/libexec/s2i/assemble
--> Installing application source...
...output omitted...
Copying config sha256:6ce5730f48d9c746e7cbd7ea7b8ed0f15b83932444d1d2bd7711d7...
21.45 KiB / 21.45 KiB 0s
Writing manifest to image destination
Storing signatures
Successfully pushed .../php-helloworld:latest@sha256:63e757a4c0edaeda497dab7...
Push successful
```

Notice the clone of the Git repository as the first step of the build. Next, the Source-to-Image process built a new container called **s2i/php-helloworld:latest**. The last step in the build process is to push this container to the OpenShift private registry.

4.4. Review the **DeploymentConfig** for this application:

```
[student@workstation openshift-s2i]$ oc describe dc/php-helloworld
Name:  php-helloworld
Namespace:      s2i
Created:        12 minutes ago
Labels:         app=php-helloworld
Annotations:    openshift.io/generated-by=OpenShiftNewApp
Latest Version: 1
Selector:       app=php-helloworld,deploymentconfig=php-helloworld
Replicas:       1
Triggers:       Config, Image(phi-helloworld@latest, auto=true)
Strategy:       Rolling
Template:
  Labels:        app=php-helloworld
                 deploymentconfig=php-helloworld
...output omitted...
  Containers:
    php-helloworld:
      Image:  docker-registry.default.svc:5000/s2i/php-
helloworld@sha256:6d274e9d6e3d4ba11e5dc3fc25e1326c8170b1562c2e3330595ed21c7dfb983
      Ports:  8080/TCP, 8443/TCP
      Environment: <none>
      Mounts:  <none>
      Volumes: <none>

  Deployment #1 (latest):
    Name:  php-helloworld-1
    Created: 5 minutes ago
    Status:  Complete
    Replicas: 1 current / 1 desired
    Selector: app=php-helloworld,deployment=php-helloworld-1,deploymentconfig=php-
helloworld
    Labels:  app=php-helloworld,openshift.io/deployment-config.name=php-helloworld
    Pods Status: 1 Running / 0 Waiting / 0 Succeeded / 0 Failed
```

...output omitted...

4.5. Add a route to test the application:

```
[student@workstation openshift-s2i]$ oc expose service php-helloworld \
> --name helloworld
route.route.openshift.io/helloworld exposed
```

4.6. Test the application by sending it an HTTP GET request:

```
[student@workstation openshift-s2i]$ curl -s \
> helloworld-s2i.apps.cluster.lab.example.com
Hello, World! php version is 5.6.25
```

- ▶ 5. Explore starting application builds by changing the application in its Git repository and executing the proper commands to start a new Source-to-Image build.

5.1. Clone the project locally using **git**:

```
[student@workstation openshift-s2i]$ git clone \
> http://services.lab.example.com/php-helloworld
Cloning into 'php-helloworld'...
remote: Counting objects: 9, done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 9 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (9/9), done.
```

5.2. Enter the source code directory.

```
[student@workstation openshift-s2i]$ cd php-helloworld
```

5.3. Edit the **index.php** file as shown below:

```
<?php
print "Hello, World! php version is " . PHP_VERSION . "\n";
print "A change is a coming!\n";
?>
```

Save the file.

5.4. Commit the changes and push the code back to the remote Git repository:

```
[student@workstation php-helloworld]$ git add .
[student@workstation php-helloworld]$ git commit -m \
'Changed index page contents.'
[master ff807f3] Changed index page contents.
Committer: Student User <student@workstation.lab.example.com>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly:
```

```
git config --global user.name "Your Name"  
git config --global user.email you@example.com
```

After doing this, you may fix the identity used for this commit with:

```
git commit --amend --reset-author  
  
1 file changed, 1 insertion(+)  
[student@workstation php-helloworld]$ git push origin master  
...output omitted...  
Counting objects: 5, done.  
Delta compression using up to 2 threads.  
Compressing objects: 100% (2/2), done.  
Writing objects: 100% (3/3), 287 bytes | 0 bytes/s, done.  
Total 3 (delta 1), reused 0 (delta 0)  
To http://services.lab.example.com/php-helloworld  
 78e0f96..eb438ba master -> master  
[student@workstation php-helloworld]$ cd ..
```

5.5. Start a new Source-to-Image build process and wait for it to build and deploy:

```
[student@workstation openshift-s2i]$ oc start-build php-helloworld  
build.build.openshift.io/php-helloworld-2 started  
[student@workstation openshift-s2i]$ oc logs php-helloworld-2-build -f  
...output omitted...  
  
Successfully pushed .../php-helloworld:latest@sha256:74e757a4c0edaeda497dab7...  
Push successful
```

After the second build has completed use the **oc get pods** command to verify that the new version of the application is running.

```
[student@workstation openshift-s2i]$ oc get pods -w  
...output omitted...  
NAME READY STATUS RESTARTS AGE  
php-helloworld-1-build 0/1 Completed 0 33m  
php-helloworld-2-2n70q 1/1 Running 0 1m  
php-helloworld-2-build 0/1 Completed 0 1m
```

Press **Ctrl+C** to exit the **oc get pods -w** command.

5.6. Test that the application serves the new content:

```
[student@workstation openshift-s2i]$ curl -s \  
> helloworld-s2i.apps.cluster.lab.example.com  
Hello, World! php version is 5.6.25  
A change is a coming!
```

Finish

On workstation, run the **lab openshift-s2i finish** script to complete this lab.

```
[student@workstation ~]$ lab openshift-s2i finish
```

This concludes the guided exercise.

CREATING APPLICATIONS WITH THE OPENSHIFT WEB CONSOLE

OBJECTIVES

After completing this section, students should be able to:

- Create an application with the OpenShift web console.
- Manage and monitor the build cycle of an application.
- Examine resources for an application.

ACCESSING THE OPENSHIFT WEB CONSOLE

The OpenShift web console allows users to execute many of the same tasks as the OpenShift command-line client. You can create projects, add applications to projects, view application resources, and manipulate application configurations as needed. The OpenShift web console runs as one or more pods, each pod running on a master node.

The web console runs in a web browser. The default URL is of the format `https://console-openshift-console.{wildcard DNS domain for the RHOCP cluster}/` By default, OpenShift generates a self-signed certificate for the web console. You must trust this certificate in order to gain access.

The web console uses a REST API to communicate with the OpenShift cluster. By default, the REST API endpoint is accessed with a different DNS name and self-signed certificate. You must also trust this certificate for the REST API endpoint.

After you have trusted for the two OpenShift certificates, the console requires authentication to proceed.

Managing Projects

Upon successful login, the Home page displays a list of projects you can access. From this page you can create, edit, or delete a project.

NAME	STATUS	REQUESTER	LABELS
PR default	Active	No requester	No labels
PR kube-public	Active	No requester	No labels
PR kube-system	Active	No requester	No labels
PR openshift	Active	No requester	No labels
PR openshift-apiserver	Active	No requester	openshift.io/run-level=1
PR openshift-apiserver-operator	Active	No requester	openshift.io/cluster-... =t... openshift.io/run-level=0
PR openshift-cluster-api	Active	No requester	name=openshift-cluster-api openshift.io/run-level=1
PR openshift-cluster-kube-scheduler-operator	Active	No requester	openshift.io/run-level=0

Figure 6.9: OpenShift web console home page

The ellipsis icon at the end of each row provides a menu with project actions. Select the appropriate entry to edit or delete the project.

If you click a project link in this view, you are redirected to the Project Status page which shows all of the applications created within that project space.

Navigating the Web Console

A navigation menu is located on the left side of the web console. Each item in the menu expands to provide access to a set of related management functions:

Catalog

The Developer Catalog option displays a page for adding common applications to a project. Each application template uses parameters that you specify to create customized OpenShift resources. Other options allow you to manage Kubernetes Operators, or add items from the OpenShift marketplace. Operators and the OpenShift marketplace are beyond the scope of this course.

Workloads

These options enable management of several types of Kubernetes and OpenShift resources, such as pods and deployment configurations. Other advanced deployment options that are accessible from this menu, such as configuration maps, secrets, and cron jobs, are beyond the scope of the course.

Networking

This menu contains options to manage OpenShift resources that affect application access, such as services and routes, for a project. Other options for configuring an OpenShift Network Policy or Ingress are available, but these topics are outside the scope of this course.

Storage

This menu contains options to configure persistent storage for project applications. In particular, persistent volumes and persistent volume claims for a project are managed from the Storage menu.

Builds

The Build Configs option displays a list of project build configurations. Click a build configuration link in this view to access an overview page for the specified build configuration. From this page, you can view and edit the application's build configuration.

The Builds option provides a list of recent build processes for application container images in the project. Click the link for a particular build to access the build logs for that particular build process.

The Image Streams option provides a list of image streams defined in the project. Click an image stream entry in this list to access an overview page to view and manage that image stream.

Monitoring

Provides options to access and manage alerts for the OpenShift cluster. Functions in the Monitoring section are outside the scope of this course.

Administration

Provides options to manage cluster and project settings, such as resource quotas and role-based access controls. Functions in the Administration section are outside the scope of this course.

CREATING NEW APPLICATIONS

Use the Developer Catalog option in the Catalog menu to add a new application to an OpenShift project. A selection of Source-to-Image (S2I) templates are available to create a technology-specific application image from the application's source code. Select a desired template, and provide the necessary information to deploy the new application.

You are not limited to deploying an application from only its source code. You can also deploy an application using:

- A container image hosted on a remote container registry.
- A YAML file that specifies the Kubernetes and OpenShift resources to create.

To create an application with either of these two methods, use the Add menu in the upper right of the Developer Catalog Page. Use the Deploy Image option to deploy an existing container image. Use the Import YAML option to create the resources specified in a YAML file, such as the type of file generated using the **oc export** command.

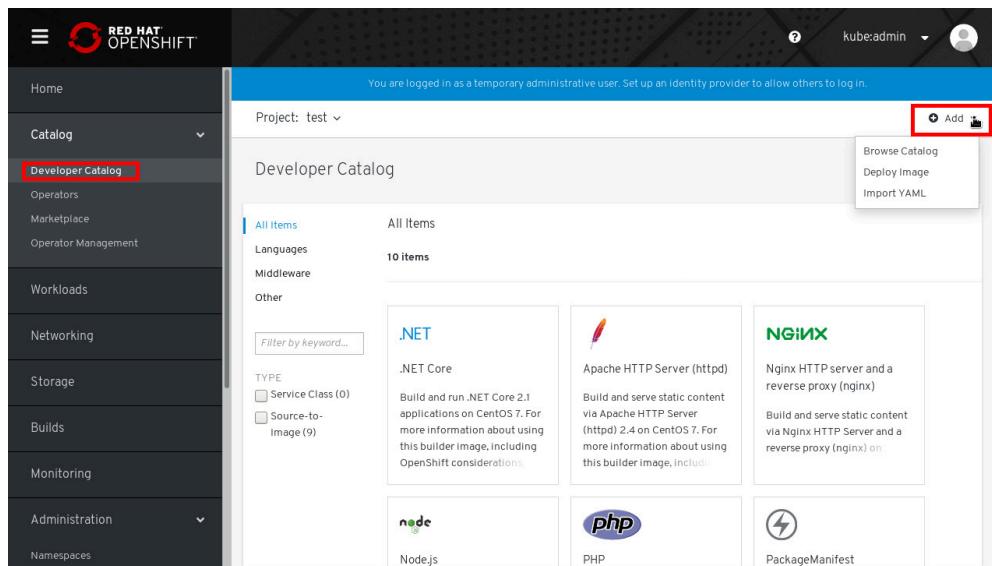


Figure 6.10: OpenShift Developer Catalog page

Managing Application Builds

Click the Build Configs option of the Builds menu after you add a Source-to-Image application to a project. The new build configuration is accessible from this view:

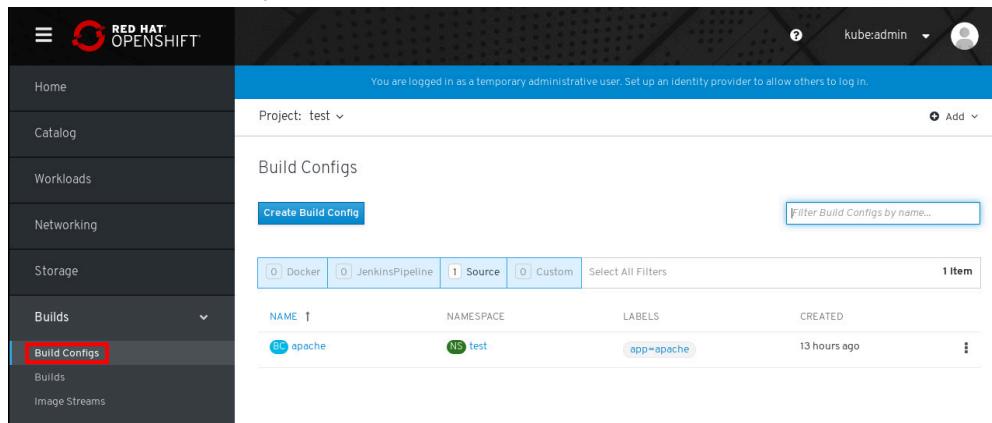


Figure 6.11: OpenShift build configurations page

Click a build configuration in the list to view an overview page for the selected build configuration. From the overview page, you can:

- View the build configuration parameters, such as the URL for the source code's Git repository.
- View and edit the environment variables that are set in the builder container, during an application build process.
- View a list of recent application builds, and click a selected build to access logs from the build process.

MANAGING DEPLOYED APPLICATIONS

The Workloads menu provides access to deployment configurations in the project.

The screenshot shows the Red Hat OpenShift web interface. On the left, there's a sidebar with a navigation menu. The 'Workloads' section is expanded, showing options like 'Pods', 'Deployments', 'Deployment Configs' (which is currently selected and highlighted in red), 'Stateful Sets', 'Secrets', and 'Config Maps'. The main content area has a header bar with the text 'You are logged in as a temporary administrative user. Set up an identity provider to allow others to log in.' and a 'kube:admin' user indicator. Below the header, it says 'Project: test'. There's a 'Create' button and a search bar labeled 'Filter Deployment Configs by name...'. A table lists a single deployment configuration named 'DC apache' in the 'test' namespace, with the label 'app=apache'. The status shows '1 of 1 pods' and the pod selector is 'app=apache, deploymentconfig=apache'.

Figure 6.12: OpenShift Workloads menu

Click a deployment configuration entry in the list to view an overview page for the selection. From the overview page, you can:

- View the deployment configuration parameters, such as the specifications of an application container image.
- Change the desired number of application pods to manually scale the application.
- View and edit the environment variables that are set in the deployed application container.
- View a list of application pods, and click a selected pod to access logs for that pod.

OTHER WEB CONSOLE FEATURES

The web console allows you to:

- Manage resources, such as project quotas, user membership, secrets, and other advanced resources.
- Create persistent volume claims.
- Monitor builds, deployments, pods, and system events.
- Create continuous integration and deployment pipelines with Jenkins.

Detailed usage for the above features is outside the scope of this course.

► GUIDED EXERCISE

CREATING AN APPLICATION WITH THE WEB CONSOLE

In this exercise, you will create, build, and deploy an application to an OpenShift cluster using the OpenShift web console.

OUTCOMES

You should be able to create, build, and deploy an application to an OpenShift cluster using the web console.

BEFORE YOU BEGIN

Get the lab files by executing the lab script:

```
[student@workstation ~]$ lab openshift-webconsole start
```

The lab script verifies that the OpenShift cluster is running.

- 1. Open a web browser and navigate to `https://console-openshift-console.apps.cluster.lab.example.com` to access the OpenShift web console. Log in and create a new project named **console**.
- 1.1. Open the Firefox browser and navigate to `https://console-openshift-console.apps.cluster.lab.example.com` to access the OpenShift web console.

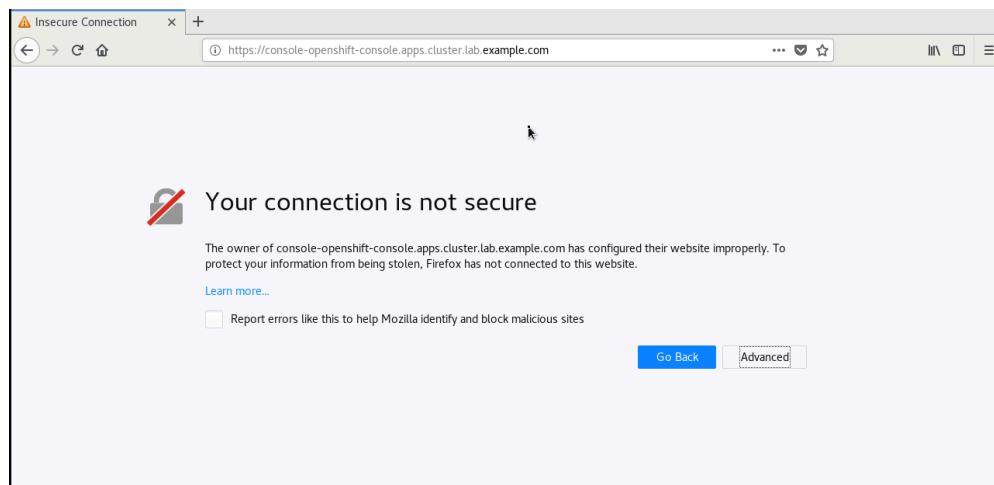


Figure 6.13: Insecure connection warning for the web console

The browser indicates that the connection to the cluster is not secure because the cluster is configured with a self-signed certificate.

- 1.2. Click **Advanced** to reveal additional detailed information about the certificate error. Scroll down and click **Add Exception**, which causes a new window to display:

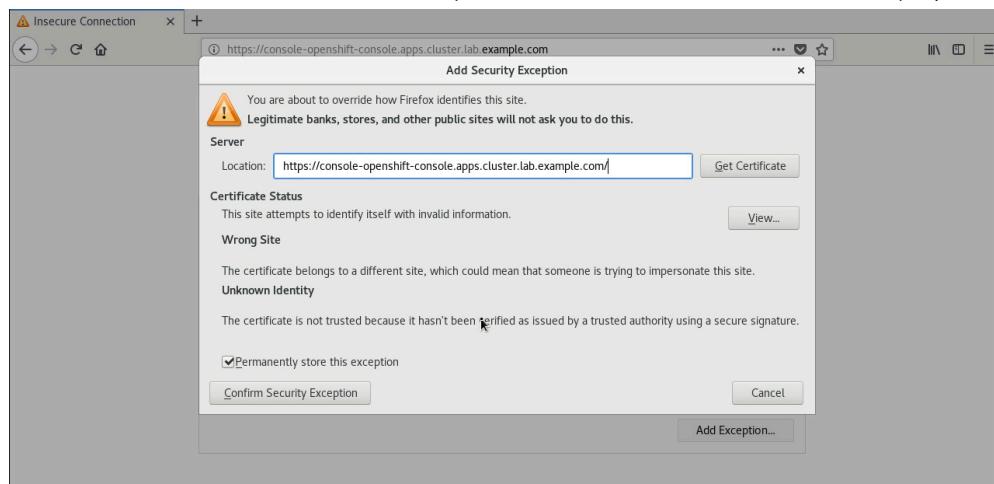


Figure 6.14: Adding a security exception

- 1.3. Click **Confirm Security Exception** to add an exception for the cluster certificate.

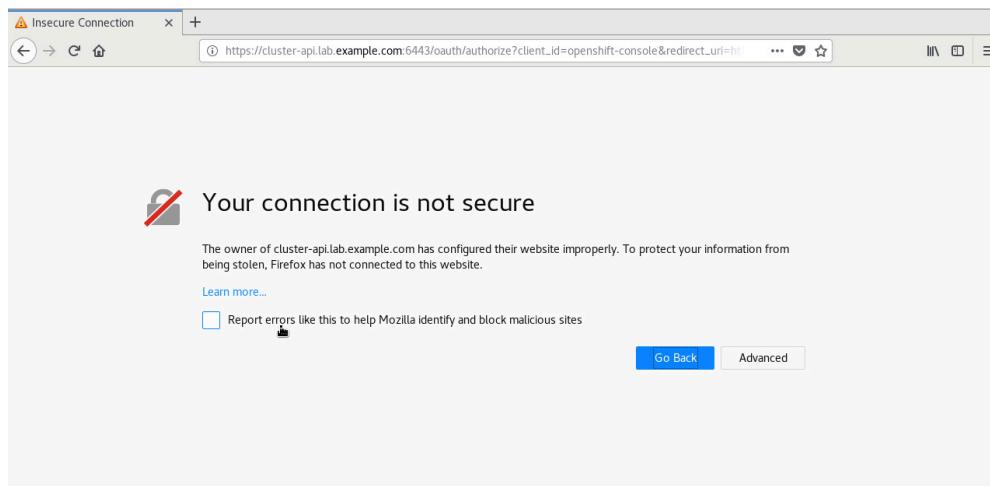


Figure 6.15: Insecure connection warning for the OpenShift REST API endpoint

Firefox displays a new secure connection warning for the `cluster-api.lab.example.com` domain.

- 1.4. The OpenShift REST API is secured with a different self-signed certificate from the web console. Repeat the process of accepting the certificate for the cluster REST API. Firefox then redirects to the OpenShift Container Platform login page:

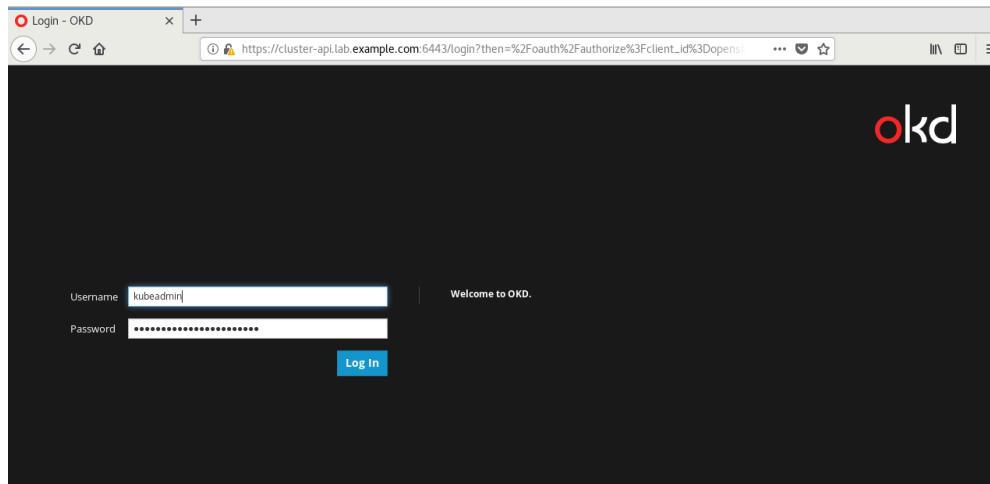


Figure 6.16: Web console login page

- 1.5. Firefox is already configured with the credentials for the `kubeadmin` user. Click `Log In`.



IMPORTANT

If authentication to the OpenShift web console fails, replace the password with the content of the file **/home/student/.kubeadmin**. Ensure the Username is **kubeadmin**, and click Log In.

- 1.6. Create a new project named **console**. You can type any values you prefer in the other fields.

NAME	STATUS	REQUESTER	LABELS
PR default	Active	No requester	No labels
PR kube-public	Active	No requester	No labels
PR kube-system	Active	No requester	No labels
PR openshift	Active	No requester	No labels
PR openshift-apiserver	Active	No requester	openshift.io/run-level=1
PR openshift-apiserver-operator	Active	No requester	openshift.io/cluster-... =t...

Figure 6.17: Create a new project - step 1

Name *

Display Name

Description

Practice creating applications with the OpenShift Web Console.

Cancel Create

Figure 6.18: Create a new project - step 2

- 1.7. After you have completed the required fields, click **Create** in the **Create Project** dialog box to go to the **Project Status** page for the **console** project:

Project: console

Project Status

Get started with your project.

Add content to your project from the catalog of web frameworks, databases, and other components. You may also deploy an existing image or create resources using YAML definitions.

Browse Catalog Deploy Image Import YAML

Figure 6.19: Project Status page

► 2. Create the new php-helloworld application with a PHP template.

2.1. Click Catalog → Developer Catalog to display a list of technology templates.

The screenshot shows the Red Hat OpenShift web interface. On the left, a sidebar menu includes Home, Catalog (which is selected), Operators, Marketplace, Operator Management, Workloads, Networking, Storage, and Builds. The main content area is titled 'Developer Catalog' and displays a grid of technology templates. One template, '.NET Core', is highlighted. Other visible templates include 'Apache HTTP Server (httpd)', 'NGINX', and 'PHP'. A search bar at the top of the catalog section is labeled 'Filter by keyword...' with the word 'php' typed into it.

Figure 6.20: Developer Catalog page

2.2. Enter **php** in the Filter by keyword field.

This screenshot shows the same developer catalog interface after filtering by the keyword 'php'. The search bar now contains 'php'. Only one template, 'PHP', is listed in the results. The 'PHP' template card shows its icon, name, type (Source-to-Image), and a brief description: 'Build and run PHP 7.1 applications on CentOS 7. For more information about using this builder image, including OpenShift considerations, see https://github.com/scrlorg/s2i-php-container/blob/master/7.1/README.md.' Below the catalog, a message says '1 items'.

Figure 6.21: Finding PHP-related templates

2.3. After filtering, click the PHP template to display the PHP dialog box. Click Create Application to display the Create Source-to-Image Application page.

This screenshot shows the 'Create Source-to-Image Application' dialog box. The 'Namespace' dropdown is set to 'PR default'. The 'Version' dropdown is set to 'ISV php:7.1'. The 'Name' input field is empty. The 'Git Repository' input field contains 'Try Sample' and has a note: 'For private Git repositories, create a source secret.' To the right of the form, there's a detailed description of what will be created: a build config, an image stream, a deployment config, a service, and a route. A sample repository URL is also provided: 'https://github.com/scrlorg/cakephp-ex.git'.

Figure 6.22: Configuring Source-to-Image for a PHP application

2.4. Ensure the Namespace is set to a value of **console**. Change the Version to PHP version 7.0.

Enter **php-helloworld** as the name for the application and the location of the source code git repository: <http://services.lab.example.com/php-helloworld>

Scroll to the bottom of the page, and select **Create route**. Click **Create** to create the required OpenShift and Kubernetes resources for the application.

You are redirected to the **Project Status** page:

A screenshot of the Red Hat OpenShift web console. The left sidebar shows navigation options like Home, Catalog, and Operators. The main content area is titled 'Project Status' and shows a single application named 'php-helloworld'. To the left of the application name is a 'DC' icon, indicating it's a Deployment Config. Below the application name, it says '0 of 1 pods'. At the top of the page, there's a message: 'You are logged in as a temporary administrative user. Set up an identity provider to allow others to log in.' There are also 'Add' and 'Filter Resources by name...' buttons.

Figure 6.23: Project Status page

This page indicates that the **php-helloworld** application is created. The **DC** annotation to the left of the **php-helloworld** link is an acronym for **Deployment Config**. This link redirects to a page containing information about the application's deployment configuration.

- ▶ 3. Use the navigation bar on the left side of the OpenShift web console to locate information for the application's OpenShift and Kubernetes resources:
 - DeploymentConfig
 - BuildConfig
 - Build Logs
 - Service
 - Route
- 3.1. Examine the deployment configuration. In the navigation bar, click **Workloads** to reveal more menu choices. Click **Deployment Configs** to display a list of deployment

configurations for the **console** project. Click the **php-helloworld** link to display deployment configuration information.

The screenshot shows the Red Hat OpenShift web interface. The left sidebar is titled 'Workloads' and includes options like 'Pods', 'Deployments', 'Deployment Configs', and 'Deployment Sets'. The main content area is titled 'Deployment Config Overview' for 'php-helloworld' in the 'console' project. It displays the following information:

- DESIRED COUNT:** 1 pod (with a pencil icon)
- UP-TO-DATE COUNT:** 1 pod
- MATCHING PODS:** 1 pod (1 available, 0 unavailable)
- NAME:** php-helloworld
- LATEST VERSION:** 1

Figure 6.24: Application deployment configuration overview page

Explore the available information from the Overview tab. The build may still be running when you reach this page, so the UP-TO-DATE COUNT and MATCHING PODS fields might not have a value of **1 pod**.

If you click the pencil icon within the DESIRED COUNT field, a dialog box displays that allows you to edit the desired number of deployed pods for the application.

- 3.2. Examine the build configuration. In the navigation bar, click Builds to reveal more menu choices. Click Build Configs to display a list of build configurations for the **console** project. Click the **php-helloworld** link to display the build configuration for the application.

The screenshot shows the Red Hat OpenShift web interface. The left sidebar is titled 'Builds' and includes options like 'Build Configs', 'Builds', and 'Image Streams'. The main content area is titled 'Build Config Overview' for 'php-helloworld' in the 'console' project. It displays the following information:

NAME	TYPE
php-helloworld	Source
NAMESPACE	GIT REPOSITORY
NS console	http://services.lab.example.com/php-helloworld
LABELS	BUILDER IMAGE
app=php-helloworld	ist php:7.0

Figure 6.25: Application build configuration overview page

Explore the available information from the Overview tab. The YAML tab allows you to view and edit the build configuration as a YAML file. The Builds tab provides an historical list of builds, along with a link to more information for each build. The Environment tab allows you to view and edit environment variables for the

application's build environment. The Events tab displays a list of build related events and metadata.

- 3.3. Examine the logs for the Source-to-Image build of the application. In the Builds menu, click Builds to display a list of recent builds for the **console** project.

Click the **php-helloworld-1** link to access information for the first build of the **php-helloworld** application:

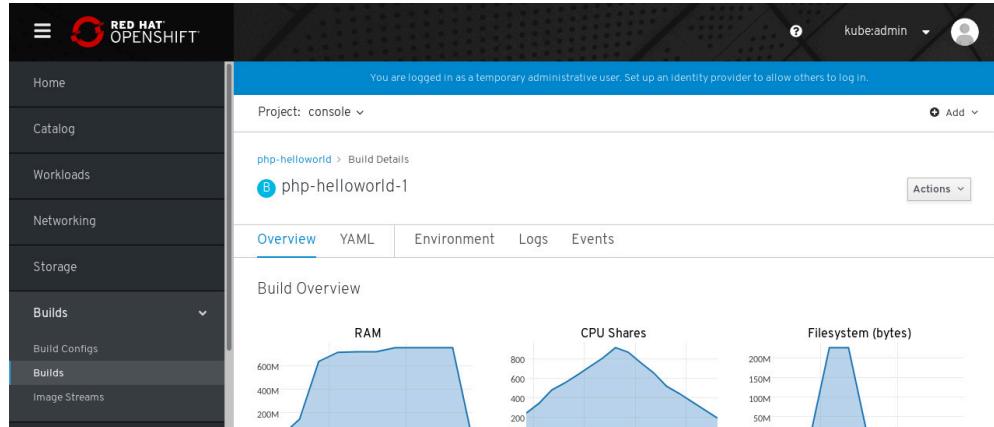


Figure 6.26: An application build overview page

Explore the available information from the Overview tab. Next, click the Logs tab. A scrollable text box contains output from the build process:

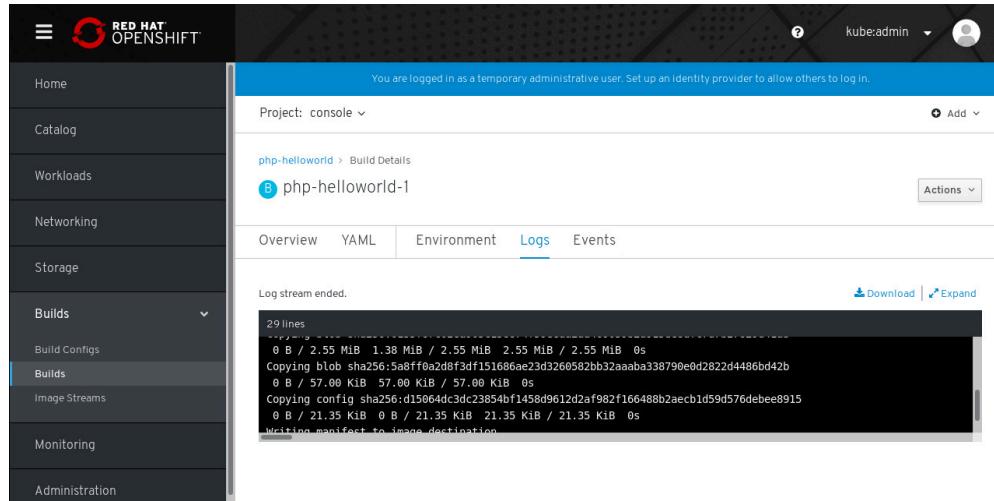


Figure 6.27: Logs for an application build

When Podman builds a container image, similar output is observed compared with the output shown in the browser.

- 3.4. Locate information for the **php-helloworld** application's service. In the navigation bar, click Networking to reveal more menu choices. Click Services to display a list

of services for the **console** project. Click the **php-helloworld** link to display the information associated with the application's service:

The screenshot shows the Red Hat OpenShift web interface. The left sidebar is collapsed. The top navigation bar shows the user is logged in as 'kube:admin'. The main content area is titled 'Project: console' and shows a service named 'php-helloworld'. The 'Overview' tab is selected. On the left, under 'Service Overview', details are provided: NAME: php-helloworld, NAMESPACE: console, LABELS: app=php-helloworld, POD SELECTOR: app=php-helloworld, deploymentconfig=php-helloworld, and ANNOTATIONS: annotations. On the right, under 'Service Routing', it shows the service address: Type: Cluster IP, Location: 172.30.242.207, Accessible within the cluster only. Below this is a table for 'SERVICE PORT MAPPING' with one entry: Name: 8080-tcp, Port: 8080, Protocol: TCP, Pod Port or Name: 8080.

Figure 6.28: Service overview page

Explore the available information from the Overview tab. The YAML tab allows you to view and edit the service configuration, as a YAML file. The Pods tab displays the current list of pods that provide the application service.

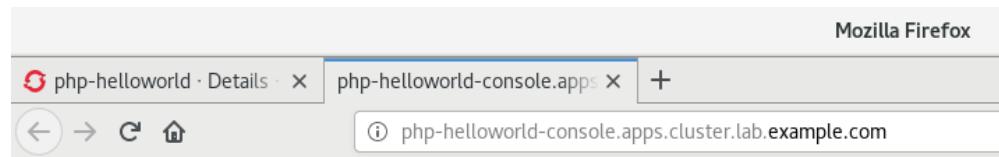
- 3.5. Locate external route information for the application. On the navigation bar, click **Networking** → **Routes** to display a list of configured routes for the **console** project. Click the **php-helloworld** link to display information associated with the application's route:

The screenshot shows the Red Hat OpenShift web interface. The left sidebar is collapsed. The top navigation bar shows the user is logged in as 'kube:admin'. The main content area is titled 'Project: console' and shows a route named 'php-helloworld'. The 'Overview' tab is selected. On the left, under 'Route Overview', details are provided: NAME: php-helloworld, NAMESPACE: console, LABELS: app=php-helloworld. On the right, under 'LOCATION', it shows the external route: http://php-helloworld-console.apps.cluster.lab.example.com. Under 'STATUS', it shows Accepted. Under 'HOSTNAME', it shows php-helloworld-console.apps.cluster.lab.example.com.

Figure 6.29: Route overview page

Explore the available information from the Overview tab. The LOCATION field provides a link to the external route for the application; <http://php-helloworld-console.apps.cluster.lab.example.com>.

`helloworld-console.apps.cluster.lab.example.com`. Click the link to access the application in a new tab:



Hello, World! php version is 7.0.27

Figure 6.30: Initial PHP application results

- ▶ 4. Modify the application code, commit the change, push the code to the remote Git repository, and trigger a new application build.

- 4.1. Clone the Git repository:

```
[student@workstation ~]$ git clone \
http://services.lab.example.com/php-helloworld
Cloning into 'php-helloworld'...
remote: Counting objects: 9, done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 9 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (9/9), done.
[student@workstation ~]$ cd php-helloworld
```

- 4.2. Add the second print line statement in the `index.php` page to read "A change is in the air!" and save the file. Add the change to the Git index, commit the change, and push the changes to the remote Git repository.

```
[student@workstation php-helloworld]$ vim index.php
[student@workstation php-helloworld]$ cat index.php
<?php
print "Hello, World! php version is " . PHP_VERSION . "\n";
print "A change is in the air!\n";
?>
[student@workstation php-helloworld]$ git add index.php
[student@workstation php-helloworld]$ git commit -m 'updated app'
[master d198fb5] updated app
...output omitted...
1 file changed, 1 insertion(+), 1 deletion(-)
[student@workstation php-helloworld]$ git push origin master
Counting objects: 5, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 286 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To http://services.lab.example.com/php-helloworld
```

78e0f96..d198fb5 master -> master

4.3. Trigger an application build manually from the web console.

On the navigation bar, click Builds → Build Configs and then click the **php-helloworld** link to access the Build Config Overview page. From the Actions menu in the upper right of the screen, click Start Build:

The screenshot shows the Red Hat OpenShift web interface. On the left, there's a sidebar with options like Home, Catalog, Workloads, Networking, Storage, Builds (selected), and Build Configs. Under Builds, there's a sub-option 'Builds'. The main area is titled 'Build Config Overview' for a project named 'console'. It shows details for a build config named 'php-helloworld' in the 'NS console' namespace. The build type is 'Source' and the git repository is 'http://services.lab.example.com/php-helloworld'. On the right, there's a 'Actions' dropdown menu with options: Start Build (highlighted), Edit Environment, Edit Labels, Edit Annotations, Edit Build Config, and Delete Build Config.

Figure 6.31: Start an application build

You are redirected to a Build Overview page for the new build. Click the Logs tab to monitor progress of the build. The last line of a successful build contains **Push successful**. Proceed to the next step when the build finishes.

4.4. Reload the `http://php-helloworld-console.apps.cluster.lab.example.com` URL in the browser. The application response corresponds to the updated source code:

The screenshot shows a Mozilla Firefox browser window. The address bar shows the URL `php-helloworld-console.apps.cluster.lab.example.com`. The page content displays the message "Hello, World! php version is 7.0.27 A change is in the air!"

Figure 6.32: Updated web application output

► 5. Delete the project. On the navigation bar, click Home → Projects. Click the icon at the right side of the row containing an entry for the **console** project. Click Delete Project from the menu that appears.

The screenshot shows the Red Hat OpenShift web interface. On the left, there's a sidebar with options like Home, Projects (selected), Status, Search, Events, Catalog, Workloads, and Networking. The main area is titled 'Projects' and shows a table of existing projects. There are three projects listed: 'console' (Active, requester 'kubeadmin', no labels), 'default' (Active, requester 'No requester', no labels), and 'kube-public' (Active, requester 'No requester', no labels). On the right, there's a 'Delete Project' button for the 'console' project, which is highlighted.

Figure 6.33: Delete a project

Enter **console** in the confirmation dialog box, and click **Delete**.

Finish

On workstation, run the **lab openshift-webconsole finish** script to complete this lab.

```
[student@workstation ~]$ lab openshift-webconsole finish
```

This concludes the guided exercise.

► LAB

DEPLOYING CONTAINERIZED APPLICATIONS ON OPENSIFT

PERFORMANCE CHECKLIST

In this lab, you will create an application using the OpenShift Source-to-Image facility.

OUTCOMES

You should be able to create an OpenShift application and access it through a web browser.

BEFORE YOU BEGIN

Open a terminal on workstation as the student user and run the following command:

```
[student@workstation ~]$ lab openshift-review start
```

1. Log in as the `kubeadmin` user and create the project `ocp`.
If the `oc login` command warns about using insecure connections, answer `y` (yes).
2. Create a temperature converter application written in PHP using the `php:7.1` image stream tag. The source code is in the Git repository at `http://services.lab.example.com/temp`s. You may use the OpenShift command-line interface or the web console to create the application.
Expose the application's service to make the application accessible from a web browser.
3. Verify that you can access the application in a web browser at `http://temp-ocp.apps.lab.example.com`.

Evaluation

On workstation, run the `lab openshift-review grade` command to grade your work. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab openshift-review grade
```

Finish

On workstation, run the `lab openshift-review finish` command to complete this lab.

```
[student@workstation ~]$ lab openshift-review finish
```

This concludes the lab.

► SOLUTION

DEPLOYING CONTAINERIZED APPLICATIONS ON OPENSHIFT

PERFORMANCE CHECKLIST

In this lab, you will create an application using the OpenShift Source-to-Image facility.

OUTCOMES

You should be able to create an OpenShift application and access it through a web browser.

BEFORE YOU BEGIN

Open a terminal on workstation as the student user and run the following command:

```
[student@workstation ~]$ lab openshift-review start
```

1. Log in as the `kubeadmin` user and create the project `ocp`.

If the `oc login` command warns about using insecure connections, answer **y** (yes).

Run the following commands:

```
[student@workstation ~]$ oc login -u kubeadmin \
> -p $(cat /home/student/.kubeadmin) https://cluster-api.lab.example.com:6443
Login successful.
```

You have access to the following projects and can switch between them with 'oc project <projectname>':

```
* default
...output omitted...
```

Using project "default".

```
[student@workstation ~]$ oc new-project ocp
```

Now using project "ocp" on server "https://cluster-api.lab.example.com:6443".

You can add applications to this project with the 'new-app' command. For example, try:

```
oc new-app centos/ruby-25-centos7~https://github.com/sclorg/ruby-ex.git
```

to build a new example application in Ruby.

2. Create a temperature converter application written in PHP using the `php:7.1` image stream tag. The source code is in the Git repository at `http://services.lab.example.com/temp`s. You may use the OpenShift command-line interface or the web console to create the application.

Expose the application's service to make the application accessible from a web browser.

2.1. If using the command-line interface, run the following commands:

```
[student@workstation ~]$ oc new-app \
  php:7.1~http://services.lab.example.com/temps
--> Found image 6e29dfe (8 days old) in image stream "openshift/php" under tag
"7.1" for "php:7.1"

  Apache 2.4 with PHP 7.1
  -----
    PHP 7.1 available as container is a base platform ...output omitted...
...output omitted...

--> Creating resources ...
  imagestream.image.openshift.io "temps" created
  buildconfig.build.openshift.io "temps" created
  deploymentconfig.apps.openshift.io "temps" created
  service "temps" created
--> Success
  Build scheduled, use 'oc logs -f bc/temps' to track its progress.
  Application is not exposed. You can expose services to the outside world by
  executing one or more of the commands below:
    'oc expose svc/temps'
  Run 'oc status' to view your app.
```

2.2. Monitor progress of the build.

```
[student@workstation ~]$ oc logs -f bc/temps
Cloning "http://services.lab.example.com/temps" ...
Commit: 350b6ca43ff05d1c395a658083f74a92c53fc7e9 (Establish remote repository)
Author: root <root@services.lab.example.com>
Date: Tue Jun 26 21:59:41 2018 +0000
...output omitted...
Successfully pushed //image-registry.openshift-image-registry.svc:5000/ocp/t...
Push successful
```

2.3. Verify that the application is deployed.

```
[student@workstation ~]$ oc get pods -w
NAME        READY     STATUS    RESTARTS   AGE
temps-1-build  0/1      Completed  0          5m
temps-1-h761t  1/1      Running   0          5m
```

Press **Ctrl+C** to exit the **oc get pods -w** command.

2.4. Expose the temps service to create an external route for the application.

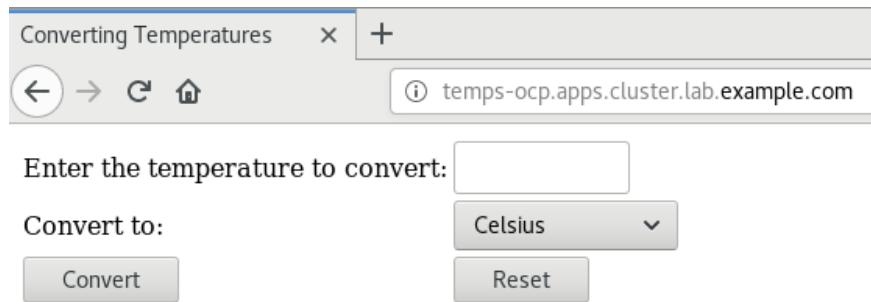
```
[student@workstation ~]$ oc expose svc/temps
route.route.openshift.io/temps exposed
```

3. Verify that you can access the application in a web browser at `http://temps-ocp.apps.lab.example.com`.

- 3.1. Determine the URL for the route.

```
[student@workstation ~]$ oc get route/temps
NAME      HOST/PORT
temps     temps-ocp.apps.lab.example.com ...
```

- 3.2. Open Firefox and navigate to `http://temps-ocp.apps.cluster.lab.example.com` to verify that the temperature converter application works.



Evaluation

On workstation, run the `lab openshift-review grade` command to grade your work. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab openshift-review grade
```

Finish

On workstation, run the `lab openshift-review finish` command to complete this lab.

```
[student@workstation ~]$ lab openshift-review finish
```

This concludes the lab.

SUMMARY

In this chapter, you learned:

- OpenShift Container Platform stores definitions of each OpenShift or Kubernetes resource instance as an object in the cluster's distributed database service, etcd. The most common OpenShift and Kubernetes resource types are: Pod, Persistent Volume (PV), Persistent Volume Claim (PVC), Service (SVC), Route, Deployment Configuration (DC), and Build Configuration (BC).
- The OpenShift command-line client **oc** is used to perform the following tasks in an OpenShift cluster:
 - Creating, changing, and deleting projects.
 - Creating application resources inside a project.
 - Deleting, inspecting, editing, and exporting resources inside a project.
 - Checking logs from application pods, deployments, and build operations.
- The **oc new-app** command can create application pods to run on OpenShift in many different ways. It can create pods from an existing container image hosted on an image registry, from Dockerfiles, and from source code using the Source-to-Image (S2I) process.
- Source-to-Image (S2I) is a tool that makes it easy to build a container image from application source code. This tool retrieves source code from a Git repository, injects the source code into a selected container image based on a desired language or technology, and produces a new container image that runs the assembled application.
- A Route connects a public-facing IP address and DNS host name to an internal-facing service IP. While services allow for network access between pods inside an OpenShift instance, routes allow for network access to pods from users and applications outside the OpenShift instance.
- You can create, build, deploy and monitor applications using the OpenShift web console.

CHAPTER 7

DEPLOYING MULTI-CONTAINER APPLICATIONS

GOAL

Deploy applications that are containerized using multiple container images.

OBJECTIVES

- Describe considerations for containerizing applications with multiple container images.
- Deploy a multi-container application on OpenShift using a template.

SECTIONS

- Considerations for Multi-Container Applications (and Guided Exercise)
- Deploying a Multi-Container Application on OpenShift (and Guided Exercise)

LAB

- Deploying Multi-Container Applications

CONSIDERATIONS FOR MULTI-CONTAINER APPLICATIONS

OBJECTIVES

After completing this section, students should be able to:

- Describe considerations for containerizing applications with multiple container images.
- Leverage networking concepts in containers.
- Create a multi-container application with Podman.
- Describe the architecture of the To Do List application.

LEVERAGING MULTI-CONTAINER APPLICATIONS

The examples shown so far throughout this course have worked fine with a single container. A more complex application, however, can get the benefits of deploying different components into different containers. Consider an application composed of a front-end web application, a REST back end, and a database server. Those components may have different dependencies, requirements and life cycles.

Although it is possible to orchestrate multi-container applications' containers manually, Kubernetes and OpenShift provide tools to facilitate orchestration. Attempting to manually manage dozens or hundreds of containers quickly becomes complicated. In this section, we are going to return to using Podman to create a simple multi-container application to demonstrate the underlying manual steps for container orchestration. In later sections, you will use Kubernetes and OpenShift to orchestrate these same application containers.

DISCOVERING SERVICES IN A MULTI-CONTAINER APPLICATION

Podman uses *Container Network Interface (CNI)* to create a *software-defined network (SDN)* between all containers in the host. Unless stated otherwise, CNI assigns a new IP address to a container when it starts.

Each container exposes all ports to other containers in the same SDN. As such, services are readily accessible within the same network. The containers expose ports to external networks only by explicit configuration.

Due to the dynamic nature of container IP addresses, applications cannot rely on either fixed IP addresses or fixed DNS host names to communicate with middleware services and other application services. Containers with dynamic IP addresses can become a problem when working with multi-container applications because each container must be able to communicate with others to use services upon which it depends.

For example, consider an application composed of a front-end container, a back-end container, and a database. The front-end container needs to retrieve the IP address of the back-end container. Similarly, the back-end container needs to retrieve the IP address of the database container. Additionally, the IP address could change if a container restarts, so a process is needed to ensure any change in IP triggers an update to existing containers.

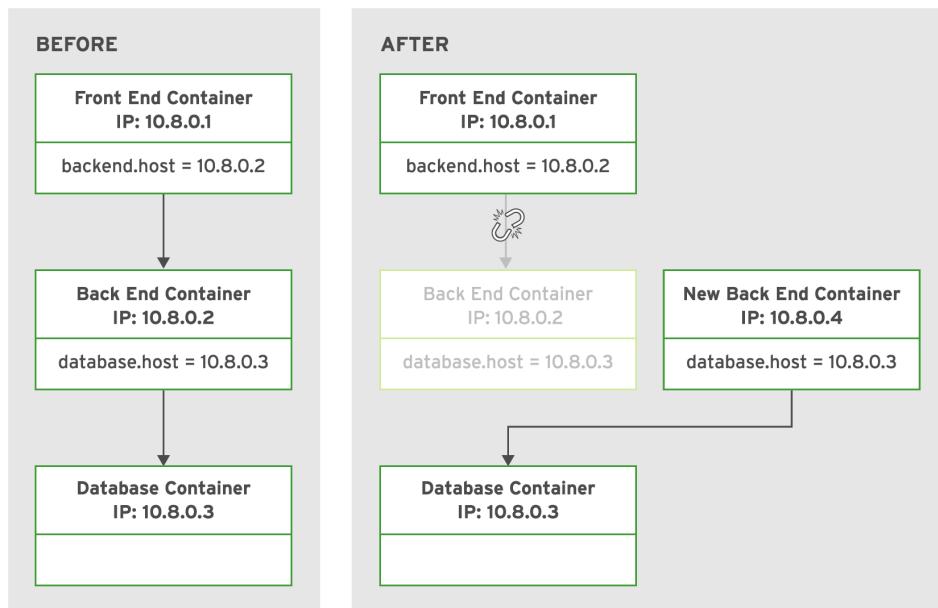


Figure 7.1: A restart breaks three-tiered application links

Both Kubernetes and OpenShift provide potential solutions to the issue of service discoverability and the dynamic nature of container networking. Some of these solutions are covered later in the chapter.

COMPARING PODMAN AND KUBERNETES

Using environment variables allows you to share information between containers with Podman. However, there are still some limitations and some manual work involved in ensuring that all environment variables stay in sync, especially when working with many containers. Kubernetes provides an approach to solve this problem by creating services for your containers, as covered in previous chapters.

Pods are attached to a Kubernetes namespace, which OpenShift calls a *project*. When a pod starts, Kubernetes automatically adds a set of environment variables for each service defined on the same namespace.

Any service defined on Kubernetes generates environment variables for the IP address and port number where the service is available. Kubernetes automatically injects these environment variables into the containers from pods in the same namespace. These environment variables usually follow a convention:

- **Uppercase:** All environment variables are set using uppercase names.
- **Snakecase:** Any environment variable created by a service is usually composed of multiple words separated with an underscore (_).
- **Service name first:** The first word for an environment variable created by a service is the service name.
- **Protocol type:** Most network environment variables include the protocol type (TCP or UDP).

These are the environment variables generated by Kubernetes for a service:

- <SERVICE_NAME>_SERVICE_HOST: Represents the IP address enabled by a service to access a pod.

- <SERVICE_NAME>_SERVICE_PORT: Represents the port where the server port is listed.
- <SERVICE_NAME>_PORT: Represents the address, port, and protocol provided by the service for external access.
- <SERVICE_NAME>_PORT_<PORT_NUMBER>_<PROTOCOL>: Defines an alias for the <SERVICE_NAME>_PORT.
- <SERVICE_NAME>_PORT_<PORT_NUMBER>_<PROTOCOL>_PROTO: Identifies the protocol type (TCP or UDP).
- <SERVICE_NAME>_PORT_<PORT_NUMBER>_<PROTOCOL>_PORT: Defines an alias for <SERVICE_NAME>_SERVICE_PORT.
- <SERVICE_NAME>_PORT_<PORT_NUMBER>_<PROTOCOL>_ADDR: Defines an alias for <SERVICE_NAME>_SERVICE_HOST.

For instance, given the following service:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    name: mysql
    name: mysql
spec:
  ports:
    - protocol: TCP
      - port: 3306
  selector:
    name: mysql
```

The following environment variables are available for each pod created after the service, on the same namespace:

```
MYSQL_SERVICE_HOST=10.0.0.11
MYSQL_SERVICE_PORT=3306
MYSQL_PORT=tcp://10.0.0.11:3306
MYSQL_PORT_3306_TCP=tcp://10.0.0.11:3306
MYSQL_PORT_3306_TCP_PROTO=tcp
MYSQL_PORT_3306_TCP_PORT=3306
MYSQL_PORT_3306_TCP_ADDR=10.0.0.11
```



NOTE

Other relevant <SERVICE_NAME>_PORT_* environment variable names are set on the basis of the protocol. IP address and port number are set in the <SERVICE_NAME>_PORT environment variable. For example, **MYSQL_PORT=tcp://10.0.0.11:3306** entry leads to the creation of environment variables with names such as **MYSQL_PORT_3306_TCP**, **MYSQL_PORT_3306_TCP_PROTO**, **MYSQL_PORT_3306_TCP_PORT**, and **MYSQL_PORT_3306_TCP_ADDR**. If the protocol component of an environment variable is undefined, Kubernetes uses the TCP protocol and assigns the variable names accordingly.

DESCRIBING THE TO DO LIST APPLICATION

Many labs from this course make use of a To Do List application. This application is divided into three tiers, as illustrated by the following figure:

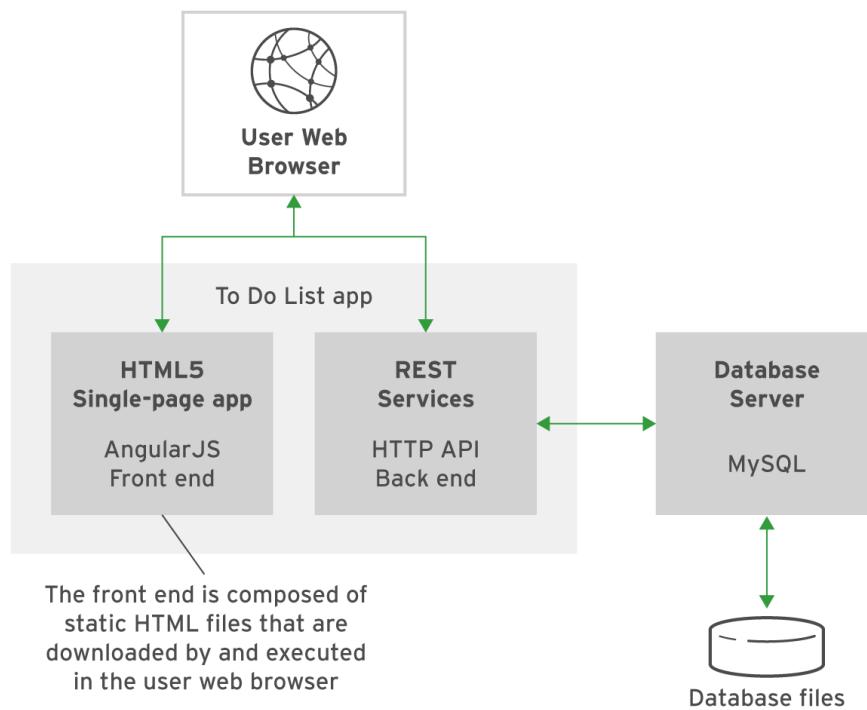


Figure 7.2: To Do List application logical architecture

- The presentation tier is built as a single-page HTML5 front-end using AngularJS.
- The business tier is composed by an HTTP API back-end, with Node.js.
- The persistence tier based on a MySQL database server.

The following figure is a screen capture of the application web interface:

To Do List Application

The screenshot shows two parts of a web application. On the left, a table titled "To Do List" displays two items:

ID	Description	Done
1	Pick up new...	false
2	Buy groceries	true

On the right, a form titled "Add Task" allows adding a new task:

Description: Add Description.

Completed:

Buttons: Clear Save

Below the table are navigation buttons: First, Previous, 1, Next, Last.

Figure 7.3: The To Do List application

On the left is a table with items to complete, and on the right is a form to add a new item.

The classroom private registry server, `services.lab.example.com`, provides the application in two versions:

nodejs

Represents the way a typical developer would create the application as a single unit, without caring to break it into tiers or services.

nodejs_api

Shows the changes needed to break the application into presentation and business tiers. Each tier corresponds to an isolated container image.

The sources of both of these application versions are available from the `todoapp` Git repository at:
<http://services.lab.example.com/todoapp>

► GUIDED EXERCISE

DEPLOYING THE WEB APPLICATION AND MYSQL CONTAINERS

In this lab, you will create a script that runs and networks a Node.js application container and the MySQL container.

OUTCOMES

You should be able to network containers to create a multi-tiered application.

BEFORE YOU BEGIN

You must have the To Do List application source code and lab files on workstation. To set up the environment for the exercise, run the following command:

```
[student@workstation ~]$ lab multicontainer-design start
```

► 1. Build the MySQL image.

A custom MySQL 5.7 image is used for this exercise. It is configured to automatically run any scripts in the **/var/lib/mysql/init** directory. The scripts load the schema and some sample data into the database for the To Do List application when a container starts.

1.1. Review the Dockerfile.

Using your preferred editor, open and examine the completed Dockerfile located at **/home/student/D0180/labs/multicontainer-design/images/mysql/Dockerfile**.

1.2. Build the MySQL database image.

To build the base image, run the following **podman build** script.

```
[student@workstation ~]$ cd ~/D0180/labs/multicontainer-design/images/mysql
[student@workstation mysql]$ sudo podman build -t do180/mysql-57-rhel7 \
> --layers=false .
STEP 1: FROM rhscl/mysql-57-rhel7
Getting image source signatures
...output omitted...
Storing signatures
STEP 2: ADD root /
ERRO[0017] HOSTNAME is not supported for OCI image format, hostname d4f8f8506f2b
will be ignored. Must use `docker` format
--> 41a68cfcd248dd79133f9f60fed563bf160352181ea946d44d6b756e41b89cc7e
STEP 3: COMMIT do180/mysql-57-rhel7
...output omitted...
Writing manifest to image destination
Storing signatures
```

```
--> 8dc111531fce3592f6fdded1881ed26482f8c9a5fbb5fd95ac58795bd2bf6933
```

The error shown is benign, and can be ignored. It will disappear in later versions of Podman.

- 1.3. This command builds a dedicated MySQL container image based on the **~/D0180/labs/multicontainer-design/images/mysql** context folder and the **Dockerfile** file in it. Wait for the build to complete, and then run the following command to verify that the image is built successfully:

```
[student@workstation mysql]$ sudo podman images
REPOSITORY                                TAG      IMAGE ID      CREATED        SIZE
localhost/do180/mysql-57-rhel7            latest   8dc111531fce  21 seconds ago  444MB
registry.[...]/rhscl/mysql-57-rhel7       latest   c07bf25398f4  4 weeks ago    444MB
```

► 2. Build the Node.js parent image using the provided Dockerfile.

- 2.1. Review the Dockerfile.

Using your preferred editor, open and examine the completed Dockerfile located at **/home/student/D0180/labs/multicontainer-design/images/nodejs/Dockerfile**.

Notice the following instructions defined in the Dockerfile:

- Two environment variables, **NODEJS_VERSION** and **HOME**, are defined using the **ENV** command.
- A custom Yum repo pointing to the local offline repository is added using the **ADD** command.
- Packages necessary for Node.js are installed with **yum** using the **RUN** command.
- A new user and group are created to run the Node.js application along with the **app-root** directory using the **RUN** command.
- The **enable-rh-nodejs4.sh** script, to run automatically at login, is added to **/etc/profile.d/** with the **ADD** command.
- The **USER** command is used to switch to the newly created **appuser** account.
- The **WORKDIR** command is used to switch to the **\$HOME** directory for application execution.

- 2.2. Build the parent image.

To build the base image, run the **podman build** command.

```
[student@workstation ~]$ cd ~/D0180/labs/multicontainer-design/images/nodejs
[student@workstation nodejs]$ sudo podman build -t do180/nodejs \
> --layers=false .
STEP 1: FROM rhel7:7.5
Getting image source signatures
...output omitted...
--> Finished Dependency Resolution
```

Dependencies Resolved

```
=====
Package           Arch    Version      Repository      Size
=====
Installing:
rh-nodejs4        x86_64  2.4-3.el7   rhel-server-rhscl-7-rpms 6.5 k
...output omitted...
Writing manifest to image destination
Storing signatures
--> 5e610ea8d94a14eba057696cb5758cbe1ccbf3d58b5de78d898e16fb9aa430de
```

- 2.3. Wait for the build to complete, and then run the following command to verify that the image has been built successfully. Several columns of the **podman images** column are not relevant, so you can use the **--format** option to format the output.

```
[student@workstation nodejs]$ sudo podman images \
> --format "table {{.ID}} {{.Repository}} {{.Tag}}"
IMAGE ID          REPOSITORY          TAG
78c2e6304f23    localhost/do180/mysql-57-rhel7    latest
5e610ea8d94a  localhost/do180/nodejs          latest
c07bf25398f4    registry.access.redhat.com/rhscl/mysql-57-rhel7  latest
7b875638cf8     registry.access.redhat.com/rhel7       7.5
```

- 3. Build the To Do List application child image using the provided Dockerfile.
- 3.1. Review the Dockerfile.
Using your preferred editor, open and examine the completed Dockerfile located at **/home/student/D0180/labs/multicontainer-design/deploy/nodejs/Dockerfile**.
- 3.2. Explore the Environment Variables.
Inspect the environment variables that allow the Node.js REST API container to communicate with the MySQL container.
- View the file **/home/student/D0180/labs/multicontainer-design/deploy/nodejs/nodejs-source/models/db.js**, containing the database configuration provided below:

```
module.exports.params = {
  dbname: process.env.MYSQL_DATABASE,
  username: process.env.MYSQL_USER,
  password: process.env.MYSQL_PASSWORD,
  params: {
    host: "10.88.100.101",
    port: "3306",
    dialect: 'mysql'
  }
}
```

```
};
```

2. Notice the environment variables used by the REST API. These variables are exposed to the container using **-e** options with the **podman run** command in this guided exercise. Those environment variables are described below.

MYSQL_DATABASE

The name of the MySQL database in the `mysql` container.

MYSQL_USER

The name of the database user used by the `todoapi` container to run MySQL commands.

MYSQL_PASSWORD

The password of the database user that the `todoapi` container uses to authenticate to the `mysql` container.



NOTE

The host and port details of the MySQL container are embedded with the REST API application. The host, as shown above in the `db.js` file, is the IP address of the `mysql` container.

- 3.3. Build the child image.

Examine the `/home/student/D0180/labs/multicontainer-design/deploy/nodejs/build.sh` script to see how the image is built. Run the following commands to build the child image.

```
[student@workstation nodejs]$ cd ~/D0180/labs/multicontainer-design/deploy/nodejs  
[student@workstation nodejs]$ ./build.sh  
STEP 1: FROM do180/nodejs  
STEP 2: MAINTAINER username <username@example.com>  
STEP 3: COPY run.sh build ${HOME}/  
...output omitted...  
Writing manifest to image destination  
Storing signatures  
--> 2b127523c28ecba38d05537588fc9ae0bb77f9426224b436557197e8caea6463
```



NOTE

The `build.sh` script lowers restrictions for write access to the build directory, allowing non-root users to install dependencies.

- 3.4. Wait for the build to complete and then run the following command to verify that the image has been built successfully:

```
[student@workstation nodejs]$ sudo podman images \  
> --format "table {{.ID}} {{.Repository}} {{.Tag}}"  
IMAGE ID      REPOSITORY                      TAG  
2b127523c28e  localhost/do180/todonodejs    latest  
6bf46c84a3c5   localhost/do180/nodejs        latest  
fa0084527d05   localhost/do180/mysql-57-rhel7  latest  
c07bf25398f4   registry.access.redhat.com/rhscl/mysql-57-rhel7  latest
```

- ▶ 4. Modify the existing script to create containers with the appropriate IP, as defined in the previous step. In this script, the order of commands is given such that it starts the mysql container and then starts the todoapi container before connecting it to the mysql container. After invoking every container, there is a wait time of 9 seconds, so each container has time to start.
- 4.1. Edit the **run.sh** file located at **~/D0180/labs/multicontainer-design/deploy/nodejs/networked** to insert the **podman run** command at the appropriate line for invoking mysql container. The following screen shows the exact **podman** command to insert into the file.

```
sudo podman run -d --name mysql -e MYSQL_DATABASE=items -e MYSQL_USER=user1 \
-e MYSQL_PASSWORD=mypa55 -e MYSQL_ROOT_PASSWORD=r00tpa55 \
-v $PWD/work/data:/var/lib/mysql/data \
-v $PWD/work/init:/var/lib/mysql/init -p 30306:3306 \
--ip 10.88.100.101 do180/mysql-57-rhel7
```

In the previous command, the **MYSQL_DATABASE**, **MYSQL_USER**, and **MYSQL_PASSWORD** are populated with the credentials to access the MySQL database. These environment variables are required for the mysql container to run. Also, the **\$PWD/work/data** and **\$PWD/work/init** local folders are mounted as volumes into the container's file system.

Note the IP assigned to the container. This IP should be the same as the one provided in the **/home/student/D0180/labs/multicontainer-design/deploy/nodejs/nodejs-source/models/db.js** file.

- 4.2. In the same **run.sh** file, insert another **podman run** command at the appropriate line to run the todoapi container. The following screen shows the **docker** command to insert into the file.

```
sudo podman run -d --name todoapi -e MYSQL_DATABASE=items -e MYSQL_USER=user1 \
-e MYSQL_PASSWORD=mypa55 -p 30080:30080 \
do180/todonodejs
```



NOTE

After each **podman run** command inserted into the **run.sh** script, ensure that there is also a **sleep 9** command. If you need to repeat this step, the **work** directory and its contents must be deleted before re-running the **run.sh** script.

- 4.3. Verify that your **run.sh** script matches the solution script located at **/home/student/D0180/solutions/multicontainer-design/deploy/nodejs/networked/run.sh**.
- 4.4. Save the file and exit the editor.

► 5. Run the containers.

- 5.1. Use the following command to execute the script that you updated to run the `mysql` and `todoapi` containers.

```
[student@workstation nodejs]$ cd \
/home/student/D0180/labs/multicontainer-design/deploy/nodejs/networked
[student@workstation networked]$ ./run.sh
```



NOTE

It is possible that the IP selected for the container (10.88.100.101) is already reserved for another container, even if that container was already deleted. In this case, delete the `todonodejs` image and container before creating an updated one with the `sudo podman rmi -f todonodejs` command. Then delete the MySQL container with the `sudo podman rm -f mysql` command. Afterwards, return to step 3, and update both `db.js` and `run.sh` with another available IP.

- 5.2. Verify that the containers started successfully.

```
[student@workstation networked]$ sudo podman ps \
> --format="table {{.ID}} {{.Names}} {{.Image}} {{.Status}}"
CONTAINER ID NAMES IMAGE STATUS
c74b4709e3ae todoapi localhost/do180/todonodejs:latest Up 3 minutes ago
3bc19f74254c mysql localhost/do180/mysql-57-rhel7:latest Up 3 minutes ago
```

► 6. Examine the environment variables of the API container.

Run the following command to explore the environment variables exposed in the API container.

```
[student@workstation networked]$ sudo podman exec -it todoapi env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=2c61a089105d
TERM=xterm
MYSQL_DATABASE=items
MYSQL_USER=user1
MYSQL_PASSWORD=mypa55
container=oci
NODEJS_VERSION=4.0
HOME=/opt/app-root/src
```

► 7. Test the application.

- 7.1. Run a `curl` command to test the REST API for the To Do List application.

```
[student@workstation networked]$ curl -w "\n" \
> http://127.0.0.1:30080/todo/api/items/1
```

```
{"description": "Pick up newspaper", "done": false, "id":1}
```

The **-w "\n"** option with **curl** command lets the shell prompt appear at the next line rather than merging with the output in the same line.

- 7.2. Open Firefox on **workstation** and point your browser to <http://127.0.0.1:30080/todo/>. You should see the To Do List application.

**NOTE**

Make sure to append the trailing slash (/).

Finish

On **workstation**, run the **lab multicontainer-design finish** script to complete this exercise.

```
[student@workstation ~]$ lab multicontainer-design finish
```

This concludes the guided exercise.

DEPLOYING A MULTI-CONTAINER APPLICATION ON OPENSHIFT

OBJECTIVES

After completing this section, students should be able to deploy a multicontainer application on OpenShift using a template.

EXAMINING THE SKELETON OF A TEMPLATE

Deploying an application on OpenShift Container Platform often requires creating several related resources within a Project. For example, a web application may require a **BuildConfig**, **DeploymentConfig**, **Service**, and **Route** resource to run in an OpenShift project. Often the attributes of these resources have the same value, such as a resource's name attribute.

OpenShift *templates* provide a way to simplify the creation of resources that an application requires. A template defines a set of related resources to be created together, as well as a set of application parameters. The attributes of template resources are typically defined in terms of the template parameters, such as a resource's name attribute.

For example, an application might consist of a front-end web application and a database server. Each consists of a service resource and a deployment configuration resource. They share a set of credentials (parameters) for the front end to authenticate to the back end. The template can be processed by specifying parameters or by allowing them to be automatically generated (for example, for a unique database password) in order to instantiate the list of resources in the template as a cohesive application.

The OpenShift installer creates several templates by default in the **openshift** namespace. Run the **oc get templates** command with the **-n openshift** option to list these preinstalled templates:

```
[student@workstation ~]$ oc get templates -n openshift
NAME                  DESCRIPTION
cakephp-mysql-example   An example CakePHP application ...
cakephp-mysql-persistent  An example CakePHP application ...
dancer-mysql-example     An example Dancer application with a MySQL ...
dancer-mysql-persistent  An example Dancer application with a MySQL ...
django-psql-example      An example Django application with a PostgreSQL ...
...output omitted...
rails-psql-persistent    An example Rails application with a PostgreSQL ...
rails-postgresql-example An example Rails application with a PostgreSQL ...
redis-ephemeral          Redis in-memory data structure store, ...
redis-persistent          Redis in-memory data structure store, ...
```

The following shows a YAML template definition:

```
$ oc get template mysql-persistent -n openshift -o yaml
apiVersion: template.openshift.io/v1
kind: Template
labels: ...value omitted...
message: ...message omitted ...
```

```
metadata:
  annotations:
    description: ...description omitted...
    iconClass: icon-mysql-database
    openshift.io/display-name: MySQL
    openshift.io/documentation-url: ...value omitted...
    openshift.io/long-description: ...value omitted...
    openshift.io/provider-display-name: Red Hat, Inc.
    openshift.io/support-url: https://access.redhat.com

  tags: database,mysql ❶
  labels: ...value omitted...

  name: mysql-persistent ❷

objects: ❸
- apiVersion: v1
  kind: Secret
  metadata:
    annotations: ...annotations omitted...
    name: ${DATABASE_SERVICE_NAME} ❹
  stringData: ...stringData omitted...
- apiVersion: v1
  kind: Service
  metadata:
    annotations: ...annotations omitted...
    name: ${DATABASE_SERVICE_NAME}
  spec: ...spec omitted...
- apiVersion: v1
  kind: PersistentVolumeClaim
  metadata:
    name: ${DATABASE_SERVICE_NAME}
  spec: ...spec omitted...
- apiVersion: v1
  kind: DeploymentConfig
  metadata:
    annotations: ...annotations omitted...
    name: ${DATABASE_SERVICE_NAME}
  spec: ...spec omitted...

parameters: ❺
- ...MEMORY_LIMIT parameter omitted...
- ...NAMESPACE parameter omitted...
- description: The name of the OpenShift Service exposed for the database.
  displayName: Database Service Name
  name: DATABASE_SERVICE_NAME ❻
  required: true
  value: mysql
- ...MYSQL_USER parameter omitted...
- description: Password for the MySQL connection user.
  displayName: MySQL Connection Password
  from: '[a-zA-Z0-9]{16}' ❼
  generate: expression
  name: MYSQL_PASSWORD
  required: true
- ...MYSQL_ROOT_PASSWORD parameter omitted...
```

```
- ...MYSQL_DATABASE parameter omitted...
- ...VOLUME_CAPACITY parameter omitted...
- ...MYSQL_VERSION parameter omitted...
```

- ➊ Defines a list of arbitrary tags to associate with this template. Enter any of these tags in the UI to find this template.
- ➋ Defines the template name.
- ➌ The **objects** section defines the list of OpenShift resources for this template. This template creates four resources: a **Secret**, a **Service**, a **PersistentVolumeClaim**, and a **DeploymentConfig**.
- ➍ All four resource objects have their names set to the value of the **DATABASE_SERVICE_NAME** parameter.
- ➎ ➏ The **parameters** section contains a list of nine parameters. Template resources often define their attributes using the values of these parameters, as demonstrated with the **DATABASE_SERVICE_NAME** parameter.
- ➐ If you do not specify a value for the **MYSQL_PASSWORD** parameter when you create an application with this template, OpenShift generates a password that matches this regular expression.

You can publish a new template to the OpenShift cluster so that other developers can build an application from the template.

Assume you have an task list application named **todo** that requires an OpenShift **DeploymentConfig**, **Service**, and **Route** object for deployment. You create a YAML template definition file that defines attributes for these OpenShift resources, along with definitions for any required parameters. Assuming the template is defined in the **todo-template.yaml** file, use the **oc create** command to publish the application template:

```
[student@workstation deploy-multicontainer]$ oc create -f todo-template.yaml
template.template.openshift.io/todonodejs-persistent created
```

By default, the template is created under the current project unless you specify a different one using the **-n** option, as shown in the following example:

```
[student@workstation deploy-multicontainer]$ oc create -f todo-template.yaml \
> -n openshift
```



IMPORTANT

Any template created under the **openshift** namespace (OpenShift project) is available in the web console under the dialog box accessible in the Catalog → Developer Catalog menu item. Moreover, any template created under the current project is accessible from that project.

Parameters

Templates define a set of *parameters*, which are assigned values. OpenShift resources defined in the template can get their configuration values by referencing *named parameters*. Parameters in a template can have default values, but they are optional. Any default value can be replaced when processing the template.

Each parameter value can be set either explicitly by using the **oc process** command, or generated by OpenShift according to the parameter configuration.

There are two ways to list available parameters from a template. The first one is using the **oc describe** command:

```
$ oc describe template mysql-persistent -n openshift
Name: mysql-persistent
Namespace: openshift
Created: 12 days ago
Labels: samplesoperator.config.openshift.io/managed=true
Description: MySQL database service, with ...description omitted...
Annotations: iconClass=icon-mysql-database
              openshift.io/display-name=MySQL
              ...output omitted...
tags=database,mysql

Parameters:
  Name: MEMORY_LIMIT
  Display Name: Memory Limit
  Description: Maximum amount of memory the container can use.
  Required: true
  Value: 512Mi

  Name: NAMESPACE
  Display Name: Namespace
  Description: The OpenShift Namespace where the ImageStream resides.
  Required: false
  Value: openshift

  ...output omitted...

  Name: MYSQL_VERSION
  Display Name: Version of MySQL Image
  Description: Version of MySQL image to be used (5.7, or latest).
  Required: true
  Value: 5.7

Object Labels: template=mysql-persistent-template

Message: ...output omitted... in your project: ${DATABASE_SERVICE_NAME}.

  Username: ${MYSQL_USER}
  Password: ${MYSQL_PASSWORD}
  Database Name: ${MYSQL_DATABASE}
  Connection URL: mysql://${DATABASE_SERVICE_NAME}:3306/

For more information about using this template, ...output omitted...

Objects:
  Secret      ${DATABASE_SERVICE_NAME}
  Service     ${DATABASE_SERVICE_NAME}
  PersistentVolumeClaim ${DATABASE_SERVICE_NAME}
  DeploymentConfig ${DATABASE_SERVICE_NAME}
```

The second way is by using the **oc process** with the **--parameters** option:

```
$ oc process --parameters mysql-persistent -n openshift
NAME           DESCRIPTION      GENERATOR      VALUE
MEMORY_LIMIT   Maximum a...    expression     512Mi
NAMESPACE      The OpenS...    expression     openshift
DATABASE_SERVICE_NAME The name ...    expression     mysql
MYSQL_USER     Username ...    expression     user[A-Z0-9]{3}
MYSQL_PASSWORD Password ...    expression     [a-zA-Z0-9]{16}
MYSQL_ROOT_PASSWORD Password ...    expression     [a-zA-Z0-9]{16}
MYSQL_DATABASE Name of t...    expression     sampledb
VOLUME_CAPACITY Volume sp...    expression     1Gi
MYSQL_VERSION  Version o...    expression     5.7
```

PROCESSING A TEMPLATE USING THE CLI

When you process a template, you generate a list of resources to create a new application. To process a template, use the **oc process** command:

```
$ oc process -f <filename>
```

The previous command processes a template file, in either JSON or YAML format, and returns the list of resources to standard output. The format of the output resource list is JSON. To output the resource list in YAML format, use the **-o yaml** with the **oc process** command:

```
$ oc process -o yaml -f <filename>
```

Another option is to process a template from the current project or the **openshift** project:

```
$ oc process <uploaded-template-name>
```



NOTE

The **oc process** command returns a list of resources to standard output. This output can be redirected to a file:

```
$ oc process -o yaml -f filename > myapp.yaml
```

Templates often generate resources with configurable attributes that are based on the template parameters. To override a parameter, use the **-p** option followed by a **<name>=<value>** pair.

```
$ oc process -o yaml -f mysql.yaml \
> -p MYSQL_USER=dev -p MYSQL_PASSWORD=$P4SSD -p MYSQL_DATABASE=bank \
> -p VOLUME_CAPACITY=10Gi > mysqlProcessed.yaml
```

To create the application, use the generated YAML resource definition file:

```
$ oc create -f mysqlProcessed.yaml
```

Alternatively, it is possible to process the template and create the application without saving a resource definition file by using a UNIX pipe:

```
$ oc process -f mysql.yaml -p MYSQL_USER=dev \
> -p MYSQL_PASSWORD=$P4SSD -p MYSQL_DATABASE=bank \
> -p VOLUME_CAPACITY=10Gi | oc create -f -
```

To use a template in the **openshift** project to create an application in your project, first export the template:

```
$ oc get template mysql-persistent -o yaml \
> -n openshift > mysql-persistent-template.yaml
```

Next, identify appropriate values for the template parameters and process the template:

```
$ oc process -f mysql-persistent-template.yaml \
> -p MYSQL_USER=dev -p MYSQL_PASSWORD=$P4SSD -p MYSQL_DATABASE=bank \
> -p VOLUME_CAPACITY=10Gi | oc create -f -
```

You can also use two slashes (//) to provide the namespace as part of the template name:

```
$ oc process openshift//mysql-persistent \
> -p MYSQL_USER=dev -p MYSQL_PASSWORD=$P4SSD -p MYSQL_DATABASE=bank \
> -p VOLUME_CAPACITY=10Gi | oc create -f -
```

Alternatively, it is possible to create an application using the **oc new-app** command passing the template name as the **--template** option argument:

```
$ oc new-app --template=mysql-persistent \
> -p MYSQL_USER=dev -p MYSQL_PASSWORD=$P4SSD -p MYSQL_DATABASE=bank \
> -p VOLUME_CAPACITY=10Gi
```

CONFIGURING PERSISTENT STORAGE FOR OPENSHIFT APPLICATIONS

OpenShift Container Platform manages persistent storage as a set of pooled, cluster-wide resources. To add a storage resource to the cluster, an OpenShift administrator creates a **PersistentVolume** object that defines the necessary metadata for the storage resource. The metadata describes how the cluster accesses the storage, as well as other storage attributes such as capacity or throughput.

To list the **PersistentVolume** objects in a cluster, use the **oc get pv** command:

```
[admin@host ~]$ oc get pv
NAME      CAPACITY   ACCESS MODES  RECLAIM POLICY  STATUS      CLAIM      ...
pv0001    1Mi        RWO          Retain        Available   ...
pv0002    10Mi       RWX          Recycle       Available   ...
...output omitted...
```

To see the YAML definition for a given **PersistentVolume**, use the **oc get** command with the **-o yaml** option:

```
[admin@host ~]$ oc get pv pv0001 -o yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  creationTimestamp: ...value omitted...
  finalizers:
  - kubernetes.io/pv-protection
  labels:
    type: local
  name: pv0001
  resourceVersion: ...value omitted...
  selfLink: /api/v1/persistentvolumes/pv0001
  uid: ...value omitted...
spec:
  accessModes:
  - ReadWriteOnce
  capacity:
    storage: 1Mi
  hostPath:
    path: /data/pv0001
    type: ""
  persistentVolumeReclaimPolicy: Retain
status:
  phase: Available
```

To add more **PersistentVolume** objects to a cluster, use the **oc create** command:

```
[admin@host ~]$ oc create -f pv1001.yaml
```



NOTE

The above **pv1001.yaml** file must contain a persistent volume definition, similar in structure to the output of the **oc get pv *pv-name* -o yaml** command.

Requesting Persistent Volumes

When an application requires storage, you create a **PersistentVolumeClaim** (PVC) object to request a dedicated storage resource from the cluster pool. The following content from a file named **pvc.yaml** is an example definition for a PVC:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: myapp
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
```

```
storage: 1Gi
```

The PVC defines storage requirements for the application, such as capacity or throughput. To create the PVC, use the **oc create** command:

```
[admin@host ~]$ oc create -f pvc.yaml
```

After you create a PVC, OpenShift attempts to find an available **PersistentVolume** resource that satisfies the PVC's requirements. If OpenShift finds a match, it binds the PersistentVolume object to the PersistentVolumeClaim object. To list the PVCs in a project, use the **oc get pvc** command:

```
[admin@host ~]$ oc get pvc
```

NAME	STATUS	VOLUME	CAPACITY	ACCESS MODES	STORAGECLASS	AGE
myapp	Bound	pv0001	1Gi	RWO		6s

The output indicates whether a persistent volume is bound to the PVC, along with attributes of the PVC (such as capacity).

To use the persistent volume in an application pod, define a volume mount for a container that references the **PersistentVolumeClaim** object. The application pod definition below references a **PersistentVolumeClaim** object to define a volume mount for the application:

```
apiVersion: "v1"
kind: "Pod"
metadata:
  name: "myapp"
  labels:
    name: "myapp"
spec:
  containers:
    - name: "myapp"
      image: openshift/myapp
      ports:
        - containerPort: 80
          name: "http-server"
      volumeMounts:
        - mountPath: "/var/www/html"
          name: "pvol" ①
  volumes:
    - name: "pvol" ②
  persistentVolumeClaim:
    claimName: "myapp" ③
```

- ①** This section declares that the **pvol** volume mounts at **/var/www/html** in the container file system.
- ②** This section defines the **pvol** volume.
- ③** The **pvol** volume references the **myapp** PVC. If OpenShift associates an available persistent volume to the **myapp** PVC, then the **pvol** volume refers to this associated volume.

Configuring Persistent Storage with Templates

Templates are often used to simplify the creation of applications requiring persistent storage. Many of these templates have a suffix of **-persistent**:

```
[student@workstation ~]$ oc get templates -n openshift | grep persistent
cakephp-mysql-persistent      An example CakePHP application with a MySQL data...
dancer-mysql-persistent       An example Dancer application with a MySQL datab...
django-psql-persistent        An example Django application with a PostgreSQL ...
dotnet-psql-persistent        An example .NET Core application with a Postgres...
jenkins-persistent            Jenkins service, with persistent storage....
mariadb-persistent            MariaDB database service, with persistent storag...
mongodb-persistent            MongoDB database service, with persistent storag...
mysql-persistent               MySQL database service, with persistent storage....
nodejs-mongo-persistent       An example Node.js application with a MongoDB da...
postgresql-persistent         PostgreSQL database service, with persistent sto...
rails-psql-persistent         An example Rails application with a PostgreSQL d...
redis-persistent                Redis in-memory data structure store, with persi...
```

The following example template defines a **PersistentVolumeClaim** object along with a **DeploymentConfig** object:

```
apiVersion: template.openshift.io/v1
kind: Template
labels:
  template: myapp-persistent-template
metadata:
  name: myapp-persistent
  namespace: openshift
objects:
- apiVersion: v1
  kind: PersistentVolumeClaim ①
  metadata:
    name: ${APP_NAME}
  spec:
    accessModes:
    - ReadWriteOnce
    resources:
      requests:
        storage: ${VOLUME_CAPACITY}
- apiVersion: v1
  kind: DeploymentConfig ②
  metadata:
    name: ${APP_NAME}
  spec:
    replicas: 1
    selector:
      name: ${APP_NAME}
    strategy:
      type: Recreate
    template:
      metadata:
        labels:
```

```

        name: ${APP_NAME}
spec:
  containers:
    - image: 'openshift/myapp'
      name: myapp
      volumeMounts:
        - mountPath: /var/lib/myapp/data
          name: ${APP_NAME}-data 3
  volumes:
    - name: ${APP_NAME}-data 4
      persistentVolumeClaim:
        claimName: ${APP_NAME}
  parameters:
    - description: The name for the myapp application.
      displayName: Application Name
      name: APP_NAME 5
      required: true
      value: myapp
    - description: Volume space available for data, e.g. 512Mi, 2Gi.
      displayName: Volume Capacity
      name: VOLUME_CAPACITY 6
      required: true
      value: 1Gi

```

1 2 The template defines a **PersistentVolumeClaim** and **DeploymentConfig** object. Both objects have names matching the value of the APP_NAME parameter. The persistent volume claim defines a capacity corresponding the value of the VOLUME_CAPACITY parameter.

3 4 The **DeploymentConfig** object defines a volume mount referencing the **PersistentVolumeClaim** created by the template.

5 6 The template defines two parameters: APP_NAME and VOLUME_CAPACITY. The template uses these parameters to specify the value of attributes for the **PersistentVolumeClaim** and **DeploymentConfig** objects.

With this template, you only need to specify the APP_NAME and VOLUME_CAPACITY parameters to deploy the **myapp** application with persistent storage:

```
[student@workstation ~]$ oc create myapp-template.yaml
template.template.openshift.io/myapp-persistent created
[student@workstation ~]$ oc process myapp-persistent \
> -p APP_NAME=myapp-dev -p VOLUME_CAPACITY=1Gi \
> | oc create -f -
deploymentconfig/myapp created
persistentvolumeclaim/myapp created
```



REFERENCES

Developer information about templates can be found in the *Templates* section of the OpenShift Container Platform documentation:

Developer Guide

https://access.redhat.com/documentation/en-us/openshift_container_platform/3.11/html-single/developer_guide/#dev-guide-templates

► GUIDED EXERCISE

CREATING AN APPLICATION WITH A TEMPLATE

In this exercise, you will deploy the To Do List application in OpenShift Container Platform using a template to define resources your application needs to run.

OUTCOMES

You should be able to build and deploy an application in OpenShift Container Platform using a provided JSON template.

BEFORE YOU BEGIN

You must have the To Do List application source code and lab files on workstation. To download the lab files and verify the status of the OpenShift cluster, run the following command in a new terminal window.

```
[student@workstation ~]$ lab multicontainer-openshift start  
[student@workstation ~]$ cd /home/student/D0180/labs/multicontainer-openshift
```

- ▶ 1. Use the **Dockerfile** in the **images/mysql** subdirectory to build the database container. Publish the container image to `registry.lab.example.com` with a tag of **do180/mysql-57-rhel7**.
 - 1.1. Build the MySQL database image.

```
[student@workstation multicontainer-openshift]$ cd images/mysql  
[student@workstation mysql]$ sudo podman build -t do180/mysql-57-rhel7 .  
STEP 1: FROM rhscl/mysql-57-rhel7  
Getting image source signatures  
Copying blob sha256:e373541...output omitted...  
 69.66 MB / 69.66 MB [=====] 6s  
Copying blob sha256:c5d2e94...output omitted...  
 1.20 KB / 1.20 KB [=====] 0s  
Copying blob sha256:b3949ae...output omitted...  
 62.03 MB / 62.03 MB [=====] 5s  
Writing manifest to image destination  
Storing signatures  
STEP 2: ADD root /  
ERRO[0001] HOSTNAME is not supported for OCI image format ...output omitted...  
--> b628b44bda6d901b4dbf560fc960898883e20f8ef8cc2bce7154c55b3d99a079  
STEP 3: COMMIT do180/mysql-57-rhel7
```

**NOTE**

The **ERRO** error message is a known issue with an early version of Podman. You can ignore this message.

- 1.2. To make the image available for OpenShift, tag it and push it up to the private registry. To do so, run the following commands in the terminal window.

```
[student@workstation mysql]$ sudo podman tag \
> do180/mysql-57-rhel7 registry.lab.example.com/do180/mysql-57-rhel7
[student@workstation mysql]$ sudo podman push \
> registry.lab.example.com/do180/mysql-57-rhel7
Getting image source signatures
Copying blob sha256:d4d4080...output omitted...
196.35 MB / 196.35 MB [=====] 45s
Copying blob sha256:5444fe2...output omitted...
10.00 KB / 10.00 KB [=====] 0s
Copying blob sha256:acb06e8...output omitted...
202.39 MB / 202.39 MB [=====] 37s
Copying blob sha256:ba00e3e...output omitted...
4.00 KB / 4.00 KB [=====] 0s
Copying config sha256:58678ce...output omitted...
3.04 KB / 3.04 KB [=====] 0s
Writing manifest to image destination
Copying config sha256:58678ce...output omitted...
0 B / 3.04 KB [-----] 0s
Writing manifest to image destination
Storing signatures
```

**NOTE**

The **config sha256:** value in the above output may differ from your output.

- ▶ 2. Build the base image for the To Do List application using the Node.js Dockerfile, located in the exercise subdirectory **images/nodejs**. Tag the image as **do180/nodejs**. Do not publish this image to the registry.

```
[student@workstation mysql]$ cd ~/D0180/labs/multicontainer-openshift
[student@workstation multicontainer-openshift]$ cd images/nodejs
[student@workstation nodejs]$ sudo podman build -t do180/nodejs .
STEP 1: FROM rhel7:7.5
Getting image source signatures
Copying blob sha256:d02c3bd...output omitted...
71.46 MB / 71.46 MB [=====] 6s
Copying blob sha256:475b016...output omitted...
1.27 KB / 1.27 KB [=====] 0s
Copying config sha256:e64297b...output omitted...
6.51 KB / 6.51 KB [=====] 0s
Writing manifest to image destination
Storing signatures
STEP 2: MAINTAINER username <username@example.com>
```

```
...output omitted...
STEP 20: CMD ["echo", "You must create your own container from this one."]
--> 97f0eb3...output omitted...
STEP 21: COMMIT do180/nodejs
```

- 3. Use the **build.sh** script in the **deploy/nodejs** subdirectory to build the To Do List application. Publish the application image to the registry with an image tag of **do180/todonodejs**.

- 3.1. Go to the **~/D0180/labs/multicontainer-openshift/deploy/nodejs** directory and run the **build.sh** command to build the child image.

```
[student@workstation nodejs]$ cd ~/D0180/labs/multicontainer-openshift
[student@workstation multicontainer-openshift]$ cd deploy/nodejs
[student@workstation nodejs]$ ./build.sh
STEP 1: FROM do180/nodejs
STEP 2: MAINTAINER username <username@example.com>
...output omitted...
STEP 10: CMD ["scl","enable","rh-nodejs4","./run.sh"]
--> f627d64...output omitted...
STEP 11: COMMIT do180/todonodejs
```

- 3.2. Push the image to the private registry.

In order to make the image available for OpenShift to use in the template, tag it and push it to the private registry. To do so, run the following commands in the terminal window.

```
[student@workstation nodejs]$ sudo podman tag do180/todonodejs \
> registry.lab.example.com/do180/todonodejs
[student@workstation nodejs]$ sudo podman push \
> registry.lab.example.com/do180/todonodejs
Getting image source signatures
Copying blob sha256:24a5c62...output omitted...
...output omitted...
Copying blob sha256:5f70bf1...output omitted...
1024 B / 1024 B [=====] 0s
Copying config sha256:c43306d...output omitted...
7.26 KB / 7.26 KB [=====] 0s
Writing manifest to image destination
Copying config sha256:c43306d...output omitted...
0 B / 7.26 KB [-----] 0s
Writing manifest to image destination
Storing signatures
```

- 4. Create the persistent volume.

- 4.1. Log in to OpenShift Container Platform and create a persistent volume.

Similar to the volumes used with Podman containers, OpenShift uses the concept of persistent volumes to provide persistent storage for pods.

```
[student@workstation nodejs]$ oc login -u kubeadmin -p $(cat ~/.kubeadmin) \
> --insecure-skip-tls-verify https://cluster-api.lab.example.com:6443
```

```
Login successful.  
...output omitted...  
Using project "default".
```

If the **oc login** command prompts about using insecure connections, answer **y** (yes).

- 4.2. A script is provided for you to create the persistent volumes needed for this exercise. Run the following commands in the terminal window to create the persistent volume.

```
[student@workstation nodejs]$ cd ~/D0180/labs/multicontainer-openshift  
[student@workstation multicontainer-openshift]$ ./create-pv.sh  
db.sql 100% 300 129.9KB/s 00:00  
db.sql 100% 300 145.4KB/s 00:00  
persistentvolume/pv0001 created  
persistentvolume/pv0002 created
```



NOTE

If you have an issue with one of the later steps in the lab, you can delete the **template** project using the **oc delete project** to remove any existing resources and restart the exercise from this step.

5. Create the To Do List application from the provided JSON template.
 - 5.1. Create a new project **template** in OpenShift to use for this exercise. Run the following command to create the **template** project.

```
[student@workstation multicontainer-openshift]$ oc new-project template  
Now using project "template" on server "https://cluster-api.lab.example.com:6443".  
  
You can add applications to this project with the 'new-app' command. For example,  
try:  
  
oc new-app centos/ruby-25-centos7~https://github.com/sclorg/ruby-ex.git  
  
to build a new example application in Ruby.
```

- 5.2. Set the security policy for the project.

To allow the container to run with the adequate privileges, set the security policy using the provided script. Run the following command to set the policy.

```
[student@workstation multicontainer-openshift]$ ./setpolicy.sh  
securitycontextconstraints.security.openshift.io/anyuid added to:  
["system:serviceaccount:template:default"]
```

- 5.3. Review the template.

Using your preferred editor, open and examine the template located at **/home/student/D0180/labs/multicontainer-openshift/todo-**

template.json. Notice the following resources defined in the template and review their configurations.

- The `todoapi` pod definition defines the Node.js application.
 - The `mysql` pod definition defines the MySQL database.
 - The `todoapi` service provides connectivity to the Node.js application pod.
 - The `mysql` service provides connectivity to the MySQL database pod.
 - The **dbinit** persistent volume claim definition defines the MySQL `/var/lib/mysql/init` volume.
 - The **db-volume** persistent volume claim definition defines the MySQL `/var/lib/mysql/data` volume.
- 5.4. Process the template and create the application resources.

Use the `oc process` command to process the template file and then use the `pipe` command to send the result to the `oc create` command. This creates an application from the template.

Run the following command in the terminal window:

```
[student@workstation multicontainer-openshift]$ oc process \  
> -f todo-template.json | oc create -f -  
pod "mysql" created  
pod "todoapi" created  
service "todoapi" created  
service "mysql" created  
persistentvolumeclaim "dbinit" created  
persistentvolumeclaim "dbclaim" created
```

- 5.5. Review the deployment.

Review the status of the deployment using the `oc get pods` command with the `-w` option to continue to monitor the pod status. Wait until both the containers are running. It may take some time for both pods to start.

```
[student@workstation multicontainer-openshift]$ oc get pods -w  
NAME      READY     STATUS            RESTARTS   AGE  
mysql     0/1      ContainerCreating   0          9s  
todoapi   1/1      Running           0          9s  
mysql     1/1      Running           0          2m
```

Press **Ctrl+C** to exit the command.

► 6. Expose the Service.

To allow the To Do List application to be accessible through the OpenShift router and to be available as a public FQDN, use the `oc expose` command to expose the `todoapi` service.

Run the following command in the terminal window.

```
[student@workstation multicontainer-openshift]$ oc expose service todoapi  
route.route.openshift.io/todoapi exposed
```

► 7. Test the application.

- 7.1. Find the FQDN of the application by running the **oc status** command and note the FQDN for the app.

Run the following command in the terminal window.

```
[student@workstation multicontainer-openshift]$ oc status
In project template on server https://cluster-api.lab.example.com:6443

svc/mysql - 172.30.234.179:3306
pod/mysql runs registry.lab.example.com/do180/mysql-57-rhel7

http://todoapi-template.apps.cluster.lab.example.com to pod port ...
pod/todoapi runs registry.lab.example.com/do180/todonodejs

2 infos identified, use 'oc status --suggest' to see details.
```

- 7.2. Use **curl** to test the REST API for the To Do List application.

```
[student@workstation multicontainer-openshift]$ curl -w "\n" \
http://todoapi-template.apps.cluster.lab.example.com/todo/api/items/1
```

```
{"id":1,"description":"Pick up newspaper","done":false}
```

The **-w "\n"** option with **curl** command lets the shell prompt appear at the next line rather than merging with the output in the same line.

- 7.3. Open Firefox on **workstation** and point your browser to <http://todoapi-template.apps.cluster.lab.example.com/todo/> and you should see the To Do List application.



NOTE

The trailing slash in the URL mentioned above is necessary. If you do not include that in the URL, you may encounter issues with the application.

The screenshot shows a web browser window titled "To Do List Application". The address bar shows the URL <http://todoapi-template.apps.cluster.lab.example.com/todo/>. The main content area has two sections: "To Do List" and "Add Task".

To Do List:

ID	Description	Done
1	Pick up newspaper	false
2	Buy groceries	true

Add Task:

Description: Add Description.

Completed:

Buttons: Clear, Save

Page navigation: First, Previous, 1, Next, Last

Figure 7.4: To Do List application

Finish

On **workstation**, run the **lab multicontainer-openshift finish** script to complete this lab.

```
[student@workstation ~]$ lab multicontainer-openshift finish
```

This concludes the guided exercise.

► LAB

DEPLOYING MULTI-CONTAINER APPLICATIONS

PERFORMANCE CHECKLIST

In this lab, you will deploy a PHP Application with a MySQL database using an OpenShift template to define the resources needed by the application.

OUTCOMES

You should be able to create an OpenShift application comprised of multiple containers and access it through a web browser.

BEFORE YOU BEGIN

Open a terminal on workstation as the student user and run the following commands:

```
[student@workstation ~]$ lab multicontainer-review start  
[student@workstation ~]$ cd ~/D0180/labs/multicontainer-review
```

1. Log in to OpenShift cluster as the `kubadmin` user and create a new project for this exercise.
2. Relax the default cluster security policy.
3. Build the Database container image located in the `images/mysql` directory and publish it to the private registry.
4. Build the PHP container image located in the `images/quote-php` and publish it to the private registry.
5. Go to the `/home/student/D0180/labs/multicontainer-review/` directory and review the provided template `quote-php-template.json` file.
Note the definitions and configuration of the pods, services, and persistent volume claims defined in the template.
6. Create the persistent volumes needed for the application using the provided `create-pv.sh` script.



NOTE

If you have an issue with one of the later steps in the lab, you can delete the `deploy` project using the `oc delete project` command to remove any existing resources and restart the exercise from this step.

7. Upload the PHP application template so that any developer with access to your project can use it.
8. Process the uploaded template and create the application resources.
9. Expose the service.

10. Test the application and verify that it outputs an inspiring message.

Evaluation

Grade your work by running the **lab multicontainer-review grade** command from your workstation machine. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab multicontainer-review grade
```

Finish

To complete this lab, run the **lab multicontainer-review finish** command on workstation.

```
[student@workstation ~]$ lab multicontainer-review finish
```

This concludes the lab.

► SOLUTION

DEPLOYING MULTI-CONTAINER APPLICATIONS

PERFORMANCE CHECKLIST

In this lab, you will deploy a PHP Application with a MySQL database using an OpenShift template to define the resources needed by the application.

OUTCOMES

You should be able to create an OpenShift application comprised of multiple containers and access it through a web browser.

BEFORE YOU BEGIN

Open a terminal on workstation as the student user and run the following commands:

```
[student@workstation ~]$ lab multicontainer-review start
[student@workstation ~]$ cd ~/D0180/labs/multicontainer-review
```

1. Log in to OpenShift cluster as the `kubadmin` user and create a new project for this exercise.
 - 1.1. From `workstation`, log in as the `kubeadmin` user.

```
[student@workstation multicontainer-review]$ oc login -u kubeadmin \
> -p $(cat /home/student/.kubeadmin) https://cluster-api.lab.example.com:6443
Login successful.
```

You have access to the following projects and can switch between them with 'oc project <projectname>':

```
* default
...output omitted...
```

Using project "default".

If the `oc login` command prompts about using insecure connections, answer **y** (yes).

- 1.2. Create a new project in OpenShift named `deploy`:

```
[student@workstation multicontainer-review]$ oc new-project deploy
Now using project "deploy" on server "https://cluster-api.lab.example.com:6443".
```

You can add applications to this project with the 'new-app' command. For example, try:

```
oc new-app centos/ruby-25-centos7-https://github.com/sclorg/ruby-ex.git
```

to build a new example application in Ruby.

2. Relax the default cluster security policy.

- 2.1. To allow the container to run with the appropriate privileges, set the security policy using the provided **setpolicy.sh** shell script.

```
[student@workstation multicontainer-review]$ ./setpolicy.sh  
securitycontextconstraints.security.openshift.io/anyuid added to: ["system:..."]
```

3. Build the Database container image located in the **images/mysql** directory and publish it to the private registry.

- 3.1. Build the MySQL Database image using the provided Dockerfile in the **images/mysql** directory.

```
[student@workstation multicontainer-review]$ cd images/mysql  
[student@workstation mysql]$ sudo podman build -t do180/mysql-57-rhel7 .  
STEP 1: FROM rhscl/mysql-57-rhel7  
...output omitted...  
STEP 5: COMMIT do180/mysql-57-rhel7
```

- 3.2. Push the MySQL image to the private registry.

In order to make the image available for OpenShift to use in the template, give it the tag **registry.lab.example.com/do180/mysql-57-rhel7** and push it to the private registry.

To tag and push the image run the following commands in the terminal window.

```
[student@workstation mysql]$ sudo podman tag do180/mysql-57-rhel7 \  
> registry.lab.example.com/do180/mysql-57-rhel7  
[student@workstation mysql]$ sudo podman push \  
> registry.lab.example.com/do180/mysql-57-rhel7  
Getting image source signatures  
...output omitted...  
Writing manifest to image destination  
Storing signatures
```

Return to the previous directory.

```
[student@workstation mysql]$ cd ~/DO180/labs/multicontainer-review
```

4. Build the PHP container image located in the **images/quote-php** and publish it to the private registry.

- 4.1. Build the PHP image using the provided Dockerfile in the **images/quote-php** directory.

```
[student@workstation multicontainer-review]$ cd images/quote-php  
[student@workstation quote-php]$ sudo podman build -t do180/quote-php .  
STEP 1: FROM rhel7:7.5  
...output omitted...
```

STEP 17: COMMIT do180/quote-php

- 4.2. Tag and push the PHP image to the private registry.

In order to make the image available for OpenShift to use in the template, give it a tag of **registry.lab.example.com/do180/quote-php** and push it to the private registry.

```
[student@workstation quote-php]$ sudo podman tag do180/quote-php \
> registry.lab.example.com/do180/quote-php
[student@workstation quote-php]$ sudo podman push \
> registry.lab.example.com/do180/quote-php
Getting image source signatures
...output omitted...
Writing manifest to image destination
Storing signatures
```

5. Go to the **/home/student/D0180/labs/multicontainer-review/** directory and review the provided template **quote-php-template.json** file.

Note the definitions and configuration of the pods, services, and persistent volume claims defined in the template.

```
[student@workstation quote-php]$ cd ~/D0180/labs/multicontainer-review
```

6. Create the persistent volumes needed for the application using the provided **create-pv.sh** script.

Run the following commands in the terminal window to create the persistent volume.

```
[student@workstation multicontainer-review]$ ./create-pv.sh
```

**NOTE**

If you have an issue with one of the later steps in the lab, you can delete the **deploy** project using the **oc delete project** command to remove any existing resources and restart the exercise from this step.

7. Upload the PHP application template so that any developer with access to your project can use it.

Use the **oc create -f** command to upload the template file to the project.

```
[student@workstation multicontainer-review]$ oc create -f quote-php-template.json
template.template.openshift.io/quote-php-persistent created
```

8. Process the uploaded template and create the application resources.

- 8.1. Use the **oc process** command to process the template file. Use the **pipe** command to send the result to the **oc create** command to create an application from the template.

```
[student@workstation multicontainer-review]$ oc process quote-php-persistent \
> | oc create -f -
pod/mysql created
```

```
pod/quote-php created
service/quote-php created
service/mysql created
persistentvolumeclaim/dbinit created
persistentvolumeclaim/dbclaim created
```

- 8.2. Verify the status of the deployment using the **oc get pods** command with the **-w** option to monitor the deployment status. Wait until both pods are running. It may take some time for both pods to start up.

```
[student@workstation multicontainer-review]$ oc get pods -w
NAME      READY   STATUS        RESTARTS   AGE
mysql     0/1     ContainerCreating   0          21s
quote-php 0/1     ContainerCreating   0          20s
quote-php 1/1     Running        0          35s
mysql     1/1     Running        0          49s
^C
```

Press **Ctrl+C** to exit the command.

9. Expose the service.

To allow the PHP Quote application to be accessible through the OpenShift router and reachable from an external network, use the **oc expose** command to expose the quote-php service.

Run the following command in the terminal window.

```
[student@workstation multicontainer-review]$ oc expose svc quote-php
route.route.openshift.io/quote-php exposed
```

10. Test the application and verify that it outputs an inspiring message.

- 10.1. Use the **oc get route** command to find the FQDN where the application is available. Note the FQDN for the app.

Run the following command in the terminal window.

```
[student@workstation multicontainer-review]$ oc get route
NAME      HOST/PORT                               PATH  SERVICES  PORT ...
quote-php quote-php-deploy.apps.cluster.lab.example.com  quote-php  8080 ...
```

- 10.2. Use the **curl** command to test the REST API for the PHP Quote application.

```
[student@workstation ~]$ curl http://quote-php-deploy.apps.cluster.lab.example.com
Always remember that you are absolutely unique. Just like everyone else.
```



NOTE

The text displayed in the output above may differ, but the **curl** command should run successfully.

Evaluation

Grade your work by running the **lab multicontainer-review grade** command from your workstation machine. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab multicontainer-review grade
```

Finish

To complete this lab, run the **lab multicontainer-review finish** command on workstation.

```
[student@workstation ~]$ lab multicontainer-review finish
```

This concludes the lab.

SUMMARY

In this chapter, you learned:

- Software defined networks enable communication between containers. Containers must be attached to the same software-defined network to communicate.
- Containerized applications cannot rely on fixed IP addresses or host names to find services.
- Podman uses Container Network Interface (CNI) to create a software-defined network and attaches all containers on the host to that network. Kubernetes and OpenShift create a software-defined network between all containers in a pod.
- Within the same project, Kubernetes injects a set of variables for each service into all pods.
- Kubernetes templates automate creating applications consisting of multiple resources. Template parameters allow using the same values when creating multiple resources.

CHAPTER 8

TROUBLESHOOTING CONTAINERIZED APPLICATIONS

GOAL

Troubleshoot a containerized application deployed on OpenShift.

OBJECTIVES

- Troubleshoot an application build and deployment on OpenShift.
- Implement techniques for troubleshooting and debugging containerized applications.

SECTIONS

- Troubleshooting S2I Builds and Deployments (and Guided Exercise)
- Troubleshooting Containerized Applications (and Guided Exercise)

LAB

- Troubleshooting Containerized Applications

TROUBLESHOOTING S2I BUILDS AND DEPLOYMENTS

OBJECTIVES

After completing this section, you should be able to:

- Troubleshoot an application build and deployment steps on OpenShift.
- Analyze OpenShift logs to identify problems during the build and deploy process.

INTRODUCTION TO THE S2I PROCESS

The *Source-to-Image (S2I)* process is a simple way to automatically create images based on the programming language of the application source code in OpenShift. While this process is often a convenient way to quickly deploy applications, problems can arise during the S2I image creation process, either by the programming language characteristics or the runtime environment that require both developers and administrators to work together.

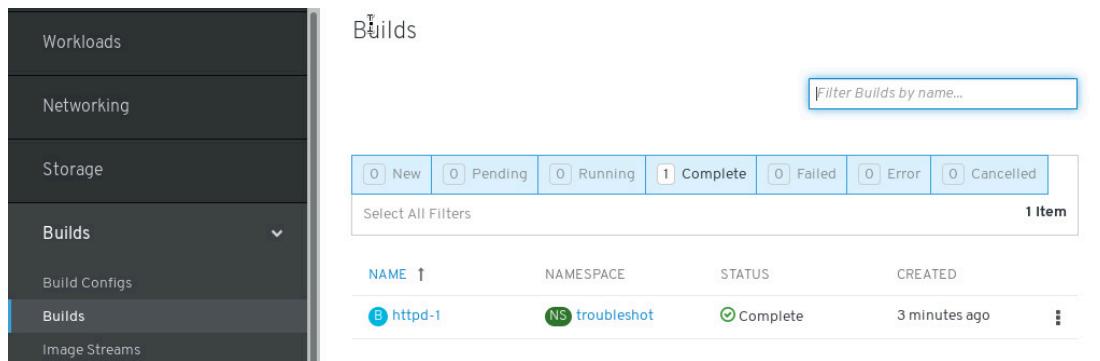
It is important to understand the basic workflow for most of the programming languages supported by OpenShift. The S2I image creation process is composed of two major steps:

- Build step: Responsible for compiling source code, downloading library dependencies, and packaging the application as a container image. Furthermore, the build step pushes the image to the OpenShift registry for the deployment step. The **BuildConfig (BC)** OpenShift resources drive the build step.
- Deployment step: Responsible for starting a pod and making the application available for OpenShift. This step executes after the build step, but only if the build step succeeded. The **DeploymentConfig (DC)** OpenShift resources drive the deployment step.

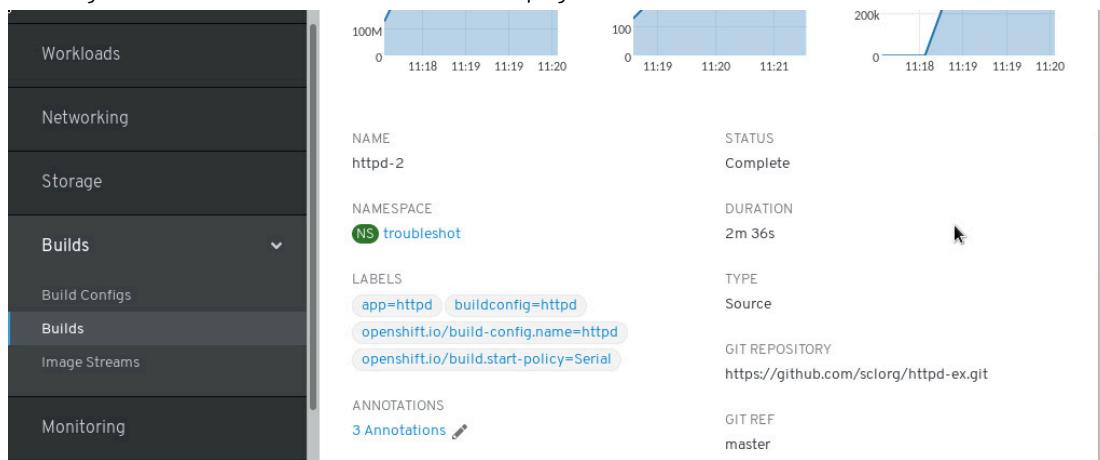
For the S2I process, each application uses its own **BuildConfig** and **DeploymentConfig** objects, the name of which matches the application name. The deployment process aborts if the build fails.

The S2I process starts each step in a separate pod. The build process creates a pod named **<application-name>-build-<number>-<string>**. For each build attempt, the entire build step executes and saves a log. Upon a successful build, the application starts on a separate pod named as **<application-name>-<string>**.

The OpenShift web console can be used to access the details for each step. To identify any build issues, the logs for a build can be evaluated and analyzed by clicking the **Builds** link from the left panel, depicted as follows.

**Figure 8.1: Build instances of a project**

For each build attempt, a history of the build, tagged with a number, is provided for evaluation. Clicking on the build name leads to the details page of the build.

**Figure 8.2: Detailed view of a build instance**

The Logs tab of the build details page shows the output generated by the build execution. Those logs are handy to identify build issues.

Use the Deployment Configs link under Workloads section from the left panel to identify issues during the deployment step.

After selecting the appropriate deployment configuration, details show in the Overview section.

The **oc** command-line interface has several subcommands for managing the logs. Likewise in the web interface, it has a set of commands which provides information about each step. For example, to retrieve the logs from a build configuration, run the following command.

```
$ oc logs bc/<application-name>
```

If a build fails, after finding and fixing the issues, run the following command to request a new build:

```
$ oc start-build <application-name>
```

By issuing that command, OpenShift automatically spawns a new pod with the build process.

Deployment logs can be checked with the **oc** command:

```
$ oc logs dc/<application-name>
```

If the deployment is running or has failed, the command returns the logs of the process deployment process. Otherwise, the command returns the logs from the application's pod.

DESCRIBING COMMON PROBLEMS

Sometimes, the source code requires some customization that may not be available in containerized environments, such as database credentials, file system access, or message queue information. Those values usually take the form of internal environment variables. Developers using the S2I process may need to access this information.

The **oc logs** command provides important information about the build, deploy, and run processes of an application during the execution of a pod. The logs may indicate missing values or options that must be enabled, incorrect parameters or flags, or environment incompatibilities.



NOTE

Application logs must be clearly labelled to identify problems quickly without the need to learn the container internals.

Troubleshooting Permission Issues

OpenShift runs S2I containers using Red Hat Enterprise Linux as the base image, and any runtime difference may cause the S2I process to fail. Sometimes, the developer runs into permission issues, such as access denied due to the wrong permissions, or incorrect environment permissions set by administrators. S2I images enforce the use of a different user than the `root` user to access file systems and external resources. Also, Red Hat Enterprise Linux 7 enforces SELinux policies that restrict access to some file system resources, network ports, or process.

Some containers may require a specific user ID, whereas S2I is designed to run containers using a random user as per the default OpenShift security policy.

The following Dockerfile creates a Nexus container. Note the **USER** instruction indicating the `nexus` user should be used:

```
FROM rhel7:7.5
...contents omitted...
RUN chown -R nexus:nexus ${NEXUS_HOME}

USER nexus
WORKDIR ${NEXUS_HOME}

VOLUME ["/opt/nexus/sonatype-work"]
...contents omitted...
```

Trying to use the image generated by this Dockerfile without addressing volume permissions drives to errors when the container starts:

```
$ oc logs nexus-1-wzjrn
...output omitted...
... org.sonatype.nexus.util.LockFile - Failed to write lock file
...FileNotFoundException: /opt/nexus/sonatype-work/nexus.lock (Permission denied)
```

```
...output omitted...
... org.sonatype.nexus.webapp.WebappBootstrap - Failed to initialize
...lStateException: Nexus work directory already in use: /opt/nexus/sonatype-work
...output omitted...
```

To solve this issue, relax the OpenShift project security with the command **oc adm policy**. As an example, the **setpolicy.sh** script used on some of the previous labs allows the user defined in the **Dockerfile** file to run the application.

The **setpolicy.sh** script enables OpenShift executing container processes with non-root users. But the file systems used in the container must also be available for the running user. This is specially important when the container contains volume mounts.

To avoid file system permission issues, local folders used for container volume mounts must satisfy the following:

- The user executing the container processes must be the owner of the folder, or have the necessary rights. Use the **chown** command to update folder ownership.
- The local folder must satisfy the SELinux requirements to be used as a container volume. Assign the **container_file_t** group to the folder by using the **semanage fcontext -a -t container_file_t <folder>** command, then refresh the permissions with the **restorecon -R <folder>** command.

Troubleshooting Invalid Parameters

Multi-container applications may share parameters, such as login credentials. Ensure that the same values for parameters reach all containers in the application. For example, for a Python application that runs in one container, connected with another container running a database, make sure that the two containers use the same user name and password for the database. Usually, logs from the application pod provide a clear idea of these problems and how to solve them.

A good practice to centralize shared parameters is to store them in **ConfigMaps**. Those **ConfigMaps** can be injected through the **Deployment Config** into containers as environment variables. Injecting the same **ConfigMap** into different containers ensures that not only the same environment variables are available, but also the same values. See the following pod resource definition:

```
apiVersion: v1
kind: Pod
...output omitted...
spec:
  containers:
    - name: test-container
  ...output omitted...
  env:
    - name: ENV_1 ①
      valueFrom:
        configMapKeyRef:
          name: configMap_name1
          key: configMap_key_1
  ...output omitted...
  envFrom:
    - configMapRef:
        name: configMap_name_2 ②
```

```
...output omitted...
```

- ① An ENV_1 environment variable is injected into the container. Its value is the value for the configMap_key_1 entry in the **configMap_name1** configMap.
- ② All entries in **configMap_name_2** are injected into the container as environment variables with the same name and values.

Troubleshooting Volume Mount Errors

When redeploying an application that uses a persistent volume on a local file system, a pod might not be able to allocate a persistent volume claim even though the persistent volume indicates that the claim is released. To resolve the issue, delete the persistent volume claim and then the persistent volume. Then recreate the persistent volume.

```
oc delete pv <pv_name>
oc create -f <pv_resource_file>
```

Troubleshooting Obsolete Images

OpenShift pulls images from the source indicated in an image stream unless it locates a locally-cached image on the node where the pod is scheduled to run. If you push a new image to the registry with the same name and tag, you must remove the image from each node the pod is scheduled on with the command **podman rmi**.

Run the **oc adm prune** command for an automated way to remove obsolete images and other resources.



REFERENCES

More information about troubleshooting images is available in the *Images* section of the OpenShift Container Platform documentation accessible at:

Creating Images

https://docs.openshift.com/container-platform/3.11/creating_images/guidelines.html

Documentation about how to consume ConfigMap to create container environment variables can be found in the *Consuming in Environment Variables* of the

OKD documentation site

https://docs.okd.io/latest/dev_guide/configmaps.html#configmaps-use-case-consuming-in-env-vars

► GUIDED EXERCISE

TROUBLESHOOTING AN OPENSHIFT BUILD

In this exercise, you will troubleshoot an OpenShift build and deployment process.

OUTCOMES

You should be able to identify and solve the problems raised during the build and deployment process of a Node.js application.

BEFORE YOU BEGIN

The OpenShift cluster must be running and accessible at `https://cluster-api.lab.example.com:6443`.

Retrieve the lab files and verify that Docker and the OpenShift cluster are running by running the following command.

```
[student@workstation ~]$ lab troubleshoot-s2i start
```

- 1. Log in to OpenShift with the `kubeadmin` username. The password is stored in the `~/.kubeadmin` file.

```
[student@workstation openshift-s2i]$ oc login -u kubeadmin \
> -p $(cat ~/.kubeadmin) \
> https://cluster-api.lab.example.com:6443
Login successful.
```

```
You have access to the following projects and can switch between them with 'oc
project <projectname>':
...output omitted...
```



NOTE

If the `oc login` command prompts about using insecure connections, answer `y` (yes). To skip this check, you can also add the `--insecure-skip-tls-verify` option to the login command.

Create a new project named `nodejs`.

```
[student@workstation ~]$ oc new-project nodejs
Now using project "nodejs" on server "https://cluster-api.lab.example.com:6443".
...output omitted...
```

- 2. Build a new Node.js application using the **Hello World** image located at <https://registry.lab.example.com/nodejs-helloworld>.
- 2.1. Run the **oc new-app** command to create the Node.js application. The command is provided in the **~/D0180/labs/troubleshoot-s2i/command.txt** file.

```
[student@workstation ~]$ oc new-app --build-env npm_config_registry=\
> http://services.lab.example.com:8081/nexus/content/groups/nodejs/ \
> nodejs:4~http://services.lab.example.com/nodejs-helloworld
--> Found image a2b5ec2 ...output omitted...

Node.js 4
...output omitted...
Run 'oc status' to view your app.
```

The **--build-env** option defines an environment variable to the builder pod. In this case, it provides the **npm_config_registry** environment variable to the builder pod, so it can reach the NPM registry.



IMPORTANT

In the previous command, there must be no spaces between **registry=** and the URL of the Nexus server. There also must be no spaces before **http:**, but there is a space before **nodejs:4**

- 2.2. Wait until the application finishes building by monitoring the progress with the **oc get pods -w** command. The pod transitions from a status of **running** to **Error**:

```
[student@workstation ~]$ oc get pods -w
NAME                  READY   STATUS    RESTARTS   AGE
nodejs-helloworld-1-build   1/1     Running   0          15s
nodejs-helloworld-1-build   0/1     Error     0          73s
^C
```

The build process fails, and therefore no application is running. Build failures are usually consequences of syntax errors in the source code or missing dependencies. The next step investigates the specific causes for this failure.

- 2.3. Evaluate the errors raised during the build process.

The build is triggered by the build configuration (**bc**) created by OpenShift when the S2I process starts. By default, the OpenShift S2I process creates a build configuration named **nodejs-helloworld**, which is responsible for triggering the build process.

Run the **oc** command with the **logs** subcommand in a terminal window to review the output of the build process:

```
[student@workstation nodejs-helloworld]$ oc logs bc/nodejs-helloworld
Cloning "http://services.lab.example.com/nodejs-helloworld" ...
Commit: 03956fd606dc034a85462fc390582737e1dcc47 (Establish remote repository)
Author: root <root@services.lab.example.com>
Date: Wed Jun 13 00:38:02 2018 +0000
...output omitted...
```

```
STEP 8: RUN /usr/libexec/s2i/assemble
---> Installing application source ...
---> Installing all dependencies
npm WARN package.json nodejs-helloworld@1.0.0 No repository field.
npm WARN package.json nodejs-helloworld@1.0.0 No README data
npm WARN package.json nodejs-helloworld@1.0.0 license should be a valid SPDX
  license expression
npm ERR! Linux 3.10.0-862.el7.x86_64
npm ERR! argv "/opt/rh/rh-nodejs4/root/usr/bin/node" "/opt/rh/rh-nodejs4/root/usr/
bin/npm" "install"
npm ERR! node v4.6.2
npm ERR! npm  v2.15.1
npm ERR! code ETARGET

npm ERR! notarget No compatible version found: express@'>=4.14.2 <4.15.0'
...output omitted...
npm ERR!      /opt/app-root/src/npm-debug.log
subprocess exited with status 1
...output omitted...
```

The log shows an error occurred during the build process. This output indicates that there is no compatible version for the **express** dependency. But the reason is that the format used by the express dependency is not valid.

▶ 3. Update the build process for the project.

The developer uses a nonstandard version of the Express framework that is available locally on each developer's workstation. Due to the company's standards, the version must be downloaded from the Node.js official registry and, from the developer's input, it is compatible with the 4.14.x version.

3.1. Clone the Git repository.

Open a new terminal window from the **workstation** VM (Applications → Utilities → Terminal) and run the **git** command from the **~/D0180/labs/troubleshoot-s2i** directory.

```
[student@workstation ~]$ cd ~/D0180/labs/troubleshoot-s2i
[student@workstation troubleshoot-s2i]$ git clone \
> http://services.lab.example.com/nodejs-helloworld
```

The source code from the application is cloned locally in the **~/D0180/labs/troubleshoot-s2i/nodejs-helloworld** directory.

3.2. Fix the **package.json** file.

Use your preferred editor to open the **~/D0180/labs/troubleshoot-s2i/nodejs-helloworld/package.json** file. Review the dependencies versions provided by the developers. It uses an incorrect version of the Express dependency,

which is incompatible with the supported version provided by the company (~4.14.2). Update the dependency version as follows.

```
"express": "4.14.x"
```

**NOTE**

Notice the **x** in the version. It indicates that the highest version should be used, but the version must begin with **4.14..**

- 3.3. Commit and push the changes made to the project.

From the terminal window, run the following command to commit and push the changes:

```
[student@workstation troubleshoot-s2i]$ cd nodejs-helloworld  
[student@workstation nodejs-helloworld]$ git commit -am "Fixed Express release"  
...output omitted...  
1 file changed, 1 insertion(+), 1 deletion(-)  
[student@workstation nodejs-helloworld]$ git push  
...output omitted...  
To http://services.lab.example.com/nodejs-helloworld  
ef6557d..73a82cd master -> master
```

- 4. Relaunch the S2I process.

- 4.1. To restart the build step, execute the following command:

```
[student@workstation nodejs-helloworld]$ oc start-build bc/nodejs-helloworld  
build "nodejs-helloworld-2" started
```

The build step is restarted, and a new build pod is created. Check the log by running the **oc logs** command.

```
[student@workstation nodejs-helloworld]$ oc logs -f bc/nodejs-helloworld  
Cloning "https://registry.lab.example.com/nodejs-helloworld"  
...output omitted...  
Successfully pushed //image-registry...svc:5000/nodejs/nodejs-helloworld:latest@sha256:7134...3954  
Push successful
```

The build is successful, however, this does not indicate that the application is started.

- 4.2. Evaluate the status of the current build process. Run the **oc get pods** command to check the status of the Node.js application.

```
[student@workstation nodejs-helloworld]$ oc get pods
```

According to the following output, the second build completed, but the application is in error state.

NAME	READY	STATUS	RESTARTS	AGE
------	-------	--------	----------	-----

<code>nodejs-helloworld-1-build</code>	0/1	Error	0	29m
<code>nodejs-helloworld-1-rpx1d</code>	0/1	CrashLoopBackOff	6	6m
<code>nodejs-helloworld-2-build</code>	0/1	Completed	0	7m

The name of the application pod (`nodejs-helloworld-1-rpx1d`) is generated randomly, and may differ from yours.

- 4.3. Review the logs generated by the application pod.

```
[student@workstation nodejs-helloworld]$ oc logs dc/nodejs-helloworld
...output omitted...
npm ERR! Linux 3.10.0-957.1.3.el7.x86_64
npm ERR! argv "/opt/rh/rh-nodejs4/root/usr/bin/node" "/opt/rh/rh-nodejs4/root/usr/bin/npm" "run" "-d" "start"
npm ERR! node v4.6.2
npm ERR! npm  v2.15.1

npm ERR! missing script: start
npm ERR!
...output omitted...
```



NOTE

The `oc logs dc/nodejs-helloworld` command dumps the logs from the deployment pod. In the case of a successful deployment, that command dumps the logs from the application pod, as previously shown.

The application fails to start because the `start` script declaration is missing.

5. Fix the problem by updating the application code.

- 5.1. Update the `package.json` file to define a startup command.

The previous output indicates that the `~/DO180/labs/troubleshoot-s2i/nodejs-helloworld/package.json` file is missing the `start` attribute in the `scripts` field. The `start` attribute defines a command to run when the application starts. It invokes the `node` binary, which runs the `app.js` application.

To fix the problem, add to the `package.json` file the following attribute. Do not forget the comma after the bracket.

```
...
  "description": "Hello World!",
  "main": "app.js",
  "scripts": {
    "start": "node app.js"
  },
  "author": "Red Hat Training",
...

```

- 5.2. Commit and push the changes made to the project:

```
[student@workstation nodejs-helloworld]$ git commit -am "Added start up script"
```

```
...output omitted...
1 file changed, 3 insertions(+)
[student@workstation nodejs-helloworld]$ git push
...output omitted...
To http://services.lab.example.com/nodejs-helloworld
    73a82cd..a5a0411  master -> master
```

Continue the deploy step from the S2I process.

5.3. Restart the build step.

```
[student@workstation nodejs-helloworld]$ oc start-build bc/nodejs-helloworld
build "nodejs-helloworld-3" started
```

5.4. Evaluate the status of the current build process. Run the command to retrieve the status of the Node.js application. Wait for the latest build to finish.

```
[student@workstation nodejs-helloworld]$ oc get pods -w
```

NAME	READY	STATUS	RESTARTS	AGE
nodejs-helloworld-1-build	0/1	Error	0	66m
nodejs-helloworld-1-mtxsh	0/1	CrashLoopBackOff	9	23m
nodejs-helloworld-2-build	0/1	Completed	0	28m
nodejs-helloworld-3-build	1/1	Running	0	2m3s
nodejs-helloworld-2-deploy	0/1	Pending	0	0s
nodejs-helloworld-2-deploy	0/1	Pending	0	0s
nodejs-helloworld-2-deploy	0/1	ContainerCreating	0	0s
nodejs-helloworld-3-build 0/1 Completed 0 3m9s				
nodejs-helloworld-2-deploy	1/1	Running	0	4s
nodejs-helloworld-2-8ts14	0/1	Pending	0	0s
nodejs-helloworld-2-8ts14	0/1	Pending	0	1s
nodejs-helloworld-2-8ts14	0/1	ContainerCreating	0	1s
nodejs-helloworld-2-8ts14 1/1 Running 0 50s				
nodejs-helloworld-1-mtxsh	0/1	Terminating	9	25m
nodejs-helloworld-1-mtxsh	0/1	Terminating	9	25m
nodejs-helloworld-2-deploy 0/1 Completed 0 61s				
nodejs-helloworld-2-deploy	0/1	Terminating	0	61s
nodejs-helloworld-2-deploy	0/1	Terminating	0	61s
nodejs-helloworld-1-mtxsh	0/1	Terminating	9	25m
nodejs-helloworld-1-mtxsh	0/1	Terminating	9	25m

According to the output, the build is successful, and the application is able to start with no errors. The output also provides insight into how the deployment pod (**nodejs-helloworld-2-deploy**) was created, and that it completed successfully and terminated. As the new application pod is available (**nodejs-helloworld-2-8ts14**), the old one (**nodejs-helloworld-1-mtxsh**) is rolled out.

5.5. Review the logs generated by the **nodejs-helloworld** application pod.

```
[student@workstation nodejs-helloworld]$ oc logs dc/nodejs-helloworld
Environment:
```

```
DEV_MODE=false
NODE_ENV=production
DEBUG_PORT=5858
Launching via npm...
npm info it worked if it ends with ok
npm info using npm@2.15.1
npm info using node@v4.6.2
npm info prestart nodejs-helloworld@1.0.0
npm info start nodejs-helloworld@1.0.0

> nodejs-helloworld@1.0.0 start /opt/app-root/src
> node app.js

Example app listening on port 8080!
```

The application is now running on port 8080.

▶ 6. Test the application.

- 6.1. Run the **oc** command with the **expose** subcommand to expose the application:

```
[student@workstation nodejs-helloworld]$ oc expose svc/nodejs-helloworld
route.route.openshift.io/nodejs-helloworld exposed
```

- 6.2. Retrieve the address associated with the application.

```
[student@workstation nodejs-helloworld]$ oc get route -o yaml
apiVersion: v1
items:
- apiVersion: route.openshift.io/v1
  kind: Route
  ...output omitted...
  spec:
    host: nodejs-helloworld-nodejs.apps.cluster.lab.example.com
    port:
      targetPort: 8080-tcp
    to:
      kind: Service
      name: nodejs-helloworld
  ...output omitted...
```

- 6.3. Access the application from the workstation VM by using the **curl** command:

```
[student@workstation nodejs-helloworld]$ curl -w "\n" \
> http://nodejs-helloworld-nodejs.apps.cluster.lab.example.com
Hello world!
```

The output demonstrates the application is up and running.

Finish

On workstation, run the **lab troubleshoot-s2i finish** script to complete this exercise.

```
[student@workstation ~]$ lab troubleshoot-s2i finish
```

This concludes the exercise.

TROUBLESHOOTING CONTAINERIZED APPLICATIONS

OBJECTIVES

After completing this section, you should be able to:

- Implement techniques for troubleshooting and debugging containerized applications.
- Use the port-forwarding feature of the OpenShift client tool.
- View container logs.
- View OpenShift cluster events.

FORWARDING PORTS FOR TROUBLESHOOTING

Occasionally developers and system administrators need special network access to a container that would not be needed by application users. For example, developers may need to use the administration console for a database or messaging service, or system administrators may make use of SSH access to a container to restart a terminated service. Such network access, in the form of network ports, are usually not exposed by the default container configurations, and tend to require specialized clients used by developers and system administrators.

Podman provides port forwarding features by using the **-p** option along with the **run** subcommand. In this case, there is no distinction between network access for regular application access and for troubleshooting. As a refresher, the following is an example of configuring port forwarding by mapping the port from the host to a database server running inside a container:

```
$ sudo podman run --name db -p 30306:3306 mysql
```

The previous command maps the host port 30306 to the port 3306 on the **db** container. This container is created from the **mysql** image, which starts a MySQL server that listens on port 3306.

OpenShift provides the **oc port-forward** command for forwarding a local port to a pod port. This is different than having access to a pod through a service resource:

- The port-forwarding mapping exists only in the workstation where the **oc** client runs, while a service maps a port for all network users.
- A service load-balances connections to potentially multiple pods, whereas a port-forwarding mapping forwards connections to a single pod.

Here is an example of the **oc port-forward** command:

```
$ oc port-forward db 30306 3306
```

The previous command forwards port 30306 from the developer machine to port 3306 on the **db** pod, where a MySQL server (inside a container) accepts network connections.

NOTE

When running this command, be sure to leave the terminal window running. Closing the window or canceling the process stops the port mapping.

While the **podman run -p** method of mapping (port-forwarding) can only be configured when the container is started, the mapping with the **oc port-forward** command can be created and destroyed at any time after a pod was created.

NOTE

Creating a service of **NodePort** type for a database pod would be similar to running **podman run -p**. However, Red Hat discourages the usage of the **NodePort** approach to avoid exposing the service to direct connections. Mapping with port-forwarding in OpenShift is considered a more secure alternative.

ENABLING REMOTE DEBUGGING WITH PORT FORWARDING

Another use for the port forwarding feature is enabling remote debugging. Many *integrated development environments (IDEs)* provide the capability to remotely debug an application.

For example, JBoss Developer Studio (JBDS) allows users to utilize the Java Debug Wire Protocol (JDWP) to communicate between a debugger (JBDS) and the Java Virtual Machine. When enabled, developers can step through each line of code as it is being executed in real time.

For JDWP to work, the Java Virtual Machine (JVM) where the application runs must be started with options enabling remote debugging. For example, WildFly and JBoss EAP users must configure these options on application server startup. The following line in the **standalone.conf** file enables remote debugging by opening the JDWP TCP port 8787, for a WildFly or EAP instance running in standalone mode:

```
JAVA_OPTS="$JAVA_OPTS \
> -agentlib:jdwp=transport=dt_socket,address=8787,server=y,suspend=n"
```

When the server starts with the debugger listening on port 8787, a port forwarding mapping needs to be created to forward connections from a local unused TCP port to port 8787 in the EAP pod. If the developer workstation has no local JVM running with remote debugging enabled, the local port can also be 8787.

The following command assumes a WildFly pod named **jappserver** running a container from an image previously configured to enable remote debugging:

```
$ oc port-forward jappserver 8787:8787
```

Once the debugger is enabled and the port forwarding mapping is created, users can set breakpoints in their IDE of choice and run the debugger by pointing to the application's host name and debug port (in this instance, 8787).

ACCESSING CONTAINER LOGS

Podman and OpenShift provide the ability to view logs in running containers and pods to facilitate troubleshooting. But neither of them is aware of application specific logs. Both expect the application to be configured to send all logging output to the standard output.

A container is simply a process tree from the host OS perspective. When Podman starts a container either directly or on the RHOCP cluster, it redirects the container standard output and standard error, saving them on disk as part of the container's ephemeral storage. This way, the container logs can be viewed using **podman** and **oc** commands, even after the container was stopped, but not removed.

To retrieve the output of a running container, use the following **podman** command.

```
$ podman logs <containerName>
```

In OpenShift, the following command returns the output for a container within a pod:

```
$ oc logs <podName> [-c <containerName>]
```



NOTE

The container name is optional if there is only one container, as **oc** defaults to the only running container and returns the output.

OPENSFIFT EVENTS

Some developers consider Podman and OpenShift logs to be too low-level, making troubleshooting difficult. Fortunately, OpenShift provides a high-level logging and auditing facility called **events**.

OpenShift events signal significant actions like starting a container or destroying a pod.

To read OpenShift events, use the **get** subcommand with the **events** resource type for the **oc** command, as follows.

```
$ oc get events
```

Events listed by the **oc** command this way are not filtered and span the whole RHOCP cluster. Using a pipe to standard UNIX filters such as **grep** can help, but OpenShift offers an alternative in order to consult cluster events. The approach is provided by the **describe** subcommand.

For example, to only retrieve the events that relate to a **mysql** pod, refer Events field from the output of **oc describe pod mysql** command.

```
$ oc describe pod mysql
...output omitted...
Events:
FirstSeen    LastSeen    Count  From            Reason          Message
Wed, 10 ...  Wed, 10 ... 1      {scheduler} scheduled  Successfully as...
...output omitted...
```

ACCESSING RUNNING CONTAINERS

The **podman logs** and **oc logs** commands can be useful for viewing output sent by any container. However, the output does not necessarily display all of the available information if the application is configured to send logs to a file. Other troubleshooting scenarios may require inspecting the container environment as seen by processes inside the container, such as verifying external connectivity.

As a solution, Podman and OpenShift provide the **exec** subcommand, allowing the creation of new processes inside a running container, with the standard output and input of these processes redirected to the user terminal. The following screen display the usage of the **podman exec** command:

```
$ sudo podman exec [options] container command [arguments]
```

The general syntax for the **oc exec** command is:

```
$ oc exec [options] pod [-c container] -- command [arguments]
```

To execute a single interactive command or start a shell, add the **-it** options. The following example starts a Bash shell for the **myhttpd** pod:

```
$ oc exec -it myhttpd /bin/bash
```

You can use this command to access application logs saved to disk (as part of the container ephemeral storage). For example, to display the Apache error log from a container, run the following command:

```
$ sudo podman exec apache-container cat /var/log/httpd/error_log
```

OVERRIDING CONTAINER BINARIES

Many container images do not contain all of the troubleshooting commands users expect to find in regular OS installations, such as **telnet**, **netcat**, **ip**, or **traceroute**. Stripping the image from basic utilities or binaries allows the image to remain slim, thus, running many containers per host.

One way to temporarily access some of these missing commands is mounting the host binaries folders, such as **/bin**, **/sbin**, and **/lib**, as volumes inside the container. This is possible because the **-v** option from **podman run** command does not require matching **VOLUME** instructions to be present in the **Dockerfile** of the container image.



NOTE

To access these commands in OpenShift, you need to change the pod resource definition in order to define **volumeMounts** and **volumeClaims** objects. You also need to create a **hostPath** persistent volume.

The following command starts a container, and overrides the image's **/bin** folder with the one from the host. It also starts an interactive shell inside the container.

```
$ sudo podman run -it -v /bin:/bin image /bin/bash
```

**NOTE**

The directory of binaries to override depends on the base OS image. For example, some commands require shared libraries from the **/lib** directory. Some Linux distributions have different contents in **/bin**, **/usr/bin**, **/lib**, or **/usr/lib**, which would require to use the **-v** option for each directory.

As an alternative, you can include these utilities in the base image. To do so, add instructions in a **Dockerfile** build definition. For example, examine the following excerpt from a **Dockerfile** definition, which is a child of the **rhe17.5** image used throughout this course. The **RUN** instruction installs the tools that are commonly used for network troubleshooting.

```
FROM rhe17.5

RUN yum install -y \
    less \
    dig \
    ping \
    iputils && \
    yum clean all
```

When the image is built and the container is created, it will be identical to a **rhe17.5** container image, plus the extra available tools.

TRANSFERRING FILES TO AND OUT OF CONTAINERS

When troubleshooting or managing an application, you may need to retrieve or transfer files to and from running containers, such as configuration files or log files. There are several ways to move files into and out of containers, as described in the following list.

Volume mounts

Another option for copying files from the host to a container is the usage of volume mounts. You can mount a local directory to copy data into a container. For example, the following command sets **/conf** host directory as the volume to use for the Apache configuration directory in the container. This provides a convenient way to manage the Apache server without having to rebuild the container image.

```
$ sudo podman run -v /conf:/etc/httpd/conf -d do180/apache
```

podman cp

The **cp** subcommand allows users to copy files both into and out of a running container. To copy a file into a container named **todoapi**, run the following command.

```
$ sudo podman cp standalone.conf todoapi:/opt/jboss/standalone/conf/
standalone.conf
```

To copy a file from the container to the host, flip the order of the previous command.

```
$ sudo podman cp todoapi:/opt/jboss/standalone/conf/standalone.conf .
```

The **podman cp** command has the advantage of working with containers that were already started, while the following alternative (volume mounts) requires changes to the command used to start a container.

podman exec

For containers that are already running, the **podman exec** command can be piped to pass files both into and out of the running container by appending commands that are executed in the container. The following example shows how to pass in and execute a SQL file inside a MySQL container:

```
$ sudo podman exec -i <container> mysql -uroot -proot </path/on/host/db.sql < db.sql
```

Using the same concept, it is possible to retrieve data from a running container and place it in the host machine. A useful example of this is the usage of the **mysqldump** utility, which creates a backup of MySQL database from the container and places it on the host.

```
$ sudo podman exec -it <containerName> sh \
> -c 'exec mysqldump -h"$MYSQL_PORT_3306_TCP_ADDR" \
> -P"$MYSQL_PORT_3306_TCP_PORT" \
> -uroot -p"$MYSQL_ENV_MYSQL_ROOT_PASSWORD" items' \
> db_dump.sql
```

The previous command uses the container environment variables to connect to the MySQL server to execute the **mysqldump** command and redirects the output to a file on the host machine. It assumes that the container image provides the **mysqldump** utility, so there is no need to install the MySQL administration tools on the host.

The **oc rsync** command provides functionality similar to **podman cp** for containers running under OpenShift pods.



REFERENCES

More information about port-forwarding is available in the *Port Forwarding* section of the OpenShift Container Platform documentation at

Architecture

https://access.redhat.com/documentation/en-us/openshift_container_platform/3.11/html-single/architecture/

More information about the CLI commands for port-forwarding are available in the *Port Forwarding* chapter of the OpenShift Container Platform documentation at

Developer Guide

https://access.redhat.com/documentation/en-us/openshift_container_platform/3.11/html-single/developer_guide/

► GUIDED EXERCISE

CONFIGURING APACHE CONTAINER LOGS FOR DEBUGGING

In this exercise, you will configure an Apache httpd container to send the logs to the **stdout**, then review Podman logs and events.

OUTCOMES

You should be able to configure an Apache httpd container to send debug logs to **stdout** and view them using the **podman logs** command.

BEFORE YOU BEGIN

Run the setup script for this exercise. The script ensures that Docker is running on workstation and retrieves the material for this exercise.

```
[student@workstation ~]$ lab troubleshoot-container start
```

- 1. Configure the Apache web server to send log messages to the standard output and update the default log level.

- 1.1. The default log level for the Apache httpd image is **warn**. Change the default log level for the container to **debug**, and redirect log messages to **stdout** by overriding the default **httpd.conf** configuration file. To do so, create a custom image from the workstation VM.

Briefly review the custom **httpd.conf** file located at **/home/student/D0180/labs/troubleshoot-container/conf/httpd.conf**.

- Observe the **ErrorLog** directive in the file:

```
ErrorLog /dev/stdout
```

The directive sends the httpd error log messages to the container's standard output.

- Observe the **LogLevel** directive in the file.

```
LogLevel debug
```

The directive changes the default log level to **debug**.

- Observe the **CustomLog** directive in the file.

```
CustomLog /dev/stdout common
```

The directive redirects the httpd access log messages to the container's standard output.

► 2. Build a custom container to save an updated configuration file to the container.

2.1. From the terminal window, run the following commands to build a new image.

```
[student@workstation ~]$ cd ~/D0180/labs/troubleshoot-container
[student@workstation troubleshoot-container]$ sudo podman build \
> -t troubleshoot-container .
STEP 1: FROM httpd:2.4
...output omitted...
--> e23dfbdceaaadbb734814367dfb61c19575c760d9db4bce23856f13fb277c1de
STEP 5: COMMIT troubleshoot-container
[student@workstation troubleshoot-container]$ cd ~
```

2.2. Verify that the image is created.

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

The new image must be available in the local storage.

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
localhost/troubleshoot-container	latest	e23df...	9 seconds ago	137MB
registry.lab.example.com/httpd	2.4	0eba3...	4 weeks ago	137MB

► 3. Create a new httpd container from the custom image.

```
[student@workstation ~]$ sudo podman run \
--name troubleshoot-container -d \
-p 10080:80 troubleshoot-container
```

► 4. Review the container's log messages and events.

4.1. View the debug log messages from the container using the **podman logs** command:

```
[student@workstation ~]$ sudo podman logs -f troubleshoot-container
... [mpm_event:notice] [pid 1:tid...] AH00489: Apache/2.4.25 (Unix) configur...
... [mpm_event:info] [pid 1:tid...] AH00490: Server built: Mar 21 2017 20:50:17
... [core:notice] [pid 1:tid...] AH00094: Command line: 'httpd -D FOREGROUND'
... [core:debug] [pid 1:tid ...]: AH02639: Using SO_REUSEPORT: yes (1)
... [mpm_event:debug] [pid 6:tid ...]: AH02471: start_threads: Using epoll
... [mpm_event:debug] [pid 7:tid ...]: AH02471: start_threads: Using epoll
```

```
... [mpm_event:debug] [pid 8:tid ...]: AH02471: start_threads: Using epoll
```

Notice the debug logs, available in the standard output.

- 4.2. Open a new terminal and access the home page of the web server by using the **curl** command:

```
[student@workstation ~]$ curl http://127.0.0.1:10080
<html><body><h1>It works!</h1></body></html>
```

- 4.3. Review the new entries in the log. Look in the terminal running the **podman logs** command to see the new entries.

```
[student@workstation ~]$ sudo podman logs troubleshoot-container
...[authz_core:debug] ...: authorization result of Require all granted: granted
...[authz_core:debug] ...: authorization result of <RequireAny>: granted
10.88.0.1 - - [08/Mar/2019:20:30:53 +0000] "GET / HTTP/1.1" 200 45
```

- 4.4. Stop the Podman command with **Ctrl+C**.

Finish

On workstation, run the **lab troubleshoot-container finish** script to complete this lab.

```
[student@workstation ~]$ lab troubleshoot-container finish
```

This concludes the guided exercise.

► LAB

TROUBLESHOOTING CONTAINERIZED APPLICATIONS

PERFORMANCE CHECKLIST

In this lab, you will troubleshoot the OpenShift build and deployment process for a Node.js application.

OUTCOMES

You should be able to identify and solve the problems raised during the build and deployment process of a Node.js application.

BEFORE YOU BEGIN

Open a terminal on `workstation` as the `student` user and run the following command:

```
[student@workstation ~]$ lab troubleshoot-review start
```

1. Log in to OpenShift with the `kubeadmin` username. The password is stored in the `~/.kubeadmin` file.
2. Create a new project named `nodejs-app`:
3. In the `nodejs-app` project, create a new application from the source code located in the Git repository at `http://registry.lab.example.com/nodejs-app`. Name the application `nodejs-dev`. To ensure that application dependencies are downloaded from the local environment, define a build environment variable `npm_config_registry` with a value of `http://services.lab.example.com:8081/nexus/content/groups/nodejs/`.
Expect the build process for the application to fail. Monitor the build process and identify the build failure.
4. Clone the Git repository (`http://services.lab.example.com/nodejs-app`) and update the version of the `express` dependency in the `package.json` file with a value of `4.x`. Commit and push the changes to the Git repository.
5. Rebuild the application. Verify that the application builds without errors.
6. Verify that the application is not running because of a runtime error. Review the logs and identify the problem.
7. Correct the spelling of the dependency in the first line of the `server.js` file. Commit and push changes to the application to the Git repository. Rebuild the application. After the application builds, verify that the application is running.
8. Create a route for the application and test access to the application. Expect an error message. Review the logs to identify the error.
9. Replace `process.environment` with `process.env` in the `server.js` file to fix the error. Commit and push the application changes to the Git repository. Rebuild the application. When

the new application deploys, verify that application does not generate errors when you access the application URL.

Evaluation

Grade your work by running the **lab troubleshoot-review grade** command from your workstation machine. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab troubleshoot-review grade
```

Finish

From workstation, run the **lab troubleshoot-review finish** command to complete this lab.

```
[student@workstation ~]$ lab troubleshoot-review finish
```

This concludes the lab.

► SOLUTION

TROUBLESHOOTING CONTAINERIZED APPLICATIONS

PERFORMANCE CHECKLIST

In this lab, you will troubleshoot the OpenShift build and deployment process for a Node.js application.

OUTCOMES

You should be able to identify and solve the problems raised during the build and deployment process of a Node.js application.

BEFORE YOU BEGIN

Open a terminal on `workstation` as the `student` user and run the following command:

```
[student@workstation ~]$ lab troubleshoot-review start
```

1. Log in to OpenShift with the `kubeadmin` username. The password is stored in the `~/.kubeadmin` file.

```
[student@workstation openshift-s2i]$ oc login -u kubeadmin \
> -p $(cat ~/.kubeadmin) https://cluster-api.lab.example.com:6443
Login successful.
```

You have access to the following projects and can switch between them with 'oc project <projectname>':
...output omitted...



NOTE

If the `oc login` command prompts you about using insecure connections, answer **y** (yes). To skip this check, you can also add the `--insecure-skip-tls-verify` option to the login command.

2. Create a new project named `nodejs-app`:

```
[student@workstation ~]$ oc new-project nodejs-app
Now using project "nodejs-app" on server "https://cluster-
api.lab.example.com:6443".
...output omitted...
```

3. In the `nodejs-app` project, create a new application from the source code located in the Git repository at `http://registry.lab.example.com/nodejs-app`. Name the application `nodejs-dev`. To ensure that application dependencies are downloaded from the local

environment, define a build environment variable `npm_config_registry` with a value of `http://services.lab.example.com:8081/nexus/content/groups/nodejs/`.

Expect the build process for the application to fail. Monitor the build process and identify the build failure.

- 3.1. Run the `oc new-app` command to create the Node.js application. The command is provided in the `~/D0180/labs/troubleshoot-review/command.txt` file.

```
[student@workstation ~]$ oc new-app --build-env npm_config_registry=\> http://services.lab.example.com:8081/nexus/content/groups/nodejs/ \> -i nodejs:4 http://services.lab.example.com/nodejs-app --name nodejs-dev--> Found image a2b5ec2 ...output omitted...  
...output omitted...  
--> Creating resources ...  
imagestream.image.openshift.io "nodejs-dev" created  
buildconfig.build.openshift.io "nodejs-dev" created  
deploymentconfig.apps.openshift.io "nodejs-dev" created  
service "nodejs-dev" created  
--> Success  
Build scheduled, use 'oc logs -f bc/nodejs-dev' to track its progress.  
Application is not exposed. You can expose services to the outside world by  
executing one or more of the commands below:  
'oc expose svc/nodejs-dev'  
Run 'oc status' to view your app.
```

- 3.2. Monitor build progress with the `oc logs -f bc/nodejs-dev` command:

```
[student@workstation ~]$ oc logs -f bc/nodejs-dev  
Cloning "http://services.lab.example.com/nodejs-app" ...  
...output omitted...  
STEP 8: RUN /usr/libexec/s2i/assemble  
---> Installing application source ...  
---> Installing all dependencies  
npm WARN package.json nodejs-app@1.0.0 No repository field.  
npm WARN package.json nodejs-app@1.0.0 No README data  
npm WARN package.json nodejs-app@1.0.0 license should be a valid SPDX license  
expression  
npm ERR! Linux 3.10.0-957.1.3.el7.x86_64  
npm ERR! argv "/opt/rh/rh-nodejs4/root/usr/bin/node" "/opt/rh/rh-nodejs4/root/usr/bin/npm" "install"  
npm ERR! node v4.6.2  
npm ERR! npm v2.15.1  
npm ERR! code ETARGET  
  
npm ERR! notarget No compatible version found: express@'>=4.20.0 <4.21.0'  
npm ERR! notarget Valid install targets:  
npm ERR! notarget [ , "4.13.1", "5.0.0-  
alpha.2", "3.21.2", "4.13.2", "4.13.3", "4.13.4", "4.14.0", "4.14.1", "5.0.0-  
alpha.3", "4.15.0", "5.0.0-alpha.4", "4.15.1", "4.15.2", "5.0.0-  
alpha.5", "4.15.3", "4.15.4", "4.15.5", "5.0.0-  
alpha.6", "4.16.0", "4.16.1", "4.16.2", "4.16.3", "4.16.4", "5.0.0-alpha.7"]
```

```
npm ERR! notarget
...output omitted...
error: build error: error building at ...output omitted...: exit status 1
```

The build process fails, and therefore no application is running. The build log indicates that there is no version of the **express** package that matches a version specification of **4.20.x**.

- 3.3. Use the **oc get pods** command to confirm that the application is not deployed:

```
[student@workstation ~]$ oc get pods
NAME          READY   STATUS    RESTARTS   AGE
nodejs-dev-1-build   0/1     Error      0          2m
```

4. Clone the Git repository (<http://services.lab.example.com/nodejs-app>) and update the version of the **express** dependency in the **package.json** file with a value of **4.x**. Commit and push the changes to the Git repository.

- 4.1. Clone the Git repository.

Open a new terminal window from the **workstation** VM (Applications → Utilities → Terminal) and run the **git** command from the **~/D0180/labs/troubleshoot-s2i** directory.

```
[student@workstation ~]$ cd ~/D0180/labs/troubleshoot-review
[student@workstation troubleshoot-review]$ git clone \
> http://services.lab.example.com/nodejs-app
Cloning into 'nodejs-app'...
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
```

The source code from the application is cloned locally into the **~/D0180/labs/troubleshoot-review/nodejs-app** directory.

- 4.2. Edit the **package.json** file in the **nodejs-app** subdirectory, and change the version of the **express** dependency to **4.x**. Save the file.

```
[student@workstation troubleshoot-review]$ sed -i \
> s/4.20/4.x/ nodejs-app/package.json
```

The file contains the following content:

```
[student@workstation troubleshoot-review]$ cat nodejs-app/package.json
{
  "name": "nodejs-app",
  "version": "1.0.0",
  "description": "Hello World App",
  "main": "server.js",
  "author": "Red Hat Training",
  "license": "ASL",
  "dependencies": {
    "express": "4.x",
```

```
        "html-errors": "latest"
    }
}
```

- 4.3. Commit and push the changes made to the project.

From the terminal window, run the following command to commit and push the changes:

```
[student@workstation troubleshoot-review]$ cd nodejs-app
[student@workstation nodejs-app]$ git commit -am "Fixed Express release"
...output omitted...
1 file changed, 1 insertion(+), 1 deletion(-)
[student@workstation nodejs-app]$ git push
...output omitted...
To http://services.lab.example.com/nodejs-app
 ef6557d..73a82cd master -> master
```

5. Rebuild the application. Verify that the application builds without errors.

- 5.1. Use the **oc start-build** command to rebuild the application.

```
[student@workstation nodejs-app]$ oc start-build bc/nodejs-dev
build "nodejs-dev-2" started
```

- 5.2. Use the **oc logs** command to monitor the build process logs:

```
[student@workstation nodejs-app]$ oc logs -f bc/nodejs-dev
Cloning "https://registry.lab.example.com/nodejs-app"
...output omitted...
Successfully pushed //image-registry.openshift-image-registry.svc:5000...
Push successful
```

The build succeeds if an image is pushed to the internal OpenShift registry.

6. Verify that the application is not running because of a runtime error. Review the logs and identify the problem.

- 6.1. Use the **oc get pods** command to check the status of the deployment of the application pod. Eventually, you see that the first application deployment has a status of **CrashLoopBackoff**.

```
[student@workstation nodejs-app]$ oc get pods
NAME          READY   STATUS      RESTARTS   AGE
nodejs-dev-1-86gg5  0/1     CrashLoopBackOff  6          7m
nodejs-dev-1-build  0/1     Error       0          26m
nodejs-dev-2-build  0/1     Completed    0          11m
```

- 6.2. Use the **oc logs -f dc/nodejs-dev** command to follow the logs for the application deployment:

```
[student@workstation nodejs-app]$ oc logs -f dc/nodejs-dev
Environment:
```

```

DEV_MODE=false
NODE_ENV=production
DEBUG_PORT=5858
...output omitted...

Error: Cannot find module 'http-error'

...output omitted...

npm info nodejs-app@1.0.0 Failed to exec start script
...output omitted...
npm ERR!
npm ERR! Failed at the nodejs-app@1.0.0 start script 'node server.js'.
npm ERR! This is most likely a problem with the nodejs-app package,
npm ERR! not with npm itself.
...output omitted...

```

The log indicates that the **server.js** file attempts to load a module named **http-error**. The dependencies variable in the **packages** file indicates that the module name is **html-errors**, not **http-error**.

7. Correct the spelling of the dependency in the first line of the **server.js** file. Commit and push changes to the application to the Git repository. Rebuild the application. After the application builds, verify that the application is running.
 - 7.1. Correct the spelling of the module in the first line of the **server.js** from **http-error** to **html-errors**. Save the file.

```
[student@workstation nodejs-app]$ sed -i \
> s/http-error/html-errors/ server.js
```

The file contains the following content:

```

[student@workstation nodejs-app]$ cat server.js
var createError = require('html-errors');

var express = require('express');
app = express();

app.get('/', function (req, res) {
  res.send('Hello World from pod: ' + process.env.HOSTNAME + '\n')
});

app.listen(8080, function () {
  console.log('Example app listening on port 8080!');
});
```

- 7.2. Commit and push the changes made to the project.

```
[student@workstation nodejs-app]$ git commit -am "Fixed module typo"
...output omitted...
1 file changed, 1 insertion(+), 1 deletion(-)
[student@workstation nodejs-app]$ git push
```

```
...output omitted...
To http://services.lab.example.com/nodejs-app
ef6557d..73a82cd master -> master
```

- 7.3. Use the **oc start-build** command to rebuild the application.

```
[student@workstation nodejs-app]$ oc start-build bc/nodejs-dev
build "nodejs-dev-3" started
```

- 7.4. Use the **oc logs** command to monitor the build process logs:

```
[student@workstation nodejs-app]$ oc logs -f bc/nodejs-dev
Cloning "https://registry.lab.example.com/nodejs-app"
...output omitted...
Successfully pushed //image-registry.openshift-image-registry.svc:5000/nodejs/
nodejs-
dev:latest@sha256:9312d5f4f8e834bda235d1f8b637f95ebc67fea801709f5302707aa11bd74342

Push successful
```

- 7.5. Use the **oc get pods -w** command to monitor the deployment of pods for the **nodejs-dev** application:

```
[student@workstation nodejs-app]$ oc get pods -w
NAME          READY   STATUS    RESTARTS   AGE
nodejs-dev-1-build  0/1     Error      0          6h9m
nodejs-dev-2-build  0/1     Completed   0          5h55m
nodejs-dev-2-xt8q4  1/1     Running     0          4m
nodejs-dev-3-build  0/1     Completed   0          7m57s
```

After a third build, the second deployment results in a status of **Running**.

8. Create a route for the application and test access to the application. Expect an error message. Review the logs to identify the error.

- 8.1. Use the **oc expose** command to create a route for the **nodejs-dev** application:

```
[student@workstation nodejs-app]$ oc expose svc nodejs-dev
route.route.openshift.io/nodejs-dev exposed
```

- 8.2. Use the **oc get route** command to retrieve the URL of **nodejs-dev** route:

```
[student@workstation nodejs-app]$ oc get route
NAME        HOST/PORT
nodejs-dev  nodejs-dev-nodejs-app.apps.cluster.lab.example.com ...
```

- 8.3. Use the **curl** to access the route. Expect an error message to display.

```
[student@workstation nodejs-app]$ curl \
> nodejs-dev-nodejs-app.apps.cluster.lab.example.com
<!DOCTYPE html>
```

```
<html lang="en">
<head>
<meta charset="utf-8">
<title>Error</title>
</head>
<body>
<pre>Internal Server Error</pre>
</body>
</html>
```

8.4. Review the logs for the **nodejs-dev** deployment configuration:

```
[student@workstation nodejs-app]$ oc logs dc/nodejs-dev
Environment:
  DEV_MODE=false
  NODE_ENV=production
  DEBUG_PORT=5858
Launching via npm...
npm info it worked if it ends with ok
npm info using npm@2.15.1
npm info using node@v4.6.2
npm info prestart nodejs-app@1.0.0
npm info start nodejs-app@1.0.0

> nodejs-app@1.0.0 start /opt/app-root/src
> node server.js

Example app listening on port 8080!
TypeError: Cannot read property 'HOSTNAME' of undefined
...output omitted...
```

The corresponding section of the **server.js** file is:

```
app.get('/', function (req, res) {
  res.send('Hello World from pod: ' + process.env.HOSTNAME + '\n')
});
```

A **process** object in Node.js contains a reference to a **env** object, not a **environment** object.

9. Replace **process.environment** with **process.env** in the **server.js** file to fix the error. Commit and push the application changes to the Git repository. Rebuild the application. When the new application deploys, verify that application does not generate errors when you access the application URL.

 - 9.1. Replace **process.environment** with **process.env** in the **server.js** file to fix the error.

```
[student@workstation nodejs-app]$ sed -i \
```

```
> s/process.environment/process.env/ server.js
```

The file contains the following content:

```
[student@workstation nodejs-app]$ cat server.js
var createError = require('html-errors');

var express = require('express');
app = express();

app.get('/', function (req, res) {
  res.send('Hello World from pod: ' + process.env.HOSTNAME + '\n')
});

app.listen(8080, function () {
  console.log('Example app listening on port 8080!');
});
```

9.2. Commit and push the changes made to the project.

```
[student@workstation nodejs-app]$ git commit -am "Fixed process.env"
...output omitted...
1 file changed, 1 insertion(+), 1 deletion(-)
[student@workstation nodejs-app]$ git push
...output omitted...
To http://services.lab.example.com/nodejs-app
 ef6557d..73a82cd master -> master
```

9.3. Use the **oc start-build** command to rebuild the application.

```
[student@workstation nodejs-app]$ oc start-build bc/nodejs-dev
build "nodejs-dev-4" started
```

9.4. Use the **oc logs** command to monitor the build process logs:

```
[student@workstation nodejs-app]$ oc logs -f bc/nodejs-dev
Cloning "https://registry.lab.example.com/nodejs-app"
...output omitted...
Successfully pushed //image-registry.openshift-image-registry.svc:...
Push successful
```

9.5. Use the **oc get pods** command to monitor the deployment of pods for the **nodejs-dev** application:

```
[student@workstation nodejs-app]$ oc get pods
NAME          READY   STATUS    RESTARTS   AGE
nodejs-dev-1-build  0/1     Error      0          7h
nodejs-dev-2-build  0/1     Completed   0          6h
nodejs-dev-3-build  0/1     Completed   0          1h
nodejs-dev-3-m7wvj  1/1     Running    0          46s
```

```
nodejs-dev-4-build  0/1    Completed   0      3m
```

After a fourth build, the third deployment has a status of **Running**.

- 9.6. Use the **curl** command to test the application. The application displays a **Hello World** message containing the host name of the application pod:

```
[student@workstation nodejs-app]$ curl \
> nodejs-dev-nodejs-app.apps.cluster.lab.example.com
Hello World from pod: nodejs-dev-3-m7wvj
```

Evaluation

Grade your work by running the **lab troubleshoot-review grade** command from your workstation machine. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab troubleshoot-review grade
```

Finish

From workstation, run the **lab troubleshoot-review finish** command to complete this lab.

```
[student@workstation ~]$ lab troubleshoot-review finish
```

This concludes the lab.

SUMMARY

In this chapter, you learned:

- Applications typically log activity, such as events, warnings and errors, to aid the analysis of application behavior.
- Container applications should print log data to standard output, instead of to a file, to enable easy access to logs.
- To review the logs for a container deployed locally with Podman, use the **podman logs** command.
- Use the **oc logs** command to access logs for **BuildConfig** and **DeploymentConfig** objects, as well as individual pods within an OpenShift project.
- The **-f** option allows you to monitor the log output in near real-time for both the **podman logs** and **oc logs** commands.
- Use the **oc port-forward** command to connect directly to a port on an application pod. You should only leverage this technique on non-production pods, because interactions can alter the behavior of the pod.

CHAPTER 9

COMPREHENSIVE REVIEW

GOAL

Demonstrate how to containerize a software application, test it with Podman, and deploy it to an OpenShift cluster.

OBJECTIVE

Review the concepts in this course to assist in completing the comprehensive review lab.

SECTIONS

- Comprehensive Review

LAB

- Comprehensive Review of Introduction to Containers, Kubernetes, and Red Hat OpenShift

COMPREHENSIVE REVIEW

OBJECTIVES

After completing this section, you should be able to demonstrate knowledge and skills learned in *Introduction to Containers, Kubernetes, and Red Hat OpenShift*.

REVIEWING INTRODUCTION TO CONTAINERS, KUBERNETES, AND RED HAT OPENSHIFT

Before beginning the comprehensive review lab for this course, students should be comfortable with the topics covered in the following chapters.

Chapter 1, *Introducing Container Technology*

Describe how applications run in containers orchestrated by Red Hat OpenShift Container Platform.

- Describe the difference between container applications and traditional deployments.
- Describe the basics of container architecture.
- Describe the benefits of orchestrating applications and OpenShift Container Platform.

Chapter 2, *Creating Containerized Services*

Provision a service using container technology.

- Create a database server from a container image.

Chapter 3, *Managing Containers*

Modify prebuilt container images to create and manage containerized services.

- Manage a container's life cycle from creation to deletion.
- Save container application data with persistent storage.
- Describe how to use port forwarding to access a container.

Chapter 4, *Managing Container Images*

Manage the life cycle of a container image from creation to deletion.

- Search for and pull images from remote registries.
- Export, import, and manage container images locally and in a registry.

Chapter 5, *Creating Custom Container Images*

Design and code a Dockerfile to build a custom container image.

- Describe the approaches for creating custom container images.

- Create a container image using common Dockerfile commands.

Chapter 6, Deploying Containerized Applications on OpenShift

Deploy single container applications on OpenShift Container Platform.

- Describe the architecture of Kubernetes and Red Hat OpenShift Container Platform.
- Create standard Kubernetes resources.
- Create a route to a service.
- Build an application using the Source-to-Image facility of OpenShift Container Platform.
- Create an application using the OpenShift web console.

Chapter 7, Deploying Multi-Container Applications

Deploy applications that are containerized using multiple container images.

- Describe considerations for containerizing applications with multiple container images.
- Deploy a multi-container application on OpenShift using a template.

Chapter 8, Troubleshooting Containerized Applications

Troubleshoot a containerized application deployed on OpenShift.

- Troubleshoot an application build and deployment on OpenShift.
- Implement techniques for troubleshooting and debugging containerized applications.

GENERAL CONTAINER, KUBERNETES, AND OPENSHIFT HINTS

These hints may save some time in completing the comprehensive review lab:

- The **podman** command allows you to build, run, and manage container images. Use the **man podman** command to access Podman documentation. Use the **man podman subcommand** command to get more information about each subcommand.
- The **oc** command allows you to create and manage OpenShift resources. Use the **man oc** or **oc help** commands to access OpenShift command-line documentation. OpenShift commands that are particularly useful include:

oc login -u <username> -p <password>

Log in to OpenShift as the specified user. In this classroom, there is currently a single user available: **kubeadmin**. You can find the password for this user in the **~/.kube/admin** file.

oc new-project

Create a new project (*namespace*) to contain OpenShift resources.

oc project

Select the current project (*namespace*) to which all subsequent commands apply.

oc create -f

Create a resource from a file.

oc process

Processes a template file applying the parameter values to each included resource. Create those resources with the **oc create** command.

oc get

Display the runtime status and attributes of OpenShift resources.

oc describe

Display detailed information about OpenShift resources.

oc delete

Delete OpenShift resources.

- Before mounting any volumes on the Podman and OpenShift host, ensure you apply the correct SELinux context to the directory. The correct context is `container_file_t`. Also, make sure the ownership and permissions of the directory are set according to the `USER` directive in the Dockerfile that was used to build the container being deployed. Most of the time you will have to use the numeric UID and GID rather than the user and group names to adjust ownership and permissions of the volume directory.
- In this classroom, all RPM repositories are defined locally. You must configure the repository definitions in a custom container image (Dockerfile) before running `yum` commands.
- When executing commands in a Dockerfile, combine as many related commands as possible into one `RUN` directive. This reduces the number of image layers in the container image.
- A best practice for designing a Dockerfile includes the use of environment variables for specifying repeated constants throughout the file.

► LAB

CONTAINERIZING AND DEPLOYING A SOFTWARE APPLICATION

In this review, you will containerize a Nexus Server, build and test it using Podman, and deploy it to an OpenShift cluster.

OUTCOMES

You should be able to:

- Write a Dockerfile that successfully containerizes a Nexus server.
- Build a Nexus server container image that deploys using Podman.
- Deploy the Nexus server container image to an OpenShift cluster.

BEFORE YOU BEGIN

Run the set-up script for this comprehensive review.

```
[student@workstation ~]$ lab comprehensive-review start
```

The lab files are located in the **/home/student/D0180/labs/comprehensive-review** directory. The solution files are located in the **/home/student/D0180/solutions/comprehensive-review** directory.

INSTRUCTIONS

Use the following steps to create and test a containerized Nexus server both locally and in OpenShift:

1. Create a container image that starts an instance of a Nexus server:
 - The **/home/student/D0180/labs/comprehensive-review/image** directory contains files for building the container image. Execute the **get-nexus-bundle.sh** script to retrieve the Nexus server files.
 - Write a Dockerfile that containerizes the Nexus server. The Dockerfile must be located in the **/home/student/D0180/labs/comprehensive-review/image** directory. The Dockerfile must also:
 - Use a base image of **rhe17:7.5** and set an arbitrary maintainer.
 - Set the environment variable **NEXUS_VERSION** to **2.14.3-02**, and set **NEXUS_HOME** to **/opt/nexus**.
 - Install the **java-1.8.0-openjdk-devel** package

The RPM repositories are configured in the provided **training.repo** file. Be sure to add this file to the container in the **/etc/yum.repos.d** directory.

- Run a command to create a nexus user and group. They both have a UID and GID of **1001**.
- Add the **nexus-2.14.3-02-bundle.tar.gz** and **nexus-start.sh** files to the **\${NEXUS_HOME}/** directory.

Run a command, **ln -s \${NEXUS_HOME}/nexus-\${NEXUS_VERSION} \${NEXUS_HOME}/nexus2**, to create a symlink in the container. Run a command to recursively change the ownership of the Nexus home directory to **nexus:nexus**.

- Make the container run as the nexus user, and set the working directory to **/opt/nexus**.
- Define a volume mount point for the **/opt/nexus/sonatype-work** container directory. The Nexus server stores data in this directory.
- Set the default container command to **nexus-start.sh**.

There are two ***.snippet** files in the **/home/student/D0180/labs/comprehensive-review/images** directory that provide the commands needed to create the nexus account and install Java. Use the files to assist you in writing the Dockerfile.

- Build the container image with the name **nexus**.
- Build and test the container image using Podman with a volume mount:
 - Use the script **/home/student/D0180/labs/comprehensive-review/deploy/local/run-persistent.sh** to start a new container with a volume mount.
 - Review the container logs to verify that the server is started and running.
 - Test access to the container service using the URL: **http://<container IP address>:8081/nexus**.
 - Remove the test container.
 - Deploy the Nexus server container image to the OpenShift cluster. You must:
 - Tag the Nexus server container image as **registry.lab.example.com/nexus:latest**, and push it to the private registry.
 - Create an OpenShift project with a name of **review**. Set the security context policy by running the **setpolicy.sh** script located in the **/home/student/D0180/labs/comprehensive-review/deploy/openshift** directory.
 - Execute the **create-pv.sh** script in the **deploy/openshift** subdirectory. This creates the persistent volume for the Nexus server's persistent data.
 - Process the **deploy/openshift/resources/nexus-template.json** template and create the Kubernetes resources.
 - Create a route for the Nexus service. Verify that you can access **http://nexus-review.apps.cluster.lab.example.com/nexus/** from workstation.

Evaluation

After deploying the Nexus server container image to the OpenShift cluster, verify your work by running the lab grading script:

```
[student@workstation ~]$ lab comprehensive-review grade
```

Finish

On workstation, run the **lab comprehensive-review finish** command to complete this lab.

```
[student@workstation ~]$ lab comprehensive-review finish
```

This concludes the lab.

► SOLUTION

CONTAINERIZING AND DEPLOYING A SOFTWARE APPLICATION

In this review, you will containerize a Nexus Server, build and test it using Podman, and deploy it to an OpenShift cluster.

OUTCOMES

You should be able to:

- Write a Dockerfile that successfully containerizes a Nexus server.
- Build a Nexus server container image that deploys using Podman.
- Deploy the Nexus server container image to an OpenShift cluster.

BEFORE YOU BEGIN

Run the set-up script for this comprehensive review.

```
[student@workstation ~]$ lab comprehensive-review start
```

The lab files are located in the **/home/student/D0180/labs/comprehensive-review** directory. The solution files are located in the **/home/student/D0180/solutions/comprehensive-review** directory.

INSTRUCTIONS

Use the following steps to create and test a containerized Nexus server both locally and in OpenShift:

1. Create a container image that starts an instance of a Nexus server:
 - The **/home/student/D0180/labs/comprehensive-review/image** directory contains files for building the container image. Execute the **get-nexus-bundle.sh** script to retrieve the Nexus server files.
 - Write a Dockerfile that containerizes the Nexus server. The Dockerfile must be located in the **/home/student/D0180/labs/comprehensive-review/image** directory. The Dockerfile must also:
 - Use a base image of **rhel7:7.5** and set an arbitrary maintainer.
 - Set the environment variable **NEXUS_VERSION** to **2.14.3-02**, and set **NEXUS_HOME** to **/opt/nexus**.
 - Install the **java-1.8.0-openjdk-devel** package

The RPM repositories are configured in the provided **training.repo** file. Be sure to add this file to the container in the **/etc/yum.repos.d** directory.

- Run a command to create a nexus user and group. They both have a UID and GID of **1001**.
- Add the **nexus-2.14.3-02-bundle.tar.gz** and **nexus-start.sh** files to the **\${NEXUS_HOME}/** directory.

Run a command, **ln -s \${NEXUS_HOME}/nexus-\$NEXUS_VERSION \${NEXUS_HOME}/nexus2**, to create a symlink in the container. Run a command to recursively change the ownership of the Nexus home directory to **nexus:nexus**.

- Make the container run as the nexus user, and set the working directory to **/opt/nexus**.
- Define a volume mount point for the **/opt/nexus/sonatype-work** container directory. The Nexus server stores data in this directory.
- Set the default container command to **nexus-start.sh**.

There are two ***.snippet** files in the **/home/student/D0180/labs/comprehensive-review/images** directory that provide the commands needed to create the nexus account and install Java. Use the files to assist you in writing the Dockerfile.

- Build the container image with the name **nexus**.

- Execute the **get-nexus-bundle.sh** script to retrieve the Nexus server files.

```
[student@workstation ~]$ cd /home/student/D0180/labs/comprehensive-review/image
[student@workstation image]$ ./get-nexus-bundle.sh
```

- Write a Dockerfile that containerizes the Nexus server. Go to the **/home/student/D0180/labs/comprehensive-review/image** directory and create the Dockerfile.

- Specify the base image to use:

```
FROM rhel7:7.5
```

- Enter an arbitrary name and email as the maintainer:

```
FROM rhel7:7.5
MAINTAINER username <username@example.com>
```

- Set the environment variables for NEXUS_VERSION and NEXUS_HOME:

```
FROM rhel7:7.5
MAINTAINER username <username@example.com>

ENV NEXUS_VERSION=2.14.3-02 \
    NEXUS_HOME=/opt/nexus
```

- Add the **training.repo** repository to the **/etc/yum.repos.d** directory. Install the Java package using **yum** command.

```
ENV NEXUS_VERSION=2.14.3-02 \
NEXUS_HOME=/opt/nexus

ADD training.repo /etc/yum.repos.d/training.repo
RUN yum install -y --setopt=tsflags=nodocs java-1.8.0-openjdk-devel && \
    yum clean all -y
```

5. Create the server home directory and service account and group.

```
...
RUN groupadd -r nexus -f -g 1001 && \
    useradd -u 1001 -r -g nexus -m -d ${NEXUS_HOME} \
    -s /sbin/nologin \
    -c "Nexus User" nexus
```

6. Install the Nexus server software at NEXUS_HOME and add the startup script.
Note that the **ADD** directive will extract the Nexus files.

Create the **nexus2** symbolic link pointing to the Nexus server directory.
Recursively change the ownership of the **`\${NEXUS_HOME}`** directory to **nexus:nexus**.

```
...
ADD nexus-${NEXUS_VERSION}-bundle.tar.gz ${NEXUS_HOME}/
ADD nexus-start.sh ${NEXUS_HOME}/

RUN ln -s ${NEXUS_HOME}/nexus-${NEXUS_VERSION} \
    ${NEXUS_HOME}/nexus2 && \
    chown -R nexus:nexus ${NEXUS_HOME}
```

7. Make the container run as the **nexus** user and make the working directory **/opt/nexus**:

```
...
USER nexus
WORKDIR ${NEXUS_HOME}
```

8. Define a volume mount point to store the Nexus server persistent data:

```
...
VOLUME ["/opt/nexus/sonatype-work"]
```

9. Set the **CMD** instruction to **["sh", "nexus-start.sh"]**. The completed Dockerfile reads as follows:

```
FROM rhel7:7.5
MAINTAINER username <username@example.com>

ENV NEXUS_VERSION=2.14.3-02 \
NEXUS_HOME=/opt/nexus

ADD training.repo /etc/yum.repos.d/training.repo
```

```
RUN yum install -y --setopt=tsflags=nodocs java-1.8.0-openjdk-devel && \
    yum clean all -y

RUN groupadd -r nexus -f -g 1001 && \
    useradd -u 1001 -r -g nexus -m -d ${NEXUS_HOME} \
        -s /sbin/nologin \
        -c "Nexus User" nexus

ADD nexus-${NEXUS_VERSION}-bundle.tar.gz ${NEXUS_HOME}/
ADD nexus-start.sh ${NEXUS_HOME}/

RUN ln -s ${NEXUS_HOME}/nexus-${NEXUS_VERSION} \
    ${NEXUS_HOME}/nexus2 && \
    chown -R nexus:nexus ${NEXUS_HOME}

USER nexus
WORKDIR ${NEXUS_HOME}

VOLUME ["/opt/nexus/sonatype-work"]

CMD ["sh", "nexus-start.sh"]
```

1.3. Build the container image with the name **nexus**.

```
[student@workstation image]$ sudo podman build -t nexus .
STEP 1: FROM rhel7:7.5
Getting image source signatures
...output omitted...
STEP 25: COMMIT nexus
```

2. Build and test the container image using Podman with a volume mount:

- Use the script **/home/student/D0180/labs/comprehensive-review/deploy/local/run-persistent.sh** to start a new container with a volume mount.
- Review the container logs to verify that the server is started and running.
- Test access to the container service using the URL: `http://<container IP address>:8081/nexus`.
- Remove the test container.

2.1. Execute the **run-persistent.sh** script. Replace the container name as shown in the output of the **podman ps** command.

```
[student@workstation images]$ cd /home/student/D0180/labs/comprehensive-review
[student@workstation comprehensive-review]$ cd deploy/local
[student@workstation local]$ ./run-persistent.sh
80970007036bbb313d8eeb7621fada0ed3f0b4115529dc50da4dccef0da34533
```

2.2. Review the container logs to verify that the server is started and running.

```
[student@workstation local]$ sudo podman ps \
> --format="table {{.ID}} {{.Names}} {{.Image}}"
```

```
CONTAINER ID NAMES IMAGE
81f480f21d47 inspiring_poincare localhost/nexus:latest
[student@workstation local]$ sudo podman logs -f inspiring_poincare
...output omitted...
... INFO [jetty-main-1] ...jetty.JettyServer - Running
... INFO [main] ...jetty.JettyServer - Started
Ctrl+C
```

- 2.3. Inspect the running container to determine its IP address. Provide this IP address to the **curl** command to test the container.

```
[student@workstation local]$ sudo podman inspect \
> -f '{{.NetworkSettings.IPAddress}}' inspiring_poincare
10.88.0.12
[student@workstation local]$ curl -v 10.88.0.12:8081/nexus/
About to connect() to 10.88.0.12 port 8081 (#0)
* Trying 10.88.0.12...
* Connected to 10.88.0.12 (10.88.0.12) port 8081 (#0)
> GET /nexus/ HTTP/1.1
> User-Agent: curl/7.29.0
> Host: 10.88.0.12:8081
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Tue, 05 Mar 2019 16:59:30 GMT
< Server: Nexus/2.14.3-02
...output omitted...
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
<title>Nexus Repository Manager</title>
...output omitted...
```

- 2.4. Remove the test container.

```
[student@workstation local]$ sudo podman kill inspiring_poincare
81f480f21d475af683b4b003ca6e002d37e6aaa581393d3f2f95a1a7b7eb768b
```

3. Deploy the Nexus server container image to the OpenShift cluster. You must:

- Tag the Nexus server container image as **registry.lab.example.com/nexus:latest**, and push it to the private registry.
- Create an OpenShift project with a name of **review**. Set the security context policy by running the **setpolicy.sh** script located in the **/home/student/D0180/labs/comprehensive-review/deploy/openshift** directory.
- Execute the **create-pv.sh** script in the **deploy/openshift** subdirectory. This creates the persistent volume for the Nexus server's persistent data.
- Process the **deploy/openshift/resources/nexus-template.json** template and create the Kubernetes resources.
- Create a route for the Nexus service. Verify that you can access **http://nexus-review.apps.cluster.lab.example.com/nexus/** from workstation.

- 3.1. Publish the Nexus server container image to the classroom private registry:

```
[student@workstation local]$ sudo podman push localhost/nexus:latest \
> registry.lab.example.com/nexus:latest
Getting image source signatures
...output omitted...
Writing manifest to image destination
Storing signatures
```

- 3.2. Create the OpenShift project and set the security context policy:

```
[student@workstation local]$ cd ~/D0180/labs/comprehensive-review/deploy/openshift
[student@workstation openshift]$ oc login -u kubeadmin -p $(cat ~/.kubeadmin) \
> --insecure-skip-tls-verify=true https://cluster-api.lab.example.com:6443
Login successful.
...output omitted...
[student@workstation openshift]$ oc new-project review
Now using project "review" on server "https://cluster-api.lab.example.com:6443".
You can add applications to this project with the 'new-app' command. For example,
try:
oc new-app centos/ruby-25-centos7-https://github.com/sclorg/ruby-ex.git
to build a new example application in Ruby.
[student@workstation openshift]$ ./setpolicy.sh
securitycontextconstraints.security.openshift.io/anyuid added to:
["system:serviceaccount:review=default"]
```

- 3.3. Execute the **create-pv.sh** script in the **deploy/openshift** subdirectory.

```
[student@workstation openshift]$ ./create-pv.sh
...output omitted...
```

```
persistentvolume "pv-nexus" created
```

- 3.4. Process the template and create the Kubernetes resources:

```
[student@workstation openshift]$ oc process -f resources/nexus-template.json \
> | oc create -f -
service/nexus created
persistentvolumeclaim/nexus created
deploymentconfig.apps.openshift.io/nexus created
[student@workstation openshift]$ oc get pods
NAME          READY   STATUS    RESTARTS   AGE
nexus-1-wk8rv 1/1     Running   1          1m
```

- 3.5. Expose the service by creating a route:

```
[student@workstation openshift]$ oc expose svc/nexus
route.route.openshift.io/nexus exposed.
[student@workstation openshift]$ oc get route -o yaml
apiVersion: v1
items:
- apiVersion: route.openshift.io/v1
kind: Route
...output omitted...
spec:
host: nexus-review.apps.cluster.lab.example.com
...output omitted...
```

- 3.6. Use a browser to connect to the Nexus server web application at <http://nexus-review.apps.cluster.lab.example.com/nexus/>.

Evaluation

After deploying the Nexus server container image to the OpenShift cluster, verify your work by running the lab grading script:

```
[student@workstation ~]$ lab comprehensive-review grade
```

Finish

On workstation, run the **lab comprehensive-review finish** command to complete this lab.

```
[student@workstation ~]$ lab comprehensive-review finish
```

This concludes the lab.

APPENDIX A

IMPLEMENTING MICROSERVICES ARCHITECTURE

GOAL

Refactor an application into microservices.

OBJECTIVES

- Divide an application across multiple containers to separate distinct layers and services.

SECTIONS

- Implementing Microservices Architectures (with Guided Exercise)

IMPLEMENTING MICROSERVICES ARCHITECTURES

OBJECTIVES

After completing this section, you should be able to:

- Divide an application across multiple containers to separate distinct layers and services.
- Describe typical approaches to breaking up a monolithic application into multiple deployable units.
- Describe how to break the To Do List application into three containers matching its logical tiers.

BENEFITS OF BREAKING UP A MONOLITHIC APPLICATION INTO CONTAINERS

Traditional application development typically has many distinct functions packaged as a single deployment unit, or a *monolithic* application. Traditional development may also deploy supporting services, such as databases and other middleware services, on the same server as the application. While monolithic applications can still be deployed into a container, many of the advantages of a container architecture, such as scalability and agility, are not as prevalent. Breaking up monoliths requires careful consideration and it is recommended that in microservices applications each microservice runs the minimum functionality that can be executed in isolation on each container.

Having smaller containers and breaking up an application and its supporting services into multiple containers provides many advantages, such as:

- Higher hardware utilization, because smaller containers are easier to fit into available host capacity.
- Easier scaling, because parts of the application can be scaled to support an increased workload without scaling other parts of the application.
- Easier upgrades, because developers can update parts of the application without affecting other parts of the same application.

Two popular ways of breaking up an application are as follows:

- Tiers: based on architectural layers.
- Services: based on application functionality.

DIVIDING BASED ON LAYERS (TIERS)

A common way developers organize applications is in tiers, based on how close the functions are to end users and how far from data stores. A good example of the traditional 3-tier architecture is presentation, business logic, and persistence.

This logical architecture usually corresponds to a physical deployment architecture, where the presentation layer would be deployed to a web server, the business layer to an application server, and the persistence layer to a database server.

Breaking up an application into tiers allows developers to specialize in particular technologies based on the application's tiers. For example, some developers focus on web applications, while others prefer database development. Another advantage is the ability to provide alternative tier implementations based on different technologies; for example, creating a mobile application as another front end for an existing application. The mobile application would be an alternative presentation tier, reusing the business and persistence tiers of the original web application.

Smaller applications usually have the presentation and business tiers deployed as a single unit. For example, to the same web server, but as the load increases, the presentation layer is moved to its own deployment unit to spread the load. Smaller applications might even embed the database. Developers often build and deploy more demanding applications in this monolithic fashion.

When developers break up a monolithic application into tiers, they usually apply several changes:

- Connection parameters to a database and other middleware services, such as messaging, were hard-coded to fixed IP addresses or host names, usually `localhost`. They need to be parameterized to point to external servers that might be different from development to production.
- In the case of web applications, Ajax calls cannot be made using relative URLs. They need to use an absolute URL pointing to a fixed public DNS host name.
- Modern web browsers refuse Ajax calls to servers different from the one containing the script that makes the call, as a security measure. The application needs to have permissions for *cross-origin resource sharing (CORS)*.

After application tiers are divided so that they can run from different servers, there should be no problem running them from different containers.

DIVIDING BASED ON DISCRETE SERVICES

Most complex applications are composed of many semi-independent services. For example, an online store would have a product catalog, shopping cart, payment, and so on.

When a particular service in a monolithic application degrades, scaling the service to improve performance implies scaling all of the other constituent application services. If however the degraded service is part of a microservices architecture, the affected service is scaled independent of the other application services. The following figure illustrates service scaling for both a monolithic and microservices-based architecture:

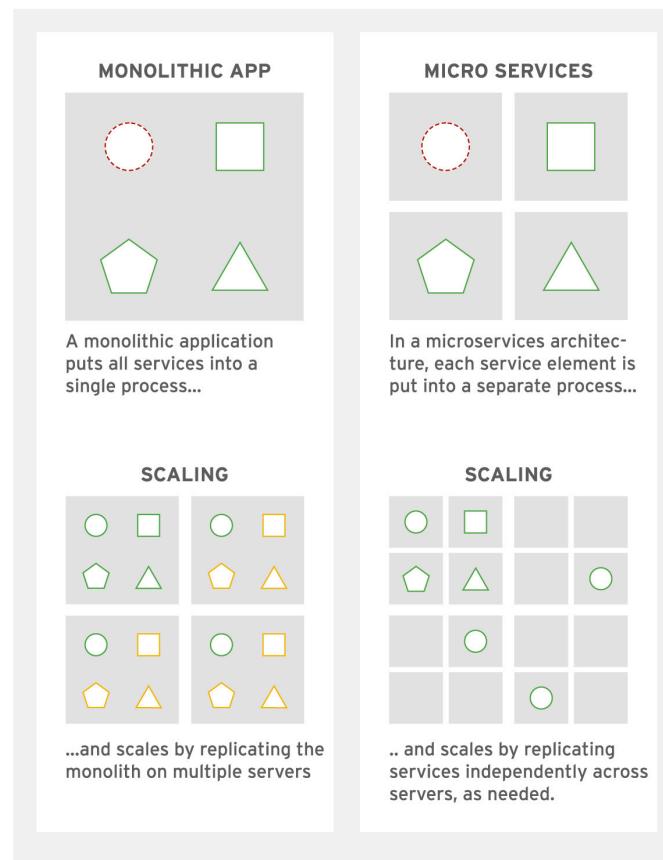


Figure A.1: Comparison of application scaling in a monolithic architecture versus a microservices architecture

Both traditional *service-oriented architectures (SOA)* and more recent *microservices* architectures package and deploy those function sets as distinct units. This allows each function set to be developed by its own team, updated, and scaled without disturbing other function sets (or services). Cross-functional concerns such as authentication can also be packaged and deployed as services that are consumed by other service implementations.

Splitting each concern into a separated server might result in many applications. They are logically architected, packaged, and deployed as a small number of units, sometimes even as a single monolithic unit using a service approach.

Containers enable architectures based on services to be materialized during deployment. That is the reason microservices and containers usually come together. However, containers alone are not enough; they need to be complemented by orchestration tools to manage dependencies among services.

The microservices architecture takes service-based architectures to the extreme. A service is as small as it can be (without breaking a function set) and is deployed and managed as an independent unit, instead of part of a bigger application. This allows existing microservices to be reused to create new applications.

To break an application into services, it needs the same kind of change as when breaking into tiers; for example, parameterize connection parameters to databases and other middleware services and deal with web browser security protections.

REFACTORING THE TO DO LIST APPLICATION

The To Do List application is a simple application with a single function set, so breaking it up into services is not really meaningful. However, refactoring it into presentation and business tiers, that is, into a front end and a back end to deploy into distinct containers, illustrates the same kind of changes that breaking up a typical application into services would need.

The following figure shows the To Do List application deployed into three containers, one for each tier:

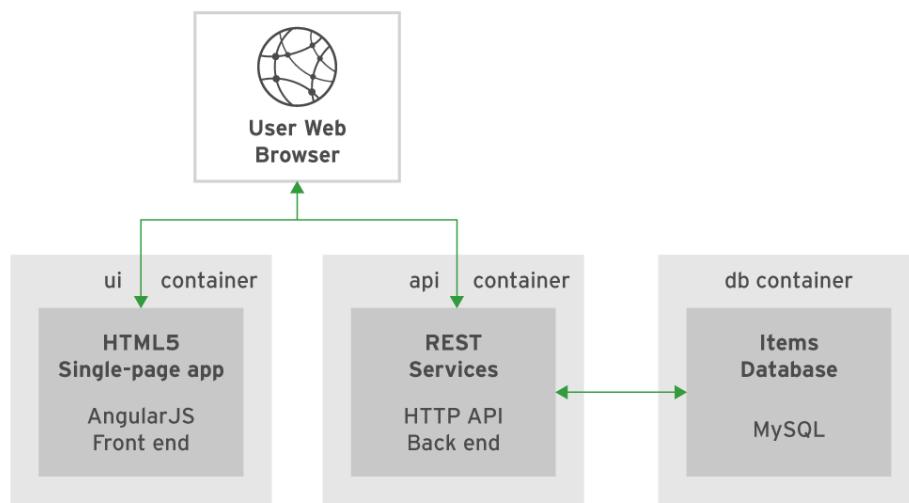


Figure A.2: To Do List application broken into tiers and each deployed as containers

Comparing the source code of the original monolithic application with the refactored one, this is an overview of the changes:

- The front-end JavaScript in **script/items.js** uses `workstation.lab.example.com` as the host name to reach the back end.
- The back end uses environment variables to get the database connection parameters.
- The back end has to reply to requests using the HTTP **OPTIONS** verb with headers telling the web browser to accept requests coming from different DNS domains using **CORS**.

Other versions of the back end service might have similar changes. Each programming language and REST framework have their own syntax and features.



REFERENCES

Monolithic application page in Wikipedia

https://en.wikipedia.org/wiki/Monolithic_application

CORS page in Wikipedia

https://en.wikipedia.org/wiki/Cross-origin_resource_sharing

► GUIDED EXERCISE

REFACTORING THE TO DO LIST APPLICATION

In this lab, you will refactor the To Do List application into multiple containers that are linked together, allowing the front-end HTML 5 application, the Node.js REST API, and the MySQL server to run in their own containers.

OUTCOMES

You should be able to refactor a monolithic application into its tiers and deploy each tier as a microservice.

BEFORE YOU BEGIN

Run the following command to set up the working directories for the lab with the To Do List application files:

```
[student@workstation ~]$ lab appendix-microservices start
```

► 1. Move the HTML Files

The first step in refactoring the To Do List application is to move the front-end code from the application into its own running container. This step guides you through moving the HTML application and its dependent files into their own directory for deployment to an Apache server running in a container.

- 1.1. Move the HTML and static files to the **src/** directory from the monolithic Node.js To Do List application:

```
[student@workstation ~]$ cd ~/D0180/labs/appendix-microservices/apps/html5/  
[student@workstation html5]$ mv \  
> ~/D0180/labs/appendix-microservices/apps/nodejs/todo/* \  
> ~/D0180/labs/appendix-microservices/apps/html5/src/
```

- 1.2. The current front-end application interacts with the API service using a relative URL. Because the API and front-end code will now run in separate containers, the front-end needs to be adjusted to point to the absolute URL of the To Do List application API.

Open the **/home/student/D0180/labs/appendix-microservices/apps/html5/src/script/item.js** file. At the bottom of the file, look for the following method:

```
app.factory('itemService', function ($resource) {  
    return $resource('api/items/:id');  
});
```

Replace that code with the following content:

```
app.factory('itemService', function ($resource) {
    return $resource('http://workstation.lab.example.com:30080/todo/api/
items/:id');
});
```

Make sure there are no line breaks in the new URL, save the file, and exit the editor.

► 2. Build the HTML Image

- 2.1. Run the build script to build the Apache parent image.

```
[student@workstation html5]$ cd ~/D0180/labs/appendix-microservices/images/apache
[student@workstation apache]$ ./build.sh
STEP 1: FROM rhel7:7.5
...output omitted...
STEP 13: COMMIT do180/httpd
```

- 2.2. Verify that the image is built correctly:

```
[student@workstation apache]$ sudo podman images
REPOSITORY          TAG      IMAGE ID      CREATED        SIZE
localhost/do180/httpd  latest   34376f2a318f  2 minutes ago  282.6 MB
...
```

- 2.3. Build the child Apache image:

```
[student@workstation apache]$ cd ~/D0180/labs/appendix-microservices/deploy/html5
[student@workstation html5]$ ./build.sh
STEP 1: FROM do180/httpd
STEP 2: COPY ./src/ ${HOME}/
--> cf11fb78e09cc7460d094384d3674a9eb27c4fa2d9565243a7d549f89913dde1
STEP 3: COMMIT do180/todo_frontend
```

- 2.4. Verify that the image is built correctly:

```
[student@workstation html5]$ sudo podman images
REPOSITORY          TAG      IMAGE ID      CREATED        SIZE
localhost/do180/todo_frontend  latest   30b3fc531bc6  2 minutes ago  286.9 MB
localhost/do180/httpd        latest   34376f2a318f  4 minutes ago  282.6 MB
...
```

► 3. Modify the REST API to Connect to External Containers

- 3.1. The REST API currently uses hard-coded values to connect to the MySQL database. Edit the **/home/student/D0180/labs/appendix-microservices/apps/nodejs/models/db.js** file, which holds the database configuration. Update the `dbname`, `username`, and `password` values to use environment variables instead. Also, update the `params.host` to point to the host name of the host running the

MySQL container and update the `params.port` to reflect the redirected port to the container. Replaced contents should look like this:

```
module.exports.params = {  
    dbname: process.env.MYSQL_DATABASE,  
    username: process.env.MYSQL_USER,  
    password: process.env.MYSQL_PASSWORD,  
    params: {  
        host: "workstation.lab.example.com",  
        port: "30306",  
        dialect: 'mysql'  
    }  
};
```



NOTE

This file can be copied and pasted from `/home/student/D0180/solutions/appendix-microservices/apps/nodejs/models/db.js`.

- 3.2. Configure the back end to handle *Cross-origin resource sharing (CORS)*. This occurs when a resource request is made from a different domain from the one in which the request was made. Because the API needs to handle requests from a different DNS domain (the front-end application), it is necessary to create security exceptions to allow these requests to succeed. Make the following modifications to the application in the language of your preference in order to handle CORS.

Add the line `.use(restify.CORS());` to the `server` variable for the default CORS settings to allow requests from any origin in the `app.js` file located at `/home/student/D0180/labs/appendix-microservices/apps/nodejs/app.js`.

Remove the semicolon from the line that reads `.use(restify.bodyParser())` and append it to the line that allows CORS.

```
var server = restify.createServer()  
.use(restify.fullResponse())  
.use(restify.queryParser())  
.use(restify.bodyParser())  
.use(restify.CORS());
```

► 4. Build the REST API Image

- 4.1. Build the REST API child image using the following command. This image uses the Node.js image.

```
[student@workstation html5]$ cd ~/D0180/labs/appendix-microservices/deploy/nodejs  
[student@workstation nodejs]$ ./build.sh  
STEP 1: FROM rhscl/nodejs-4-rhel7:latest  
...output omitted...
```

STEP 11: COMMIT do180/todonodejs

- 4.2. Run the **podman images** command to verify that all of the required images are built successfully:

```
[student@workstation nodejs]$ sudo podman images
REPOSITORY           TAG      IMAGE ID   CREATED     SIZE
localhost/do180/httpd    latest   2963ca81ac51  5 seconds ago  249 MB
localhost/do180/todonodejs  latest   7b64ef105c50  7 minutes ago  533 MB
localhost/do180/todo_frontend  latest   53ad57d2306c  9 minutes ago  254 MB
...output omitted...
```

▶ 5. Run the Containers

- 5.1. Use the **run.sh** script to run the containers:

```
[student@workstation nodejs]$ cd linked/
[student@workstation linked]$ ./run.sh
• Creating database volume: OK
• Launching database: OK
• Importing database: OK
• Launching To Do application: OK
```

- 5.2. Run the **podman ps** command to confirm that all three containers are running:

```
[student@workstation linked]$ sudo podman ps
... IMAGE                  ... PORTS          NAMES
... localhost/do180/todo_frontend ... 0.0.0.0:30000->80/tcp    todo_frontend
... localhost/do180/todonodejs    ... 8080/tcp, 0.0.0.0:30080... todoapi
... localhost/rhscl/mysql-57-rhel7 ... 0.0.0.0:30306->3306/tcp    mysql
```

▶ 6. Test the Application

- 6.1. Use the **curl** command to verify that the REST API for the To Do List application is working correctly:

```
[student@workstation linked]$ curl -w "\n" 127.0.0.1:30080/todo/api/items/1
{"description": "Pick up newspaper", "done": false, "id":1}
```

- 6.2. Open Firefox on workstation and navigate to `http://127.0.0.1:30000`, where you should see the To Do List application.

Finish

On workstation, run the **lab appendix-microservices finish** script to complete this lab.

```
[student@workstation ~]$ lab appendix-microservices finish
```

This concludes the guided exercise.

SUMMARY

In this chapter, you learned:

- Breaking a monolithic application into multiple containers allows for greater application scalability, makes upgrades easier, and allows higher hardware utilization.
- The three common tiers for logical division of an application are the presentation tier, the business tier, and the persistence tier.
- *Cross-Origin Resource Sharing (CORS)* can prevent Ajax calls to servers different from the one where the pages were downloaded. Be sure to make provisions to allow CORS from other containers in the application.
- Container images are intended to be immutable, but configurations can be passed in either at image build time or by creating persistent storage for configurations.