

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 3.9 DO180

INTRODUCTION TO CONTAINERS, KUBERNETES, AND RED HAT OPENSHIFT

Edition 1



INTRODUCTION TO CONTAINERS, KUBERNETES, AND RED HAT OPENSHIFT



OCP 3.9 DO180
Introduction to Containers, Kubernetes, and Red Hat
OpenShift
Edition 1 20180926
Publication date 20180926

Authors: Artur Golowski, Richard Allred, Michael Jarrett, Fernando Lozano,
Razique Mahroua, Saumik Paul, Jim Rigsbee, Ravishankar Srinivasan,
Richard Taniguchi
Editor: David O'Brien, Dave Sacco

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: George Hacker, Philip Sweany, and Rob Locke

Document Conventions	vii
Introduction	ix
Introduction to Containers, Kubernetes, and Red Hat OpenShift	ix
Orientation to the Classroom Environment	x
Internationalization	xiii
1. Getting Started with Container Technology	1
Overview of the Container Architecture	2
Quiz: Overview of the Container Architecture	5
Overview of the Docker Architecture	9
Quiz: Overview of the Docker Architecture	12
Describing Kubernetes and OpenShift	14
Quiz: Describing Kubernetes and OpenShift	18
Summary	22
2. Creating Containerized Services	23
Building a Development Environment	24
Quiz: Building a Development Environment	32
Provisioning a Database Server	34
Guided Exercise: Creating a MySQL Database Instance	43
Lab: Creating Containerized Services	46
Summary	51
3. Managing Containers	53
Managing the Life Cycle of Containers	54
Guided Exercise: Managing a MySQL Container	64
Attaching Docker Persistent Storage	68
Guided Exercise: Persisting a MySQL Database	71
Accessing Docker Networks	74
Guided Exercise: Loading the Database	77
Lab: Managing Containers	80
Summary	88
4. Managing Container Images	89
Accessing Registries	90
Quiz: Working With Registries	96
Manipulating Container Images	100
Guided Exercise: Creating a Custom Apache Container Image	105
Lab: Managing Images	111
Summary	117
5. Creating Custom Container Images	119
Design Considerations for Custom Container Images	120
Quiz: Approaches to Container Image Design	124
Building Custom Container Images with Dockerfile	126
Guided Exercise: Creating a Basic Apache Container Image	135
Lab: Creating Custom Container Images	139
Summary	147
6. Deploying Containerized Applications on OpenShift	149
Installing the OpenShift Command-line Tool	150
Quiz: OpenShift CLI	153
Creating Kubernetes Resources	155
Guided Exercise: Deploying a Database Server on OpenShift	168
Creating Applications with Source-to-Image	173
Guided Exercise: Creating a Containerized Application with Source-to-Image	182
Creating Routes	189
Guided Exercise: Exposing a Service as a Route	192
Creating Applications with the OpenShift Web Console	195

Guided Exercise: Creating an Application with the Web Console	197
Lab: Deploying Containerized Applications on OpenShift	204
Summary	207
7. Deploying Multi-Container Applications	209
Considerations for Multi-Container Applications	210
Quiz: Multi-Container Application Considerations	215
Deploying a Multi-Container Application with Docker	217
Guided Exercise: Deploying the Web Application and MySQL Containers	221
Deploying a Multi-Container Application on OpenShift	228
Guided Exercise: Creating an Application with a Template	235
Lab: Deploying Multi-Container Applications	241
Summary	249
8. Troubleshooting Containerized Applications	251
Troubleshooting S2I Builds and Deployments	252
Guided Exercise: Troubleshooting an OpenShift Build	255
Troubleshooting Containerized Applications	262
Guided Exercise: Configuring Apache Container Logs for Debugging	270
Lab: Troubleshooting Containerized Applications	274
Summary	281
9. Comprehensive Review of Introduction to Containers, Kubernetes, and Red Hat OpenShift	283
Comprehensive Review	284
Lab: Containerizing and Deploying a Software Application	287
A. Implementing Microservices Architecture	297
Implementing Microservices Architectures	298
Guided Exercise: Refactoring the To Do List Application	301
Summary	306

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

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 Docker, 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 Docker containers.
- 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 docker 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, **master**, with the host name `master.lab.example.com`, hosts the OpenShift Container Platform cluster.

The fourth and fifth VMs, **node1** and **node2**, with the host names `node1.lab.example.com`, and `node2.lab.example.com` respectively are nodes part of 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>master.lab.example.com</code>	172.25.250.10	OpenShift Container Platform cluster server
<code>node1.lab.example.com</code>	172.25.250.11	OpenShift Container Platform cluster node
<code>node2.lab.example.com</code>	172.25.250.12	OpenShift Container Platform cluster node
<code>services.lab.example.com</code> , <code>registry.lab.example.com</code>	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 **setup**, **grade**, or **reset**.

- The **setup** subcommand is used at the beginning of an exercise. It verifies that the systems are ready for the activity, possibly making some configuration changes to them.
- The **grade** subcommand is executed at the end of an exercise. It provides external confirmation that the activity's requested steps were performed correctly.
- The **reset** subcommand can be used to return the VM to its original state and re-start. This is usually followed by the **lab setup** command.

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.
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	Delete the ROL classroom. This destroys all virtual machines in the classroom. 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	Open a new tab in the browser and connect to the console of the virtual machine. Students can log in directly to the machine and run commands. In most cases, students should log in to the workstation.lab.example.com machine and use ssh to connect to the other virtual machines.
ACTION → Start	Start (“power on”) the machine.
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	Forcefully shut down the machine and reset the disk to its initial state. 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

LANGUAGE SUPPORT

Red Hat Enterprise Linux 7 officially supports 22 languages: English, Assamese, Bengali, Chinese (Simplified), Chinese (Traditional), French, German, Gujarati, Hindi, Italian, Japanese, Kannada, Korean, Malayalam, Marathi, Odia, Portuguese (Brazilian), Punjabi, Russian, Spanish, Tamil, and Telugu.

PER-USER LANGUAGE SELECTION

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

Language Settings

In the GNOME desktop environment, the user may 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. Run the command **gnome-control-center region**, or from the top bar, select (User) → Settings. In the window that opens, select Region & Language. The user can click the Language box and select their preferred language from the list that appears. This will also update the Formats setting to the default for that language. The next time the user logs in, these changes will take full effect.

These settings affect the GNOME desktop environment and any applications, including **gnome-terminal**, started inside it. However, they do not apply to that account if accessed through an **ssh** login from a remote system or a local text console (such as **tty2**).



NOTE

A user can make their shell environment use the same **LANG** setting as their graphical environment, even when they log in through a text console or over **ssh**. One way to do this is to place code similar to the following in the user's **~/.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=$(grep 'Language=' /var/lib/AccountService/users/${USER} \
    | sed 's/Language=//')
if [ "$i" != "" ]; then
    export LANG=$i
fi
```

Japanese, Korean, Chinese, or other languages with a non-Latin character set may not display properly on local text 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 24 17:55:01 CDT 2014
```

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

Input Method Settings

GNOME 3 in Red Hat Enterprise Linux 7 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's 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.

Once 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 find also 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 the user knows the character's Unicode code point, by typing **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, *root* can change the system-wide locale settings with the **localectl** command. If **localectl** is run with no arguments, it will display the current system-wide locale settings.

To set the system-wide language, run the command **localectl set-locale LANG=locale**, where *locale* is the appropriate **\$LANG** 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 and clicking the Login Screen button at the upper-right corner of the window. Changing the Language of the login screen will also adjust the system-wide default language setting stored in the `/etc/locale.conf` configuration file.



IMPORTANT

Local text consoles such as `tty2` are more limited in the fonts that they can display than `gnome-terminal` and `ssh` sessions. For example, Japanese, Korean, and Chinese characters may not display as expected on a local text console. For this reason, it may make sense to use English or another language with a Latin character set for the system's text console.

Likewise, local text 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 local text virtual consoles and the X11 graphical environment. See the `localectl(1)`, `kbd(4)`, and `vconsole.conf(5)` man pages for more information.

LANGUAGE PACKS

When using non-English languages, you may want to install additional "language packs" to provide additional translations, dictionaries, and so forth. To view the list of available langpacks, run `yum langavailable`. To view the list of langpacks currently installed on the system, run `yum langlist`. To add an additional langpack to the system, run `yum langinstall code`, where `code` is the code in square brackets after the language name in the output of `yum langavailable`.



REFERENCES

`locale(7)`, `localectl(1)`, `kbd(4)`, `locale.conf(5)`, `vconsole.conf(5)`, `unicode(7)`, `utf-8(7)`, and `yum-langpacks(8)` 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

Language Codes

LANGUAGE	\$LANG VALUE
English (US)	en_US.utf8
Assamese	as_IN.utf8
Bengali	bn_IN.utf8
Chinese (Simplified)	zh_CN.utf8
Chinese (Traditional)	zh_TW.utf8
French	fr_FR.utf8

LANGUAGE	\$LANG VALUE
German	de_DE.utf8
Gujarati	gu_IN.utf8
Hindi	hi_IN.utf8
Italian	it_IT.utf8
Japanese	ja_JP.utf8
Kannada	kn_IN.utf8
Korean	ko_KR.utf8
Malayalam	ml_IN.utf8
Marathi	mr_IN.utf8
Odia	or_IN.utf8
Portuguese (Brazilian)	pt_BR.utf8
Punjabi	pa_IN.utf8
Russian	ru_RU.utf8
Spanish	es_ES.utf8
Tamil	ta_IN.utf8
Telugu	te_IN.utf8

CHAPTER 1

GETTING STARTED WITH CONTAINER TECHNOLOGY

GOAL

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

OBJECTIVES

- Describe the architecture of Linux containers.
 - Describe how containers are implemented using Docker.
 - Describe the architecture of a Kubernetes cluster running on the Red Hat OpenShift Container Platform.
-
- Container Architecture (and Quiz)
 - Docker Architecture (and Quiz)
 - Container Orchestration with Kubernetes and OpenShift (and Quiz)

SECTIONS

OVERVIEW OF THE CONTAINER ARCHITECTURE

OBJECTIVES

After completing this section, students should be able to:

- Describe the architecture of Linux containers.
- Describe the characteristics of software applications.
- List the approaches of using a container

CONTAINERIZED APPLICATIONS

Software applications are typically deployed as a single set of libraries and configuration files to a runtime environment. They are traditionally deployed to an operating system with a set of services running, such as a database server or an HTTP server, but they can also be deployed to any environment that can provide the same services, such as a virtual machine or a physical host.

The major drawback to using a software application is that it is entangled with the runtime environment and any updates or patches applied to the base OS might break the application. For example, an OS update might include multiple dependency updates, including libraries (that is, operating system libraries shared by multiple programming languages) that might affect the running application with incompatible updates.

Moreover, if another application is sharing the same host OS and the same set of libraries, as described in the Figure 1.1, there might be a risk of preventing the application from properly running it if an update that fixes the first application libraries affects the second application.

Therefore, for a company developing typical software applications, any maintenance on the running environment might require a full set of tests to guarantee that any OS update does not affect the application as well.

Depending on the complexity of an application, the regression verification might not be an easy task and might require a major project. Furthermore, any update normally requires a full application stop. Normally, this implies an environment with high-availability features enabled to minimize the impact of any downtime, and increases the complexity of the deployment process. The maintenance might become cumbersome, and any deployment or update might become a complex process. The Figure 1.1 describes the difference between applications running as containers and applications running on the host operating system.

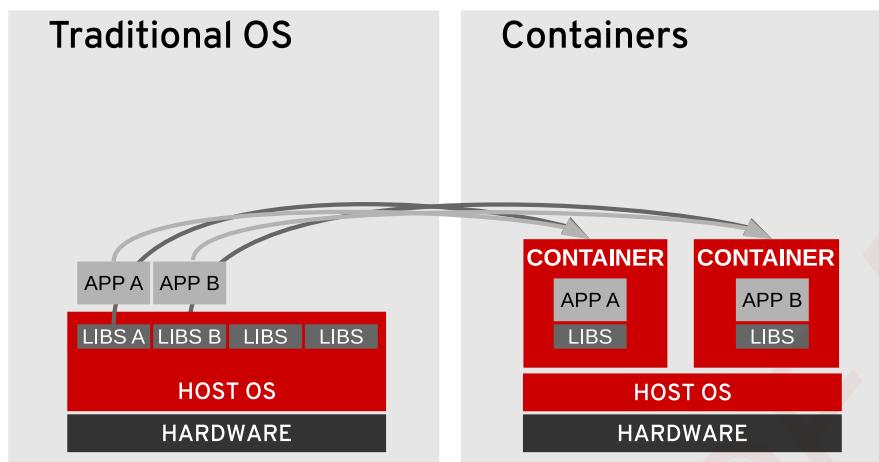


Figure 1.1: Container versus operating system differences

Alternatively, a system administrator can work with *containers*, which are a kind of isolated partition inside a single operating system. Containers provide many of the same benefits as virtual machines, such as security, storage, and network isolation, while requiring far fewer hardware resources and being quicker to launch and terminate. They also isolate the libraries and the runtime environment (such as CPU and storage) used by an application to minimize the impact of any OS update to the host OS, as described in Figure 1.1.

The use of containers helps not only with the efficiency, elasticity, and reusability of the hosted applications, but also with portability of the platform and applications. There are many container providers available, such as Rocket, Drawbridge, and LXC, but one of the major providers is Docker.

Some of the major advantages of containers are listed below.

Low hardware footprint

Uses 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 provided by containers.

Environment isolation

Works 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, which might not work with the update.

Quick deployment

Deploys any container 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 only requires a restart without stopping any services on the host OS.

Multiple environment deployment

In a traditional deployment scenario using a single host, any environment differences might potentially break the application. Using containers, however, the differences and incompatibilities are mitigated because the same container image is used.

Reusability

The same container can be reused by multiple applications without the need to set up a full OS. A database container can be used to create a set of tables for a software application, and

it can be quickly destroyed and recreated without the need to run a set of housekeeping tasks. Additionally, the same database container can be used by the production environment to deploy an application.

Often, a software application with all its dependent services (databases, messaging, file systems) are made to run in a single container. However, container characteristics and agility requirements might make this approach challenging or ill-advised. In these instances, a multi-container deployment may be more suitable. Additionally, be aware that some application actions may not be suited for a containerized environment. For example, applications accessing low-level hardware information, such as memory, file-systems and devices may fail due to container constraints.

Finally, containers boost the microservices development approach because they provide a lightweight and reliable environment to create and run services that can be deployed to a production or development environment without the complexity of a multiple machine environment.

► QUIZ

OVERVIEW OF THE CONTAINER ARCHITECTURE

Choose the correct answers to the following questions:

► 1. Which two options are examples of software applications that might run in a container? (Select 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 of the two following use cases are better suited for containers? (Select 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. A data center is looking for alternatives to shared hosting for database applications to minimize the amount of hardware processing needed.
- 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? (Select 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? (Select 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 THE CONTAINER ARCHITECTURE

Choose the correct answers to the following questions:

► 1. Which two options are examples of software applications that might run in a container? (Select 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 of the two following use cases are better suited for containers? (Select 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. A data center is looking for alternatives to shared hosting for database applications to minimize the amount of hardware processing needed.
- 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? (Select 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? (Select 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 THE DOCKER ARCHITECTURE

OBJECTIVES

After completing this section, students should be able to:

- Describe how containers are implemented using Docker.
- List the key components of the Docker architecture.
- Describe the architecture behind the Docker command-line interface (CLI).

DESCRIBING THE DOCKER ARCHITECTURE

Docker is one of the container implementations available for deployment and supported by companies such as Red Hat in their Red Hat Enterprise Linux Atomic Host platform. Docker Hub provides a large set of containers developed by the community.

Docker uses a client-server architecture, as described below:

Client

The command-line tool (**docker**) is responsible for communicating with a server using a RESTful API to request operations.

Server

This service, which runs as a **daemon** on an operating system, does the heavy lifting of building, running, and downloading container images.

The daemon can run either on the same system as the **docker** client or remotely.



NOTE

For this course, both the client and the server will be running on the **workstation** machine.



NOTE

In a Red Hat Enterprise Linux environment, the daemon is represented by a **systemd** unit called **docker.service**.

DOCKER CORE ELEMENTS

Docker depends on three major elements:

Images

Images are read-only templates that contain a runtime environment that includes application libraries and applications. Images are used to create containers. Images can be created, updated, or downloaded for immediate consumption.

Registries

Registries store images for public or private use. The well-known public registry is Docker Hub, and it stores multiple images developed by the community, but private registries can be

created to support internal image development under a company's discretion. This course runs on a private registry in a virtual machine where all the required images are stored for faster consumption.

Containers

Containers are segregated user-space environments for running applications isolated from other applications sharing the same host OS.



REFERENCES

Docker Hub website

<https://hub.docker.com>



NOTE

In a RHEL environment, the registry is represented by a **systemd** unit called **docker-registry.service**.

CONTAINERS AND THE LINUX KERNEL

Containers created by Docker, from Docker-formatted container images, are isolated from each other by several standard features of the Linux kernel. These include:

Namespaces

The kernel can place specific system resources that are normally visible to all processes into a namespace. Inside a namespace, only processes that are members of that namespace can see those resources. Resources that can be placed into a namespace include network interfaces, the process ID list, mount points, IPC resources, and the system's own hostname information. As an example, two processes in two different mounted namespaces have different views of what the mounted root file system is. Each container is added to a specific set of namespaces, which are only used by that container.

Control groups (cgroups)

Control groups partition sets of processes and their children into groups in order to manage and limit the resources they consume. Control groups place restrictions on the amount of system resources the processes belonging to a specific container might use. This keeps one container from using too many resources on the container host.

SELinux

SELinux is a mandatory access control system that is used to protect containers from each other and to protect the container host from its own running containers. Standard SELinux type enforcement is used to protect the host system from running containers. Container processes run as a confined SELinux type that has limited access to host system resources. In addition, sVirt uses SELinux *Multi-Category Security (MCS)* to protect containers from each other. Each container's processes are placed in a unique category to isolate them from each other.

DOCKER CONTAINER IMAGES

Each image in Docker consists of a series of layers that are combined into what is seen by the containerized applications a single virtual file system. Docker images are immutable; any extra layer added over the preexisting layers overrides their contents without changing them directly. Therefore, any change made to a container image is destroyed unless a new image is generated using the existing extra layer. The UnionFS file system provides containers with a single file system view of the multiple image layers.



REFERENCES

UnionFS wiki page

<https://en.wikipedia.org/wiki/UnionFS>

In summary, there are two approaches to create a new image:

- **Using a running container:** An immutable image is used to start a new container instance and any changes or updates needed by this container are made to a read/ write extra layer. **Docker** commands can be issued to store that read/write layer over the existing image to generate a new image. Due to its simplicity, this approach is the easiest way to create images, but it is not a recommended approach because the image size might become large due to unnecessary files, such as temporary files and logs.
- **Using a Dockerfile:** Alternatively, container images can be built from a base image using a set of steps called *instructions*. Each instruction creates a new layer on the image that is used to build the final container image. This is the suggested approach to building images, because it controls which files are added to each layer.

► QUIZ

OVERVIEW OF THE DOCKER ARCHITECTURE

Choose the correct answers to the following questions:

- ▶ **1. Which of the following three tasks are managed by a component other than the Docker client? (Select three.)**
 - a. Downloading container image files from a registry.
 - b. Requesting a container image deployment from a server.
 - c. Searching for images from a registry.
 - d. Building a container image.

- ▶ **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 two kernel components does Docker use to create and manage the runtime environment for any container? (Choose two.)**
 - a. Namespaces
 - b. iSCSI
 - c. Control groups
 - d. LVM
 - e. NUMA support

- ▶ **4. An existing image of a WordPress blog was updated on a developer's machine to include new homemade extensions. Which is the best approach to create a new image with those updates provided by the developer? (Select one.)**
 - a. The updates made to the developer's custom WordPress should be copied and transferred to the production WordPress, and all the patches should be made within the image.
 - b. The updates made to the developer's custom WordPress should be assembled as a new image using a Dockerfile to rebuild the container image.
 - c. A **diff** should be executed on the production and the developer's WordPress image, and all the binary differences should be applied to the production image.
 - d. Copy the updated files from the developer's image to the **/tmp** directory from the production environment and request an image update.

► SOLUTION

OVERVIEW OF THE DOCKER ARCHITECTURE

Choose the correct answers to the following questions:

- ▶ **1. Which of the following three tasks are managed by a component other than the Docker client? (Select three.)**
 - a. Downloading container image files from a registry.
 - b. Requesting a container image deployment from a server.
 - c. Searching for images from a registry.
 - d. Building a container image.

- ▶ **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 two kernel components does Docker use to create and manage the runtime environment for any container? (Choose two.)**
 - a. Namespaces
 - b. iSCSI
 - c. Control groups
 - d. LVM
 - e. NUMA support

- ▶ **4. An existing image of a WordPress blog was updated on a developer's machine to include new homemade extensions. Which is the best approach to create a new image with those updates provided by the developer? (Select one.)**
 - a. The updates made to the developer's custom WordPress should be copied and transferred to the production WordPress, and all the patches should be made within the image.
 - b. The updates made to the developer's custom WordPress should be assembled as a new image using a Dockerfile to rebuild the container image.
 - c. A **diff** should be executed on the production and the developer's WordPress image, and all the binary differences should be applied to the production image.
 - d. Copy the updated files from the developer's image to the **/tmp** directory from the production environment and request an image update.

DESCRIBING KUBERNETES AND OPENSHIFT

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 (OCP).
- List the main resource types provided by Kubernetes and OCP.
- Identify the network characteristics of Docker, Kubernetes, and OCP.
- List mechanisms to make a pod externally available.

OPENSIFT TERMINOLOGY

Red Hat OpenShift Container Platform (OCP) is a set of modular components and services built on top of Red Hat Enterprise Linux and Docker. OCP adds PaaS capabilities such as remote management, multitenancy, increased security, monitoring and auditing, application life-cycle management, and self-service interfaces for developers.

Throughout this course, the terms OCP and OpenShift are used to refer to the Red Hat OpenShift Container Platform. The Figure 1.2 illustrates the OpenShift Container Platform stack.

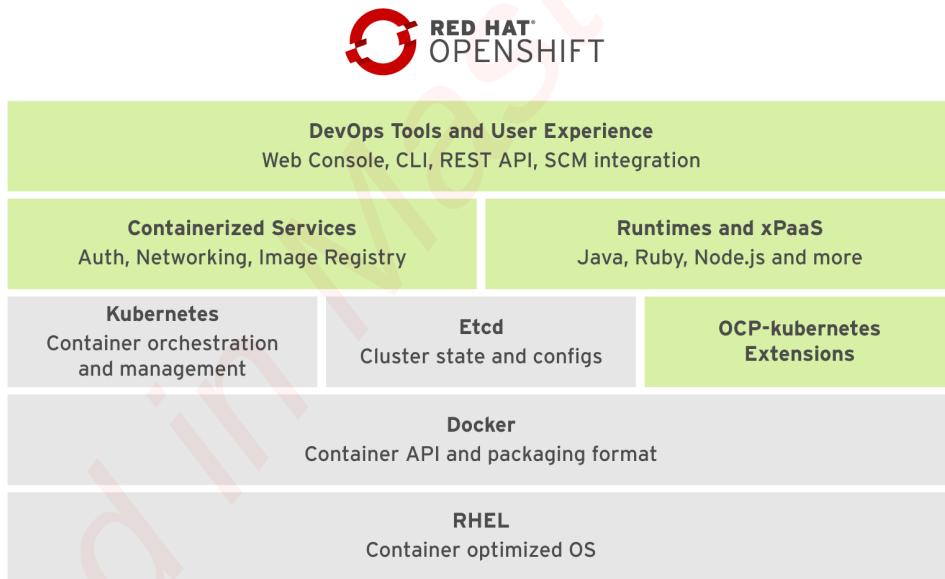


Figure 1.2: OpenShift architecture

In the figure, going from bottom to top, and from left to right, the basic container infrastructure is shown, integrated and enhanced by Red Hat:

- The base OS is Red Hat Enterprise Linux (RHEL).
- Docker provides the basic container management API and the container image file format.
- 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. If Docker is the &core& of OCP, Kubernetes is the &heart& that orchestrates the core.

- *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.

OpenShift adds the capabilities required to provide a production PaaS platform to the Docker and Kubernetes container infrastructure. Continuing from bottom to top and from left to right:

- OCP-Kubernetes extensions are additional resource types stored in Etcd and managed by Kubernetes. These additional resource types form the OCP internal state and configuration.
- Containerized services fulfill many PaaS infrastructure functions, such as networking and authorization. OCP leverages the basic container infrastructure from Docker and Kubernetes for most internal functions. That is, most OCP 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 JBoss middleware products such as JBoss EAP and ActiveMQ.
- DevOps tools and user experience: OCP provides Web and CLI management tools for managing user applications and OCP services. The OpenShift Web and CLI tools are built from REST APIs which can be used by external tools such as IDEs and CI platforms.

A Kubernetes cluster is a set 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
Master	A server that manages the workload and communications in a Kubernetes cluster.
Node	A server that host applications in a Kubernetes cluster.
Label	A key/value pair that can be assigned to any Kubernetes resource. A selector uses labels to filter eligible resources for scheduling and other operations.

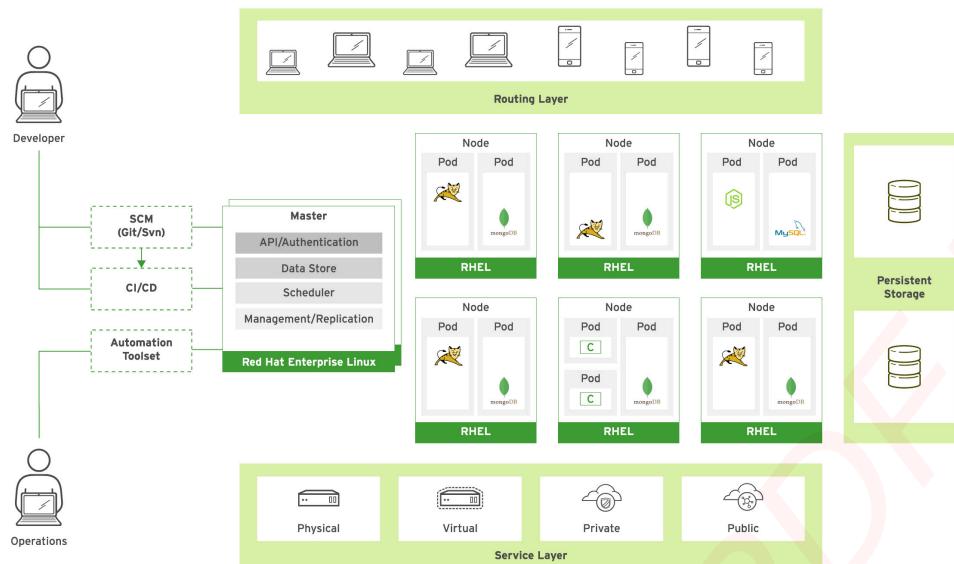


Figure 1.3: OpenShift and Kubernetes architecture

An OpenShift cluster is a Kubernetes cluster which can be managed the same way, but using the management tools provided OpenShift, such as the command-line interface or the web console. This allows for more productive workflows and makes common tasks much easier.

KUBERNETES RESOURCE TYPES

Kubernetes has five main resource types that can be created and configured using a YAML or a JSON file, or using OpenShift management tools:

Pods

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

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

A framework for defining pods that are meant to be horizontally scaled. A replication controller includes a pod definition that is to be replicated, and the pods created from it can be scheduled to different nodes.

Persistent Volumes (PV)

Provision persistent networked storage to pods that can be mounted inside a container to store data.

Persistent Volume Claims (PVC)

Represent a request for storage by a pod to Kubernetes.

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 Configurations (dc)

Represent a set of pods created from the same container image, managing workflows such as rolling updates. A **dc** also provides a basic but extensible continuous delivery workflow.

Build Configurations (bc)

Used by the OpenShift *Source-to-Image (S2I)* feature to build a container image from application source code stored in a Git server. 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.

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 by a **docker** daemon has an IP address assigned from an internal network that is accessible only from the host 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 are not supposed to connect to each other's dynamic IP address directly. It is recommended that they connect to the more stable IP addresses assigned to services, and thus benefit from scalability and fault tolerance.

External access to containers, without OpenShift, requires redirecting a port from the router to the host, then to the internal container IP address, or from the node to a service IP address in the SDN. A Kubernetes service can specify a **NodePort** attribute that is a network port redirected by all the cluster nodes to the SDN. Unfortunately, none of these approaches scale well.

OpenShift makes external access to containers both scalable and simpler, by defining *route* resources. HTTP and TLS accesses to a route are forwarded to service addresses inside the Kubernetes SDN. The only requirement is that the desired DNS host names are mapped to the OCP routers nodes' external IP addresses.



REFERENCES

Docker documentation website
<https://docs.docker.com/>

Kubernetes documentation website
<https://kubernetes.io/docs/>

OpenShift documentation website
<https://docs.openshift.com/>

► QUIZ

DESCRIBING KUBERNETES AND OPENSIFT

Choose the correct answers to the following questions:

► 1. Which three sentences are correct regarding Kubernetes architecture? (Choose three.)

- a. Kubernetes nodes can be managed without a master.
- b. Kubernetes masters manage pod scaling.
- c. Kubernetes masters schedule pods to specific nodes.
- d. A pod is a set of containers managed by Kubernetes as a single unit.
- e. Kubernetes tools cannot be used to manage resources in an OpenShift cluster.

► 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.
- e. Containers created from Kubernetes pods cannot be managed using standard Docker tools.

► 3. Which two statements are true regarding Kubernetes and OpenShift networking? (Select 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. 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 on 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 on a node, subject to a limit managed by the cluster administrator.

► **5. Which statement is correct regarding OpenShift additions over Kubernetes?**

- a. OpenShift adds features required for real-world usage of Kubernetes.
- b. Container images created for OpenShift cannot be used with plain Kubernetes much less with Docker alone.
- c. Red Hat maintains forked versions of Docker and Kubernetes internal to the OCP product.
- d. Doing Continuous Integration and Continuous Deployment with OCP requires external tools.

► SOLUTION

DESCRIBING KUBERNETES AND OPENSHIFT

Choose the correct answers to the following questions:

► 1. Which three sentences are correct regarding Kubernetes architecture? (Choose three.)

- a. Kubernetes nodes can be managed without a master.
- b. Kubernetes masters manage pod scaling.
- c. Kubernetes masters schedule pods to specific nodes.
- d. A pod is a set of containers managed by Kubernetes as a single unit.
- e. Kubernetes tools cannot be used to manage resources in an OpenShift cluster.

► 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.
- e. Containers created from Kubernetes pods cannot be managed using standard Docker tools.

► 3. Which two statements are true regarding Kubernetes and OpenShift networking? (Select 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. 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 on 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 on a node, subject to a limit managed by the cluster administrator.

► **5. Which statement is correct regarding OpenShift additions over Kubernetes?**

- a. OpenShift adds features required for real-world usage of Kubernetes.
- b. Container images created for OpenShift cannot be used with plain Kubernetes much less with Docker alone.
- c. Red Hat maintains forked versions of Docker and Kubernetes internal to the OCP product.
- d. Doing Continuous Integration and Continuous Deployment with OCP requires external tools.

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 its dependencies, making it easier to run the application in different environments.
- Docker creates 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 are key to exposing containerized applications to external users in a manageable way

CHAPTER 2

CREATING CONTAINERIZED SERVICES

GOAL

Provision a server using container technology.

OBJECTIVES

- Describe three container development environment scenarios and build one using OpenShift.
- Create a database server from a container image stored on Docker Hub.

SECTIONS

- Development Environment (and Quiz)
- Provisioning a Database Server (and Guided Exercise)

LAB

- Creating Containerized Services

BUILDING A DEVELOPMENT ENVIRONMENT

OBJECTIVES

After completing this section, students should be able to describe three container development environment scenarios and build one using OpenShift.

CONTAINER DEVELOPMENT ENVIRONMENT SCENARIOS

Container images are usually developed locally on the developer's workstation. After a container image has been tested and accepted, there are many ways to automate building container images. In this course we develop container images using Docker. Containers are deployed and tested on both Docker and Red Hat OpenShift Container Platform.

The following scenarios represent three development environments used to containerize applications:

- Installed OpenShift Cluster
- Red Hat Container Development Kit
- Local OpenShift Cluster

Installed OpenShift Cluster

An installed OpenShift cluster is one in which the OpenShift software is installed using the RPM method. The OpenShift and Kubernetes processes run as services on the operating system. The cluster might be installed by the customer or the customer might use the Red Hat OpenShift Online or Red Hat OpenShift Dedicated environments. All of these types of clusters are accessed remotely using the OpenShift web console or the command-line interface. This method of installing OpenShift is designed for permanent clusters, usually used by many users simultaneously. This installation method is beyond the scope of this course.



NOTE

Red Hat Training offers the *Red Hat OpenShift Administration (DO280)* course providing instruction and hands-on experience installing and configuring Red Hat OpenShift Container Platform.

Local OpenShift Cluster

Anywhere Docker is supported and the OpenShift client can be installed, a local OpenShift cluster can be created using the `oc cluster up` command. In this configuration, OpenShift runs a single-node, single-master cluster in a single container. Internal cluster services, such as the router, run as additional containers. The cluster can be ephemeral or persistent.

Red Hat Container Development Kit

Red Hat Container Development Kit (CDK) version 3 is developed from an upstream project called *Minishift*. The CDK contains a single binary called `minishift`. From this binary a virtual machine disk image can be extracted. This disk image is built using Red Hat Enterprise Linux 7. The `minishift` command is used to create a virtual machine that runs Docker and a local OpenShift cluster. Developers can access the OpenShift cluster with the command-line interface installed by

the **minishift** command. Minishift supports the Linux, MacOS, and Windows operating systems and many types of virtualization technologies, including KVM, VirtualBox, HyperV, and more.

DEVELOPING WITH RED HAT CONTAINER DEVELOPMENT KIT

This section covers creating a development environment using the CDK version 3. The CDK can be downloaded from the Red Hat Developers portal at <http://developers.redhat.com>. Look for it in the technologies section of the web site. The **minishift** binary comes in three different forms: Linux, MacOS, and Windows. Download the appropriate binary for your system.

In addition to the CDK binary, you need to install and configure virtualization software compatible with the operating system. The following is a list of the supported hypervisors, sorted by operating system.

HYPERVISORS SUPPORTED BY MINISHIFT	
OS	HYPERVERISOR
GNU/Linux	KVM (default)
	VirtualBox
MacOS	xhyve (default)
	VirtualBox
	VMware Fusion (supported in upstream Minishift only)
Windows	Hyper-V (default)
	VirtualBox

The following procedure assumes that the CDK is running on GNU/Linux. Prepare the system by enabling **rhel-7-server-rpms**, **rhel-7-server-devtools-rpms**, **rhel-server-rhscl-7-rpms** repositories using **subscription-manager**.

```
[root@foundation0 ~]# subscription-manager repos --enable=rhel-7-server-rpms
[root@foundation0 ~]# subscription-manager repos --enable=rhel-7-server-devtools-rpms
[root@foundation0 ~]# subscription-manager repos --enable=rhel-server-rhscl-7-rpms
```

Import the Red Hat Developer Tools key to your system so yum can verify the integrity of the downloaded packages.

```
[root@foundation0 ~]# rpm --import https://www.redhat.com/security/data/a5787476.txt
```

Once the public key is successfully imported, install **minishift**.

```
[root@foundation0 ~]# yum install cdk-minishift docker-machine-kvm
```

**NOTE**

The **docker-machine-kvm** is the utility that makes **minishift** support the usage of **Linux KVM** as the back-end hypervisor.

Minishift offers Red Hat Enterprise Linux 7 VM and a local OpenShift cluster running inside the VM, as part of the Container Development Kit. Set up the components of the CDK environment with the following command:

```
[kiosk@foundation0 ~]$ minishift setup-cdk
```

This command unpacks the RHEL 7 ISO file and OpenShift command-line, **oc**, in the **~/.minishift/cache** directory. Make sure that **~/.minishift/cache/oc/v3.9.14/linux** is in the **PATH** environment variable of the user intending to use the OpenShift command-line interface. The version number in the path may vary depending on the version of the CDK.

The **minishift config view** command shows all the properties set for the creation of the virtual machine and OpenShift cluster. To update values, use the **minishift config set PROPERTY_NAME PROPERTY_VALUE** command. The most common properties include:

vm-driver

Specifies which virtual machine driver Minishift uses to create, run, and manage the VM in which Docker and OpenShift runs. See the documentation for valid values.

cpus

Specify the number of virtual CPUs the virtual machine uses.

memory

Sets the amount of memory, in MB, that the virtual machine consumes.

More options can be displayed with the **minishift config** command.

To start an OpenShift cluster and create the virtual machine, run the following command:

```
[kiosk@foundation0 ~]$ minishift --username {RH-USERNAME} --password {RH-PASSWORD}
  start
Starting local OpenShift cluster using 'virtualbox' hypervisor...
-- Checking OpenShift client ... OK
-- Checking Docker client ... OK
-- Checking Docker version ... OK
-- Checking for existing OpenShift container ...
-- Checking for openshift/origin:v1.5.0 image ... OK
-- Checking Docker daemon configuration ... OK
-- Checking for available ports ... OK
-- Checking type of volume mount ...
Using Docker shared volumes for OpenShift volumes
-- Creating host directories ... OK
-- Finding server IP ...
Using 192.168.99.100 as the server IP
-- Starting OpenShift container ...
Starting OpenShift using container 'origin'
Waiting for API server to start listening
OpenShift server started
-- Removing temporary directory ... OK
-- Checking container networking ... OK
-- Server Information ...
```

```
OpenShift server started.
The server is accessible via web console at:
https://192.168.99.100:8443

To login as administrator:
oc login -u system:admin
```

The **RH-USERNAME** and **RH-PASSWORD** represent the user's Red Hat subscription credentials that allow Minishift to register the virtual machine to the Red Hat Customer Portal. Use environment variables like **MINISHIFT_USERNAME** and **MINISHIFT_PASSWORD** to set these credentials so you do not have to provide them manually on the command line. These are required parameters and allow packages to be installed on the virtual machine. Developers can sign up for a developer account at <http://developers.redhat.com>.

The **oc** command can now be used to access the OpenShift cluster, using the server address listed in the output of the **minishift start** command. The following list details some of the **minishift** commands.

- Use the following command to stop the cluster.

```
minishift stop
```

- The following command destroys the virtual machine and deletes the development OpenShift cluster permanently.

```
minishift delete
```

- Use the following command to access the virtual machine that runs the cluster.

```
minishift ssh
```

- Use the following command to determine the status of the virtual machine.

```
minishift status
```

- Use the following command to launch and access the web console in the system's default web browser.

```
minishift console
```

DEVELOPING WITH A LOCAL OPENSFIFT CLUSTER

If the developer's workstation can run Docker and the OpenShift command-line interface, a cluster can be created without a virtual machine. The OpenShift client is available for Linux, MacOS, and Windows. The client can be downloaded from the Red Hat Customer Portal. Install the client following the directions for the relevant operating system. Run the **oc version** command to verify the installation.

By default, the Docker daemon supports **Transport Layer Security** (TLS) for implementing an encrypted communication with Docker registry. This encryption method tightens the security of the images being distributed in the Docker registry.

Docker, by nature, looks up the official Docker Hub Registry while pulling any image not in the local cache. However, specific registry lookups can be blocked by modifying **BLOCK_REGISTRY** parameter in **/etc/sysconfig/docker** file.

```
BLOCK_REGISTRY='--block-registry docker.io --block-registry
registry.access.redhat.com'
```

This configuration line causes Docker to disregard the public Docker Hub Registry as well as the Red Hat public registry when pulling container images.

The cluster is started on the local machine using the **oc cluster up** command. For more information, run the **oc cluster up -h** command. It is often desirable to configure a persistent cluster. A persistent cluster is one in which the configuration and runtime data is preserved over a shutdown of the cluster. For a persistent cluster, the following command-line options are recommended:

--use-existing-config

If a configuration already exists, it is reused. Otherwise, a new default configuration is created.

--host-config-dir

Specifies an absolute path for storing/retrieving the cluster configuration files. The default value is **/var/lib/origin**.

--host-data-dir

Specifies where the **etcd** (OpenShift cache) data is written. If the **--host-data-dir** option is not specified, no data is saved and the cluster state is not preserved after shutdown.

--host-volumes-dir

Specifies where Kubernetes volumes are written. The default value is **/var/lib/origin/openshift.local.volumes**.

The version of OpenShift can be controlled by the **--version** option. If you need to retrieve the OpenShift container images from a location other than **registry.access.redhat.com**, the complete image can be specified using the **--image** option. The following is an example of using a custom image location:

```
$ oc cluster up --image='registry.lab.example.com/openshift3/ose'
```

The complete suite of OpenShift containers is pulled from the same registry as specified in this parameter.

To control the router default subdomain, use the **--routing-suffix** option. The host name for the web console can be set using the **--public-hostname** option.

DEMONSTRATION: BUILDING A CONTAINER DEVELOPMENT ENVIRONMENT USING minishift

1. Open a terminal and view the configuration settings of **minishift**.

```
[kiosk@foundation0 ~]$ minishift config view
- iso-url           : file:///home/kiosk/.minishift/cache/iso/minishift-
rhel7.iso
- memory            : 4096
- vm-driver          : kvm
```

2. Set the memory of **minishift** to **6144**.

```
[kiosk@foundation0 ~]$ minishift config set memory 6144
```

- Set the hypervisor for **minishift** to **kvm**.

```
[kiosk@foundation0 ~]$ minishift config set vm-driver kvm
```

**NOTE**

The only hypervisor technology, available in the classroom environment, is KVM, which is the one that the demo uses.

- Set the number of vCPUs for **minishift** to **2**.

```
[kiosk@foundation0 ~]$ minishift config set cpus 2
```

- Display all the enabled and adjusted configuration settings of **minishift** and compare to the output of **step 1**.

```
[kiosk@foundation0 ~]$ minishift config view
- cpus : 2
- iso-url : file:///home/kiosk/.minishift/cache/iso/minishift-
rhel7.iso
- memory : 6144
- vm-driver : kvm
```

- Start the OpenShift cluster with the **minishift** command.

```
[kiosk@foundation0 ~]$ minishift start
-- Starting profile 'minishift'
...output omitted...
Starting OpenShift using registry.access.redhat.com/OpenShift3/ose:v3.9.14 ...
OpenShift server started.

The server is accessible via web console at:
https://192.168.42.150:8443

You are logged in as:
User: developer
Password: <any value>

To login as administrator:
oc login -u system:admin
...output omitted...
```

- View the web console URL using the **minishift** command. Use this URL to access the web interface of **OpenShift**.

```
[kiosk@foundation0 ~]$ minishift console --url
https://192.168.42.150:8443
```

- Optionally, access the OpenShift web interface directly from shell using the **minishift** command.

```
[kiosk@foundation0 ~]$ minishift console
```

This command opens the web console of **OpenShift** in the default web browser of your system. While passing the login credentials in the web console, use *developer* as the username and password.

- Set the path of the **OpenShift** client binary, **oc**, to interact with the **OpenShift** cluster deployed by **minishift** in earlier steps.

```
[kiosk@foundation0 ~]$ minishift oc-env
export PATH="/home/kiosk/.minishift/cache/oc/v3.9.14/linux:$PATH"
# Run this command to configure your shell:
# eval $(minishift oc-env)
[kiosk@foundation0 ~]$ eval $(minishift oc-env)
```

- Using the **oc** command, log in to the OpenShift environment with the *developer* user's credentials.

```
[kiosk@foundation0 ~]$ oc login -u developer
Logged into "https://192.168.42.227:8443" as "developer" using existing
credentials.

You have one project on this server: "myproject"

Using project "myproject".
```

- Create a new project.

```
[kiosk@foundation0 ~]$ oc new-project demo-project
Now using project "demo-project" on server "https://192.168.42.227:8443".

You can add applications to this project with the 'new-app' command. For example,
try:

  oc new-app centos/ruby-22-centos7~https://github.com/openshift/ruby-ex.git

to build a new example application in Ruby.
```

- Create a new application from this URL: <https://github.com/openshift/nodejs-ex>. The *demo-project* project contains this new application.

```
[kiosk@foundation0 ~]$ oc new-app https://github.com/openshift/nodejs-ex
--> Found image 605d09f (3 days old) in image stream "openshift/nodejs" under tag
"8" for "nodejs"
...output omitted...
--> Success
  Build scheduled, use 'oc logs -f bc/nodejs-ex' 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-ex'
    Run 'oc status' to view your app.
```

- Expose the nodejs-ex service as a route.

```
[kiosk@foundation0 ~]$ oc expose service nodejs-ex  
route "nodejs-ex" exposed
```

14. From the home page of OpenShift web console, navigate to *demo-project*, located under the *My Projects* on the right panel of the page. Click the **Overview** page that appears on the left. View the summary of nodejs-ex application in the **Overview** page.
15. In the same **Overview** page, click on the `http://<application-name>-<project-name>.<some_ip>.nip.io` URL under **Routes - External Traffic** of the **Networking** section to access the nodejs-ex application.



REFERENCES

Minishift Documentation

<https://www.openshift.org/minishift/>

Download CDK 3

<https://developers.redhat.com/products/cdk/download/>

Using Red Hat CDK 3

<https://developers.redhat.com/blog/2017/02/28/using-red-hat-container-development-kit-3-beta/>

► QUIZ

BUILDING A DEVELOPMENT ENVIRONMENT

Choose the correct answers to the following questions:

► 1. Which of the following statements is correct?

- a. A production OpenShift cluster is built using the Container Development Kit.
- b. Setting up the Container Development Kit facilitates a development OpenShift cluster.
- c. Without a Red Hat Enterprise Linux 7 OS platform, it is impossible to run an OpenShift cluster.
- d. An OpenShift cluster runs applications on virtual machines.

► 2. Which of the following minishift commands sets up a user's development environment for OpenShift cluster?

- a. `minishift start`
- b. `minishift setup-cdk`
- c. `minishift profile`
- d. `minishift config`

► 3. Which of the following statements is correct?

- a. The Container Development Kit runs the OpenShift cluster within a virtual machine.
- b. The Container Development Kit runs the OpenShift cluster within a container.
- c. The Container Development Kit runs the OpenShift cluster as a service on the host OS.
- d. The Container Development Kit requires administrative privileges for the user to deploy an OpenShift cluster.

► 4. In a Container Development Kit environment, which of the following minishift commands starts an OpenShift cluster?

- a. `minishift setup-cdk`
- b. `minishift enable`
- c. `minishift config view`
- d. `minishift start`

► 5. Which of the following minishift commands revokes the OpenShift cluster in its entirety?

- a. `minishift stop`
- b. `minishift completion`
- c. `minishift delete`
- d. `minishift oc-env`

► SOLUTION

BUILDING A DEVELOPMENT ENVIRONMENT

Choose the correct answers to the following questions:

► 1. Which of the following statements is correct?

- a. A production OpenShift cluster is built using the Container Development Kit.
- b. Setting up the Container Development Kit facilitates a development OpenShift cluster.
- c. Without a Red Hat Enterprise Linux 7 OS platform, it is impossible to run an OpenShift cluster.
- d. An OpenShift cluster runs applications on virtual machines.

► 2. Which of the following minishift commands sets up a user's development environment for OpenShift cluster?

- a. `minishift start`
- b. `minishift setup-cdk`
- c. `minishift profile`
- d. `minishift config`

► 3. Which of the following statements is correct?

- a. The Container Development Kit runs the OpenShift cluster within a virtual machine.
- b. The Container Development Kit runs the OpenShift cluster within a container.
- c. The Container Development Kit runs the OpenShift cluster as a service on the host OS.
- d. The Container Development Kit requires administrative privileges for the user to deploy an OpenShift cluster.

► 4. In a Container Development Kit environment, which of the following minishift commands starts an OpenShift cluster?

- a. `minishift setup-cdk`
- b. `minishift enable`
- c. `minishift config view`
- d. `minishift start`

► 5. Which of the following minishift commands revokes the OpenShift cluster in its entirety?

- a. `minishift stop`
- b. `minishift completion`
- c. `minishift delete`
- d. `minishift oc-env`

PROVISIONING A DATABASE SERVER

OBJECTIVES

After completing this section, students should be able to:

- Create a database server from a container image stored on Docker Hub.
- Search for containers on the Docker Hub site.
- Start containers using the **docker** command.
- Access containers from the command line.
- Leverage the Red Hat Container Catalog.

FINDING AN IMAGE ON DOCKER HUB

Many container images are available for download from the Docker community website at <https://docker.io>. It is a large repository where developers and administrators can get a number of container images developed by the community and some companies.

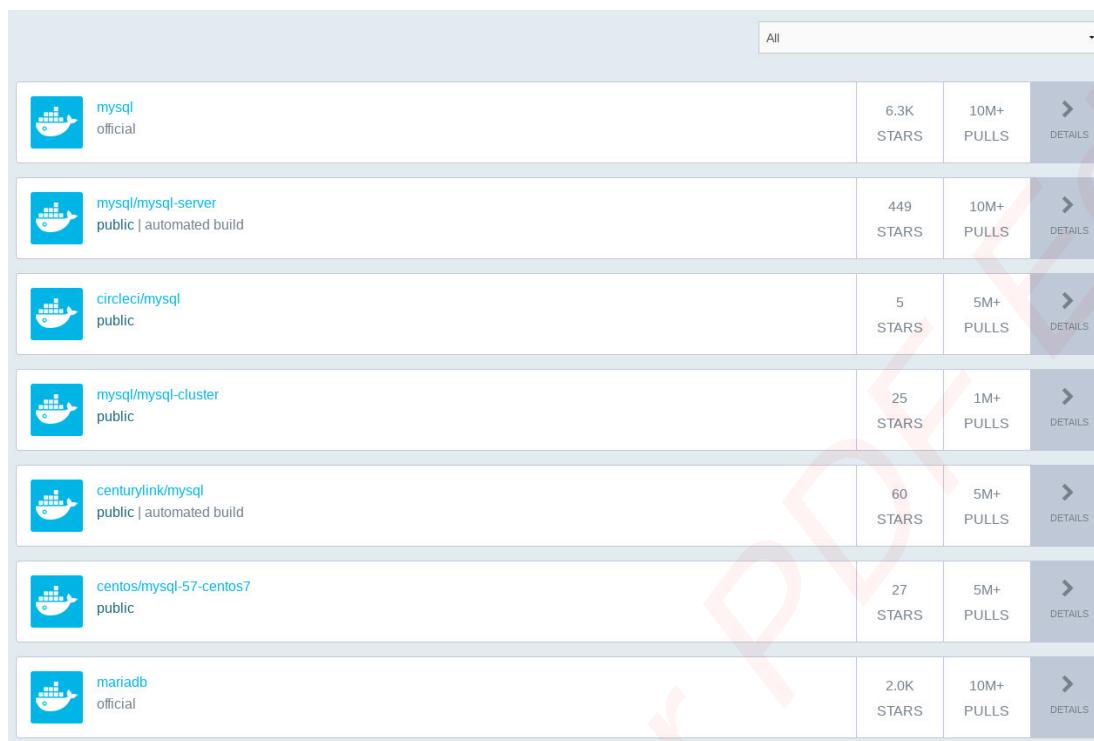
By default, Docker downloads image layers from the Docker Hub image registry. However, images do not provide textual information about themselves, and a search engine tool called Docker Hub was created to look for information about each image and its functionality.



NOTE

Red Hat also provides a private registry with tested and certified container images. By default, RHEL 7 is configured to look for the Red Hat registry in addition to Docker Hub.

The Docker Hub search engine is a simple but effective search engine used to find container images. It looks for a project name and all similar image names from the Docker Hub container image registry. The search results page lists the string used to pull the image files. For example, for the following screen output, the first column is the name of the container image.

**Figure 2.1: Search results page from Docker Hub**

Click an image name to display the container image details page. The details page is not guaranteed to display any particular information, because each author provides different levels of information about their images.

OFFICIAL REPOSITORY

mysql ☆

Last pushed: a month ago

Repo Info Tags

Short Description	Docker Pull Command
MySQL is a widely used, open-source relational database management system (RDBMS).	<code>docker pull mysql</code>

Full Description

Supported tags and respective Dockerfile links

- 8.0.11, 8.0, 8, latest ([8.0/Dockerfile](#))
- 5.7.22, 5.7, 5 ([5.7/Dockerfile](#))
- 5.6.40, 5.6 ([5.6/Dockerfile](#))
- 5.5.60, 5.5 ([5.5/Dockerfile](#))

Quick reference

- Where to get help:**
[the Docker Community Forums](#), [the Docker Community Slack](#), or [Stack Overflow](#)
- Where to file issues:**
<https://github.com/docker-library/mysql/issues>
- Maintained by:**
[the Docker Community](#) and the MySQL Team
- Supported architectures:** ([more info](#))
amd64

- **Published image artifact details:**
[repo-info](#) [repo's](#) [repos/mysql/](#) [directory](#) ([history](#))
 (image metadata, transfer size, etc)
- **Image updates:**
[official-images](#) [PRs with label](#) [library/mysql](#)
[official-images](#) [repo's](#) [library/mysql](#) [file](#) ([history](#))
- **Source of this description:**
[docs](#) [repo's](#) [mysql/](#) [directory](#) ([history](#))
- **Supported Docker versions:**
[the latest release](#) (down to 1.6 on a best-effort basis)

What is MySQL?

MySQL is the world's most popular open source database. With its proven performance, reliability and ease-of-use, MySQL has become the leading database choice for web-based applications, covering the entire range from personal projects and websites, via e-commerce and information services, all the way to high profile web properties including Facebook, Twitter, YouTube, Yahoo! and many more.

Figure 2.3: Detailed information about the image

SEARCHING FROM THE DOCKER CLIENT

The **docker** command can also be used to search for container images:

```
[root@workstation ~]# docker search mysql
```

The **search** verb uses the Docker Hub registry but also any other registries running version 1 of the API.

Running the **docker** command requires special privileges. A development environment usually manages that requirement by assigning the developer to the **docker** group. Without the correct privileges to run the **docker** command, an error message such as the following appears:

```
Got permission denied while trying to connect to the Docker daemon socket at
unix:///var/run/docker.sock: Get http://%2Fvar%2Frun%2Fdocker.sock/v1.26/images/
search?limit=25&term=mysql: dial unix /var/run/docker.sock: connect: permission
denied
```



NOTE

For a production environment, the **docker** command access should be given with the **sudo** command because the **docker** group is vulnerable to privilege escalation attacks.

FETCHING AN IMAGE

To download an image from the Docker Hub container image registry, look for the first column name from the Docker Hub search results page, or the second column from the output of **docker search** command, and use **docker pull** command:

```
[root@workstation ~]# docker pull mysql
```

Many versions of the same image can be provided. Each one receives a different tag name. For example, from the MySQL image details page, there are three different image names available, each one having multiple tag names assigned to them.

If no tag name is provided, then **docker pull** uses the tag called **latest** by default.

To download a specific tag, append the tag name to the image name separated by a colon (:) in the **docker pull** command:

```
[root@workstation ~]# docker pull mysql:5.5
```

LISTING THE IMAGES AVAILABLE IN THE LOCAL CACHE

To list all images that were already downloaded by the local Docker daemon, use the **docker images** command:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
docker.io/mysql	5.5	f13c4be36ec5	3 weeks ago	205 MB
docker.io/mysql	latest	a8a59477268d	3 weeks ago	445 MB

The REPOSITORY column contains the image name as the last path component. A Docker daemon installation comes without any images, so the images list is empty until the system administrator downloads images.

An image name is prefixed with a registry name, which is usually the FQDN name of the registry host, but could be any string. An image name can also include a tag. Thus, the full syntax for an image name, including all optional parameters, is as follows:

`[registry_uri/] [user_name/] image_name[:tag]`

For example: `docker .io/library/mysql:latest`.

Image names from the Docker Hub without a user name actually use the *library* user, so the previous image name is the same as: `docker .io/mysql:latest`. The user *library* is associated with the formally organized images, in official Docker Hub Registry.

Multiple tags can be applied to the same image, as noted previously on MySQL container images. Each tag is listed individually, even though they may represent the same image.

The **docker images** command shows that each image has a unique ID. This allows you to see which image names and tag names refer to the same container image contents. Most **docker** commands that work on images can take either an image name or an image ID as argument.

CREATING A CONTAINER

To create and start a process within a new container, use the **docker run** command. The container is created from the container image name passed as argument.

```
[root@workstation ~]# docker run mysql
```

If the image is not available from the local Docker daemon cache, the **docker run** command tries to pull the image as if a **docker pull** command had been used.

Whatever output the **docker run** command shows is generated by the process inside the container, which is a regular process from the host OS perspective. Killing that process stops the container. In the previous sample output, the container was started with a noninteractive process. Stopping that process with **Ctrl+C (SIGINT)** also stopped the container.

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

```
[root@workstation ~]# docker run -d mysql:5.5
```

Each container has to be assigned a name when created. Docker automatically generates a name if one is not provided. To make container tracking easier, the **--name** option may be passed to the **docker run** command:

```
[root@workstation ~]# docker run --name mysql-container mysql:5.5
```

The container image itself specifies the command to start the process inside the container, but a different one can be specified after the container image name in the **docker run** command:

```
[root@workstation ~]# docker run --name mysql-container -it mysql:5.5 /bin/bash  
[root@f0f46d3239e0:/]#
```

The **-t** and **-i** options are usually needed for interactive text-based programs, so they get allocated a pseudo-terminal, but not for background daemons. The program must exist inside the container image.

Many container images require parameters to be started, such as the MySQL official image. They should be provided using the **-e** option from the **docker** command, and are seen as environment variables by the processes inside the container. The following image is a snapshot from the MySQL official image documentation listing all environment variables recognized by the container image:

Environment Variables

When you start the `mysql` image, you can adjust the configuration of the MySQL instance by passing one or more environment variables on the `docker run` command line. Do note that none of the variables below will have any effect if you start the container with a data directory that already contains a database: any pre-existing database will always be left untouched on container startup.

See also <https://dev.mysql.com/doc/refman/5.7/en/environment-variables.html> for documentation of environment variables which MySQL itself respects (especially variables like `MYSQL_HOST`, which is known to cause issues when used with this image).

`MYSQL_ROOT_PASSWORD`

This variable is mandatory and specifies the password that will be set for the MySQL `root` superuser account. In the above example, it was set to `my-secret-pw`.

`MYSQL_DATABASE`

This variable is optional and allows you to specify the name of a database to be created on image startup. If a user/password was supplied (see below) then that user will be granted superuser access ([corresponding to GRANT ALL](#)) to this database.

`MYSQL_USER`, `MYSQL_PASSWORD`

These variables are optional, used in conjunction to create a new user and to set that user's password. This user will be granted superuser permissions (see above) for the database specified by the `MYSQL_DATABASE` variable. Both variables are required for a user to be created.

Do note that there is no need to use this mechanism to create the root superuser, that user gets created by default with the password specified by the `MYSQL_ROOT_PASSWORD` variable.

`MYSQL_ALLOW_EMPTY_PASSWORD`

This is an optional variable. Set to `yes` to allow the container to be started with a blank password for the root user. *NOTE:* Setting this variable to `yes` is not recommended unless you really know what you are doing, since this will leave your MySQL instance completely unprotected, allowing anyone to gain complete superuser access.

`MYSQL_RANDOM_ROOT_PASSWORD`

This is an optional variable. Set to `yes` to generate a random initial password for the root user (using `pwgen`). The generated root password will be printed to stdout (`GENERATED ROOT PASSWORD:`).

`MYSQL_ONETIME_PASSWORD`

Sets root (not the user specified in `MYSQL_USER`!) user as expired once init is complete, forcing a password change on first login. *NOTE:* This feature is supported on MySQL 5.6+ only. Using this option on MySQL 5.5 will throw an appropriate error during initialization.

Figure 2.5: Environment variables supported by the official MySQL Docker Hub image

To start the MySQL server with different user credentials, pass the following parameters to the `docker run` command:

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

LEVERAGING RED HAT CONTAINER CATALOG

Red Hat maintains its own 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 images that are not tested. Whereas the standard `docker` command works perfectly fine with this repository, there is a web portal called *Red Hat Container Catalog*, or *RHCC*, providing a user-friendly interface to search and explore container images from Red Hat. This web portal is available at <https://access.redhat.com/containers/>.

RHCC 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 among multiple versions of container image on the basis of health index grades. The health index grade indicates how current an image is, and whether the latest security updates have been applied. These grades range from **A** through **F**, with grade **A** being the best.

RHCC 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 screenshots highlight some of the features of *Red Hat Container Catalog*.

The screenshot shows the Red Hat Container Catalog homepage. At the top, there are navigation links for SUBSCRIPTIONS, DOWNLOADS, CONTAINERS, and SUPPORT CASES. The main header includes the Red Hat logo and links for CUSTOMER PORTAL, Products & Services, Tools, Security, and Community. A search bar is located in the top right corner. Below the header, a sidebar on the left lists "Your trust" and "Recently Updated" sections. The "Recently Updated" section highlights the "rhpam-7/rhpam70-businesscentral-openshift" platform. The main content area displays search results for "Apache". The results are categorized into "Products" and "Image Repositories". Under "Products", there are links for "Red Hat JBoss Web Server", "JBoss A-MQ", "ScaleOut StateServer", and "Hazelcast". Under "Image Repositories", there are links for "Apache httpd 2.4- rhel7/httpd-24-rhel7", "Apache 2.4 with PHP 7.0- rhel7/php-70-rhel7", "Apache 2.4 with mod_perl/5.24- rhel7/perl-524-rhel7", and "Apache 2.4 with PHP 7.1- rhel7/php-71-rhel7". Each result item includes a thumbnail, a brief description, the last update time ("an hour ago"), a "HEALTH INDEX" rating (A), and a "View Repository" link.

Figure 2.6: Red Hat Container Catalog home page

As displayed in the screenshot above, typing *Apache* in the search box of *Red Hat 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-rhel7/httpd-24-rhel7* from the suggested list.

Apache httpd 2.4 ☆

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

registry.access.redhat.com/rhscl/httpd-24-rhel7 Updated 6 days ago 2.4-55 : Health Index A

Description

Apache httpd 2.4 available as container, is a powerful, efficient, and extensible web server. Apache supports a variety of features, many implemented as compiled modules which extend the core functionality. These can range from server-side programming language support to authentication schemes. Virtual hosting allows one Apache installation to serve many different Web sites.

Evaluate Image

There is a preview of this image available on OpenShift Online. Deploy it from the Console and try it out.
[Log in to OpenShift Online](#)

Repository Specifications

Registry	registry.access.redhat.com
Namespace/Repository	rhscl/httpd-24-rhel7
Release Category	Generally Available
Application Categories	Web Services

Most recent tag [View All Tags](#)

- Updated 6 days ago 2.4-55
- Health Index A
- Security Signed Unprivileged
- Size 111.9 MB
- RPM Packages 222

Figure 2.7: Apache httpd 2.4 (rhscl/httpd-24-rhel7) image page

Details and several tabs are displayed under the *Apache httpd 2.4* panel. This page states that **Red Hat, Inc.** maintains the image repository. It also indicates that the image repository is associated with Red Hat Enterprise Linux. Under the *Overview* tab, there are other details:

- **Description:** A short summary of what 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 applied to the image, the health of the image, and more.

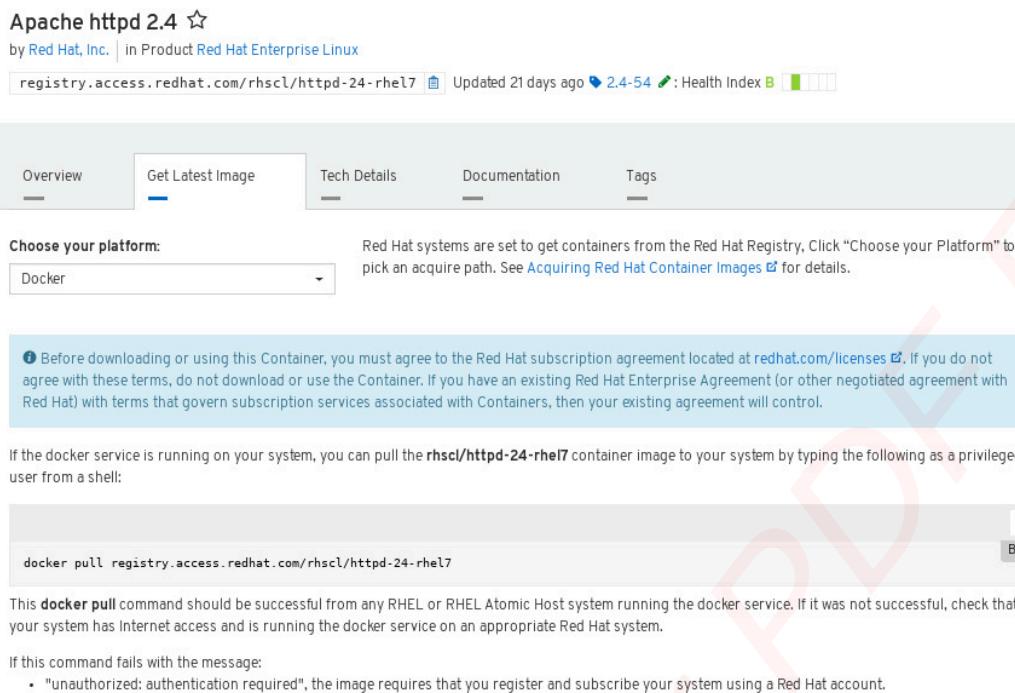


Figure 2.8: Apache httpd 2.4 (rhscl/httpd-24-rhel7) image page

The *Get Latest Image* tab provides the procedure to get the most current version of the image. Specify the intended platform for the image under *Choose your platform* drop-down menu, and the page provides the appropriate command to be executed.



REFERENCES

Docker Hub website

<https://hub.docker.com>

Red Hat Registry website

<https://registry.access.redhat.com>

► 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.

RESOURCES

Files:	N/A
Resources:	Red Hat Container Catalog MySQL 5.7 image (mysql-57-rhel7)

OUTCOMES

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

BEFORE YOU BEGIN

Your workstation must have Docker running. To verify that Docker is running, use the following command in a terminal:

```
[student@workstation ~]$ lab create-basic-mysql setup
```

► 1. Create a MySQL container instance.

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

```
[student@workstation ~]$ docker run --name mysql-basic \
-e MYSQL_USER=user1 -e MYSQL_PASSWORD=mypa55 \
-e MYSQL_DATABASE=items -e MYSQL_ROOT_PASSWORD=r00tpa55 \
-d rhscl/mysql-57-rhel7:5.7-3.14
```

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 ~]$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
NAMES
2ae7af15746d        rhscl/mysql-57-rhel7:5.7-3.14   "container-entrypo..."   4 seconds ago      Up 3 seconds       3306/tcp      mysql-basic
```

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

```
[student@workstation ~]$ docker exec -it mysql-basic bash
```

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

- 3. Add data to the database.

3.1. Log in 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
```

The **mysql** command opens the MySQL database interactive prompt. Run the following command to check the database availability:

```
mysql> show databases;
+-----+
| Database      |
+-----+
| information_schema |
| items          |
| mysql          |
| performance_schema |
| sys            |
+-----+
5 rows in set (0.00 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 (0.29 sec)
```



NOTE

You can optionally use the `~student/D0180/solutions/create-mysql-basic/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             |
+-----+
```

```
+-----+
```

- 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 |
+----+-----+
```

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

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

- ▶ 4. Verify that the database was configured correctly by running the following script:

```
[student@workstation ~]$ lab create-basic-mysql grade
```

- ▶ 5. Revert the modifications made during the lab.

- 5.1. Use the command **docker stop** to stop the container.

```
[student@workstation ~]$ docker stop mysql-basic
```

- 5.2. Remove the data from the stopped container using the **docker rm** command.

```
[student@workstation ~]$ docker rm mysql-basic
```

- 5.3. Use the command **docker rmi** to remove the container image.

```
[student@workstation ~]$ docker rmi rhsc1/mysql-57-rhel7:5.7-3.14
Untagged: rhsc1/mysql-57-rhel7:5.7-3.14
Untagged: registry.lab.example.com/rhsc1/mysql-57-
rhel7@sha256:0a8828385c63d6a7cdb1cf899303357d5cbe500fa1761114256d5966aacce3
Deleted: sha256:4ae3a3f4f409a8912cab9fb71d3564d011ed2e68f926d50f88f2a3a72c809c5
Deleted: sha256:aeecef47e8af04bce5918cf2ecd5bc2e6440c34af30ad36cfb902f8814e9002
Deleted: sha256:238aeee18c5ad3ebcab48db60799cc60607d9f630881a6530dcfe58d117adabf2
Deleted: sha256:d4d40807755515379fa58b2b225b72f7b8e1722ff2bc0ecd39f01f6f8174b3e3
```

This concludes the guided exercise.

▶ LAB

CREATING CONTAINERIZED SERVICES

PERFORMANCE CHECKLIST

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

RESOURCES	
Files:	N/A
Application URL:	http://localhost:8080
Resources:	Official Docker Hub Apache httpd image (httpd)

OUTCOMES

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

Open a terminal on **workstation** as *student* user and run the following command to verify that Docker is running:

```
[student@workstation ~]$ lab httpd-basic setup
```

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 **docker 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.

- 3.1. Start a Bash session inside the container and customize the existing content of **index.html** page.

- 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** page 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.

4. Grade your work.

Run the following command:

```
[student@workstation ~]$ lab httpd-basic grade
```

5. Clean up.

5.1. Stop and remove the **httpd-basic** container.

5.2. Remove the **httpd-basic** container image from the local Docker cache.

This concludes this lab.

► SOLUTION

CREATING CONTAINERIZED SERVICES

PERFORMANCE CHECKLIST

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

RESOURCES	
Files:	N/A
Application URL:	http://localhost:8080
Resources:	Official Docker Hub Apache httpd image (httpd)

OUTCOMES

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

Open a terminal on **workstation** as **student** user and run the following command to verify that Docker is running:

```
[student@workstation ~]$ lab httpd-basic setup
```

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 **docker run** command to forward the port.

Run the following command:

```
[student@workstation ~]$ docker run -d -p 8080:80 \
--name httpd-basic \
httpd:2.4
Unable to find image 'httpd:2.4' locally
Trying to pull repository registry.lab.example.com/httpd ...
2.4: Pulling from registry.lab.example.com/httpd
3d77ce4481b1: Pull complete
73674f4d9403: Pull complete
d266646f40bd: Pull complete
ce7b0dda0c9f: Pull complete
01729050d692: Pull complete
014246127c67: Pull complete
7cd2e04cf570: Pull complete
Digest: sha256:58270ec746bed598ec109aef58d495fca80ee0a89f520bd2430c259ed31ee144
Status: Downloaded newer image for registry.lab.example.com/httpd:2.4
```

```
45ede3ccf26ff43079603cec68bc7a7c9105ac1ce8334052afab48e77ee65a03
```

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.

3.1. Start a Bash session inside the container and customize the existing content of **index.html** page.

Run the following command:

```
[student@workstation ~]$ docker exec -it httpd-basic bash
```

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

```
root@2d2417153059:/usr/local/apache2# ls -la /usr/local/apache2/htdocs
total 4
drwxrwxr-x. 2      1000      1000 24 Jun 15 11:02 .
drwxr-sr-x. 1 www-data www-data 18 Apr 30 04:30 ..
-rw-rw-r--. 1      1000      1000 10 Jun 15 11:02 index.html
```

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

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

```
root@2d2417153059:/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@2d2417153059:/usr/local/apache2# exit
[student@workstation ~]$ curl http://localhost:8080
Hello World
```

4. Grade your work.

Run the following command:

```
[student@workstation ~]$ lab httpd-basic grade
```

5. Clean up.

5.1. Stop and remove the **httpd-basic** container.

```
[student@workstation ~]$ docker stop httpd-basic
```

```
httpd-basic  
[student@workstation ~]$ docker rm httpd-basic  
httpd-basic
```

5.2. Remove the **httpd-basic** container image from the local Docker cache.

```
[student@workstation ~]$ docker rmi httpd:2.4  
Untagged: httpd:2.4  
Untagged: registry.lab.example.com/  
httpd@sha256:58270ec746bed598ec109aef58d495fca80ee0a89f520bd2430c259ed31ee144  
Deleted: sha256:fb2f3851a97186bb0eaf551a40b94782712580c2feac0d15ba925bef2da5fc18  
Deleted: sha256:aed0dee33d7b24c5d60b42e6a9e5f15fd040dbeb9b7f2fbf6ec93f205fc7e5a4  
Deleted: sha256:2f645c8ada1c73b101ceb7729275cebed1fa643a2b056412934a100d1afe615  
Deleted: sha256:838f5356324c57929eb42e6885c0dde76703e9d2c11bf0b6a39afbaa3e6cf443  
Deleted: sha256:8837e56bcf087feeaa70b2b5162365a46a41cbf3bbcfa643f2c3b7e39fe7448d2  
Deleted: sha256:1b2a502dceb0d29a50064d64d96282063e6dd1319de112ab18e69557dcc5c915  
Deleted: sha256:13f2da842fdccaa301d6b853a9644476e6e1fed9150c32bbc4c3f9d5699169a2  
Deleted: sha256:2c833f307fd8f18a378b71d3c43c575fabdb88955a2198662938ac2a08a99928
```

This concludes this lab.

SUMMARY

In this chapter, you learned:

- Red Hat OpenShift Container Platform can be installed from RPM packages, from the client as a container, and in a dedicated virtual machine from Red Hat Container Development Kit (CDK).
- The RPM installation usually configures a cluster of multiple Linux nodes for production, QA, and testing environments.
- The containerized and CDK installations can be performed on a developer workstation, and supports Linux, MacOS, and Windows operating systems.
- The **minishift** command from the CDK unpacks the virtual machine images, installs client tools, and starts the OpenShift cluster inside the virtual machine.
- The **oc cluster up** command from the OpenShift client starts the local all-in-one cluster inside a container. It provides many command-line options to adapt to offline environments.
- Before starting the local all-in-one cluster, the Docker daemon must be configured to allow accessing the local insecure registry.
- The Docker Hub website provides a web interface to search for container images developed by the community and corporations. The Docker client can also search for images in Docker Hub.
- The **docker run** command creates and starts a container from an image that can be pulled by the local Docker daemon.
- Container images might require environment variables that are set using the **-e** option from the **docker run** command.
- *Red Hat Container Catalog* assists in searching, exploring, and analyzing container images from Red Hat's official container image repository.

Created in Master PDF Editor

CHAPTER 3

MANAGING CONTAINERS

GOAL

Manipulate pre-built container images to create and manage containerized services.

OBJECTIVES

- Manage the life cycle of a container from creation to deletion.
- Save application data across container restarts through the use of persistent storage.
- Describe how Docker provides network access to containers, and access a container through port forwarding.

SECTIONS

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

DOCKER CLIENT VERBS

The Docker client, implemented by the `docker` command, provides a set of verbs to create and manage containers. The following figure shows a summary of the most commonly used verbs that change container state.

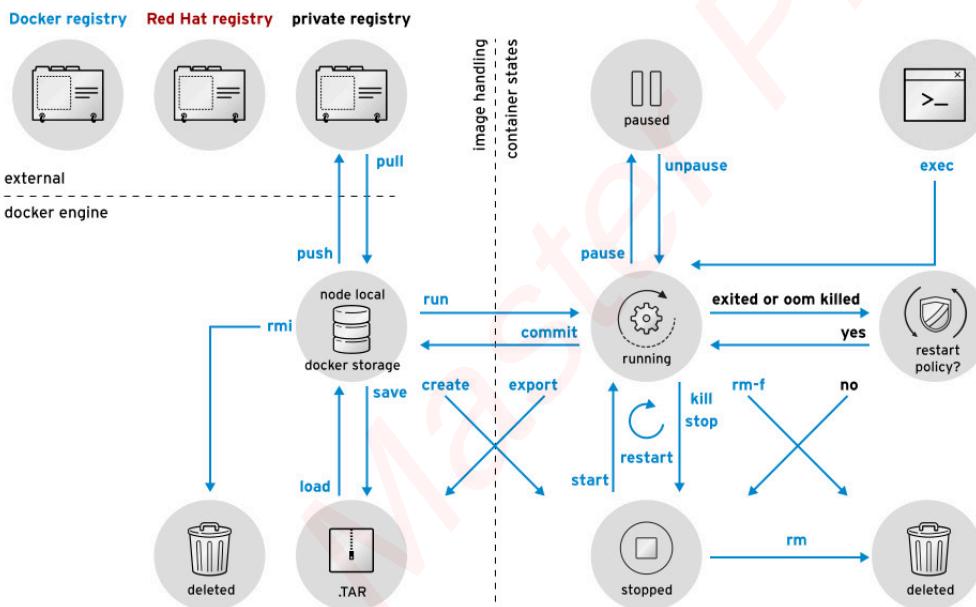


Figure 3.1: Docker client action verbs

The Docker client also provides a set of verbs to obtain information about running and stopped containers. The following figure shows a summary of the most commonly used verbs that query information related to Docker containers.

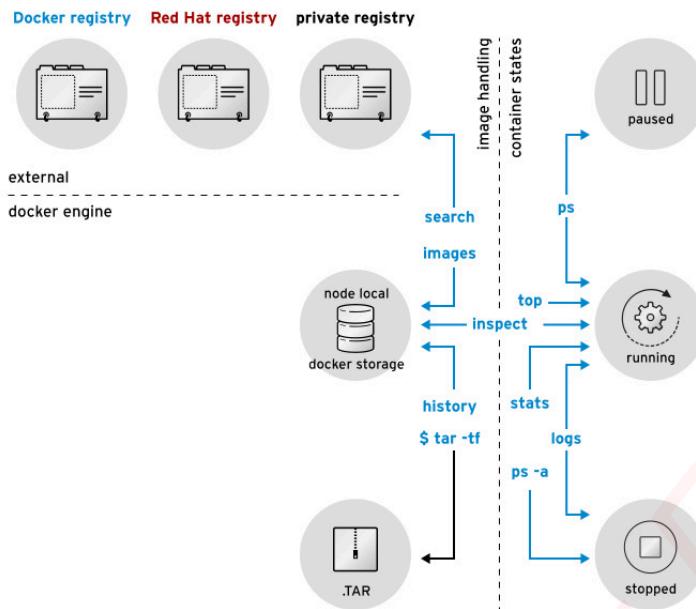


Figure 3.2: Docker client query verbs

Use these two figures as a reference while you learn about the **docker** command verbs along this course.

CREATING CONTAINERS

The **docker run** command creates a new container from an image and starts a process inside the new container. If the container image is not available, this command also tries to download it:

```
$ docker run rhscl/httpd-24-rhel7
Unable to find image 'rhscl/httpd-24-rhel7:latest' locally
Trying to pull repository registry.lab.example.com/rhscl/httpd-24-rhel7 ...
latest: Pulling from registry.lab.example.com/rhscl/httpd-24-rhel7
b12636467c49: Pull complete
b538cc6febe6: Pull complete
fd87b2ca7715: Pull complete
6861c26d5818: Pull complete
Digest: sha256:35f2b43891a7ebfa5330ef4c736e171c42380aec95329d863dcde0e608ffff1e
...
[Wed Jun 13 09:40:34.098087 2018] [core:notice] [pid 1] AH00094: Command line:
'httpd -D FOREGROUND'
$ ^C
```

In the previous output sample, the container was started with a noninteractive process, and stopping that process with **Ctrl+C (SIGINT)** also stops the container.

The management docker commands require an ID or a name. The **docker run** command generates a random ID and a random name that are unique. The **docker ps** command is responsible for displaying these attributes:

```
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS	NAMES	
347c9aad6049❶	rhscl/httpd-24-rhel7	"httpd -D FOREGROUND"	31 seconds
ago	80/tcp	<i>focused_fermat</i> ❷	

- ❶ This ID is generated automatically and must be unique.
- ❷ This name can be generated automatically or manually specified.

If desired, the container name can be explicitly defined. The **--name** option is responsible for defining the container name:

```
$ docker run --name my-nginx-container do180/nginx
```

NOTE

The name must be unique. An error is thrown if another container has the same name, including containers that are stopped.

Another important option is to run the container as a daemon, running the containerized process in the background. The **-d** option is responsible for running in detached mode. Using this option, the container ID is displayed on the screen:

```
$ docker run --name my-nginx-container -d do180/nginx
77d4b7b8ed1fd57449163bcb0b78d205e70d2314273263ab941c0c371ad56412
```

The container image itself specifies the command to run to start the containerized process, but a different one can be specified after the container image name in **docker run**:

```
$ docker run do180/nginx ls /tmp
anaconda-post.log
ks-script-1j4CXN
yum.log
```

The specified command must exist inside the container image.

NOTE

Since a specified command was provided in the previous example, the *HTTPD* service does not start.

Sometimes it is desired to run a container executing a Bash shell. This can be achieved with:

```
$ docker run --name my-nginx-container -it do180/nginx /bin/bash
bash-4.2#
```

Options **-t** and **-i** are usually needed for interactive text-based programs, so they get a proper terminal, but not for background daemons.

RUNNING COMMANDS IN A CONTAINER

When a container is created, a default command is executed according to what is specified by the container image. However, it may be necessary to execute other commands to manage the running container.

The **docker exec** command starts an additional process inside a running container:

```
$ docker exec 7ed6e671a600 cat /etc/hostname  
7ed6e671a600
```

The previous example used the container ID to execute the command. It is also possible to use the container name:

```
$ docker exec my-httdp-container cat /etc/hostname  
7ed6e671a600
```

DEMONSTRATION: CREATING CONTAINERS

1. Run the following command:

```
[student@workstation ~]$ docker run --name demo-container rhel7 \  
dd if=/dev/zero of=/dev/null
```

This command downloads the official Red Hat Enterprise Linux 7 container and starts it using the **dd** command. The container exits when the **dd** command returns the result. For educational purposes, the provided **dd** never stops.

2. Open a new terminal window from the workstation VM and check if the container is running:

```
[student@workstation ~]$ docker ps
```

Some information about the container, including the container name **demo-container** specified in the last step, is displayed.

3. Open a new terminal window and stop the container using the provided name:

```
[student@workstation ~]$ docker stop demo-container
```

4. Return to the original terminal window and verify that the container was stopped:

```
[student@workstation ~]$ docker ps
```

5. Start a new container without specifying a name:

```
[student@workstation ~]$ docker run rhel7 dd if=/dev/zero of=/dev/null
```

If a container name is not provided, **docker** generates a name for the container automatically.

6. Open a terminal window and verify the name that was generated:

```
[student@workstation ~]$ docker ps
```

An output similar to the following will be listed:

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS	NAMES	
05b725c0fd5a	rhel7	"dd if=/dev/zero of=/"	13 seconds ago
Up 11 seconds		reverent_blackwell	

The `reverent_blackwell` is the generated name. Students probably will have a different name for this step.

- Stop the container with the generated name:

```
[student@workstation ~]$ docker stop reverent_blackwell
```

- Containers can have a default long-running command. For these cases, it is possible to run a container as a daemon using the `-d` option. For example, when a MySQL container is started it creates the databases and keeps the server actively listening on its port. Another example using `dd` as the long-running command is as follows:

```
[student@workstation ~]$ docker run --name demo-container-2 -d rhel7 \
dd if=/dev/zero of=/dev/null
```

- Stop the container:

```
[student@workstation ~]$ docker stop demo-container-2
```

- Another possibility is to run a container to just execute a specific command:

```
[student@workstation ~]$ docker run --name demo-container-3 rhel7 ls /etc
```

This command starts a new container, lists all files available in the `/etc` directory in the container, and exits.

- Verify that the container is not running:

```
[student@workstation ~]$ docker ps
```

- It is possible to run a container in interactive mode. This mode allows for staying in the container when the container runs:

```
[student@workstation ~]$ docker run --name demo-container-4 -it rhel7 \
/bin/bash
```

The `-i` option specifies that this container should run in interactive mode, and the `-t` allocates a pseudo-TTY.

- Exit the Bash shell from the container:

```
[root@8b1580851134 /]# exit
```

- Remove all stopped containers from the environment by running the following from a terminal window:

```
[student@workstation ~]$ docker rm demo-container demo-container-2 \
demo-container-3 demo-container-4
```

15. Remove the container started without a name. Replace the <container_name> with the container name from the step 7:

```
[student@workstation ~]$ docker rm <container_name>
```

16. Remove the rhel7 container image:

```
[student@workstation ~]$ docker rmi rhel7
```

This concludes the demonstration.

MANAGING CONTAINERS

Docker provides the following commands to manage containers:

- **docker ps**: This command is responsible for listing running containers:

\$ docker ps	CONTAINER ID	IMAGE	COMMAND	CREATED
	STATUS	PORTS	NAMES	
	77d4b7b8ed1f ①	do180/httpd ②	"httpd -D FOREGROUND" ③	15 hours ago ④
	Up 15 hours ⑤	80/tcp ⑥	my-httpd-container ⑦	

- ① Each container, when created, gets a **container ID**, which is a hexadecimal number and looks like an image ID, but is actually unrelated.
- ② Container image that was used to start the container.
- ③ Command that was executed when the container started.
- ④ Date and time the container was started.
- ⑤ Total container uptime, if still running, or time since terminated.
- ⑥ Ports that were exposed by the container or the port forwards, if configured.
- ⑦ The container name.

Stopped containers are not discarded immediately. Their local file systems and other states are preserved so they can be inspected for *post-mortem* analysis. Option **-a** lists all containers, including containers that were not discarded yet:

\$ docker ps -a	CONTAINER ID	IMAGE	COMMAND	CREATED
	STATUS	PORTS	NAMES	
	4829d82fbbff	do180/httpd	"httpd -D FOREGROUND"	15 hours ago
	Exited (0) 3 seconds ago		my-httpd-container	

- **docker inspect**: This command is responsible for listing metadata about a running or stopped container. The command produces a **JSON** output:

```
$ docker inspect my-httpd-container
[
```

```
{
  "Id": "980e45b5376a4e966775fb49cbef47ee7bbd461be8bfd1a75c2cc5371676c8be",
  ...OUTPUT OMITTED...
  "NetworkSettings": {
    "Bridge": "",
    "EndpointID":
    "483fc91363e5d877ea8f9696854a1f14710a085c6719afc858792154905d801a",
    "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, the following command can be executed:

```
$ docker inspect -f '{{ .NetworkSettings.IPAddress }}' my-httdp-container
```

- **docker stop**: This command is responsible for stopping a running container gracefully:

```
$ docker stop my-httdp-container
```

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

- **docker kill**: This command is responsible for stopping a running container forcefully:

```
$ docker kill my-httdp-container
```

It is possible to specify the signal with the **-s** option:

```
$ docker kill -s SIGKILL my-httdp-container
```

The following signals are available:

SIGNAL	DEFAULT ACTION	DESCRIPTION
SIGHUP	Terminate process	Terminate line hangup
SIGINT	Terminate process	Interrupt program
SIGQUIT	Create core image	Quit program
SIGABRT	Create core image	Abort program
SIGKILL	Terminate process	Kill program
SIGTERM	Terminate process	Software termination signal
SIGUSR1	Terminate process	User-defined signal 1
SIGUSR2	Terminate process	User-defined signal 2

- **docker restart**: This command is responsible for restarting a stopped container:

```
$ docker restart my-httdp-container
```

The **docker restart** command creates a new container with the same container ID, reusing the stopped container state and filesystem.

- **docker rm**: This command is responsible for deleting a container, discarding its state and filesystem:

```
$ docker rm my-httdp-container
```

It is possible to delete all containers at the same time. The **docker ps** command has the **-q** option that returns only the ID of the containers. This list can be passed to the **docker rm** command:

```
$ docker rm $(docker ps -aq)
```

Before deleting all containers, all running containers must be stopped. It is possible to stop all containers with:

```
$ docker stop $(docker ps -q)
```



NOTE

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

DEMONSTRATION: MANAGING A CONTAINER

1. Run the following command:

```
[student@workstation ~]$ docker run --name demo-container -d rhsc1/httpd-24-rhel7
```

This command will start a **HTTPD** container as a daemon.

2. List all running containers:

```
[student@workstation ~]$ docker ps
```

3. Stop the container with the following command:

```
[student@workstation ~]$ docker stop demo-container
```

4. Verify that the container is not running:

```
[student@workstation ~]$ docker ps
```

5. Run a new container with the same name:

```
[student@workstation ~]$ docker run --name demo-container -d rhsc1/httpd-24-rhel7
```

A conflict error is displayed. Remember that a stopped container is not discarded immediately and their local file systems and other states are preserved so they can be inspected for *post-mortem* analysis.

6. It is possible to list all containers with the following command:

```
[student@workstation ~]$ docker ps -a
```

7. Start a new **HTTPD** container:

```
[student@workstation ~]$ docker run --name demo-1-httdp -d rhel7/httpd-24-rhel7
```

8. An important feature is the ability to list metadata about a running or stopped container. The following command returns the metadata:

```
[student@workstation ~]$ docker inspect demo-1-httdp
```

9. It is possible to format and retrieve a specific item from the **inspect** command. To retrieve the **IPAddress** attribute from the **NetworkSettings** object, use the following command:

```
[student@workstation ~]$ docker inspect -f '{{ .NetworkSettings.IPAddress }}' \
demo-1-httdp
```

Make a note about the IP address from this container. It will be necessary for a further step.

10. Run the following command to access the container **bash**:

```
[student@workstation ~]$ docker exec -it demo-1-httdp /bin/bash
```

11. Create a new HTML file on the container and exit:

```
bash-4.2# echo do180 > \
/opt/rh/httpd24/root/var/www/html/do180.html
bash-4.2# exit
```

12. Using the IP address from step 8, try to access the previously created page:

```
[student@workstation ~]$ curl IP:8080/do180.html
```

The following output is be displayed:

```
do180
```

13. It is possible to restart the container with the following command:

```
[student@workstation ~]$ docker restart demo-1-httdp
```

14. When the container is restarted, the data is preserved. Verify the IP address from the restarted container and check that the do180 page is still available:

```
[student@workstation ~]$ docker inspect demo-1-httdp | grep IPAddress
```

```
[student@workstation ~]$ curl IP:8080/do180.html
```

15. Stop the **HTTP** container:

```
[student@workstation ~]$ docker stop demo-1-httdp
```

16. Start a new **HTTP** container:

```
[student@workstation ~]$ docker run --name demo-2-httdp -d rhsc1/httdp-24-rhel7
```

17. Verify the IP address from the new container and check if the do180 page is available:

```
[student@workstation ~]$ docker inspect demo-2-httdp | grep IPAddress  
[student@workstation ~]$ curl IP:8080/do180.html
```

The page is not available because this page was created just for the previous container. New containers will not have the page since the container image did not change.

18. In case of a freeze, it is possible to kill a container like any process. The following command will kill a container:

```
[student@workstation ~]$ docker kill demo-2-httdp
```

This command kills the container with the **SIGKILL** signal. It is possible to specify the signal with the **-s** option.

19. Containers can be removed, discarding their state and filesystem. It is possible to remove a container by name or by its ID. Remove the **demo-httdp** container:

```
[student@workstation ~]$ docker ps -a  
[student@workstation ~]$ docker rm demo-1-httdp
```

20. It is also possible to remove all containers at the same time. The **-q** option returns the list of container IDs and the **docker rm** accepts a list of IDs to remove all containers:

```
[student@workstation ~]$ docker rm $(docker ps -aq)
```

21. Verify that all containers were removed:

```
[student@workstation ~]$ docker ps -a
```

22. Clean up the images downloaded by running the following from a terminal window:

```
[student@workstation ~]$ docker rmi rhsc1/httdp-24-rhel7
```

This concludes the demonstration.

► GUIDED EXERCISE

MANAGING A MYSQL CONTAINER

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

RESOURCES	
Files:	NA
Application URL:	NA
Resources:	RHSCL MySQL 5.7 container image (rhscl/mysql-57-rhel7)

OUTCOMES

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

BEFORE YOU BEGIN

The workstation should have docker running. To check if this is true, run the following command from a terminal window:

```
[student@workstation ~]$ lab managing-mysql setup
```

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

```
[student@workstation ~]$ docker run --name mysql-db rhscl/mysql-57-rhel7
```

This command downloads the MySQL database container image and tries to start it, but it does not start. The reason for this is the image requires a few 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 ~]$ docker 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 to start the new container with the correct name.

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

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

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS	NAMES	
5cd89eca81dd	rhscl/mysql-57-rhel7	"container-entrypoint"	9 seconds ago
Up 8 seconds	3306/tcp	mysql	

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

```
[student@workstation ~]$ docker inspect -f '{{ .NetworkSettings.IPAddress }}' \
mysql
172.17.0.2
```



NOTE

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

- 5. Connect to the MySQL database from the host:

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

Use **mypa55** as password.

- 6. You are connected to the **items** database. Create a new table:

```
MySQL [items]> CREATE TABLE Projects (id int(11) NOT NULL, \
name varchar(255) DEFAULT NULL, code varchar(255) DEFAULT NULL, \
PRIMARY KEY (id));
```

Use **mypa55** as password.



NOTE

Alternatively you can upload the database using the provided file:

```
[student@workstation ~]$ mysql -uuser1 -h IP -pmypa55 items \
< DO180/labs/managing-mysqldb/db.sql
```

- 7. Insert a row into the table by running the following command:

```
MySQL [items]> insert into Projects (id, name, code) values (1, 'DevOps', 'D0180');
```

- 8. Exit from the MySQL prompt:

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

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

```
[student@workstation ~]$ docker run --name mysql-2 -it rhsc1/mysql-57-rhel7 \
/bin/bash
bash-4.2$
```

- 10. Try to connect to the MySQL database:

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

The following error is displayed:

```
ERROR 2002 (HY000): Can't connect to local MySQL server through socket '/var/lib/
mysql/mysql.sock' (2)
```

The reason for this error is that the MySQL database server is not running because we changed the default command responsible for starting the database to **/bin/bash**.

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

```
bash-4.2$ exit
```

- 12. When you exit the **bash** shell, the container was stopped. Verify that the container **mysql-2** is not running:

```
[student@workstation ~]$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              NAMES
8b2c0ee86419        rhsc1/mysql-57-rhel7   "container-entrypoint"   4 minutes ago   Up 4 minutes      mysql
```

- 13. Verify that the database was correctly set up. Run the following from a terminal window:

```
[student@workstation ~]$ lab managing-mysql grade
```

► 14. Delete the containers and resources created by this lab.

14.1. Stop the running container, by running the following commands:

```
[student@workstation ~]$ docker stop mysql
```

14.2. Remove the container data by running the following commands:

```
[student@workstation ~]$ docker rm mysql
[student@workstation ~]$ docker rm mysql-2
[student@workstation ~]$ docker rm mysql-db
```

14.3. Remove the container image by running the following command:

```
[student@workstation ~]$ docker rmi rhsc1/mysql-57-rhel7
```

This concludes the guided exercise.

ATTACHING DOCKER PERSISTENT STORAGE

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 are supposed to work on the assumption that they always start with empty storage, and this makes creating and destroying containers relatively inexpensive operations.

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

Previously in this course, container images were characterized as **immutable** and **layered**, meaning they are never changed, but 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.

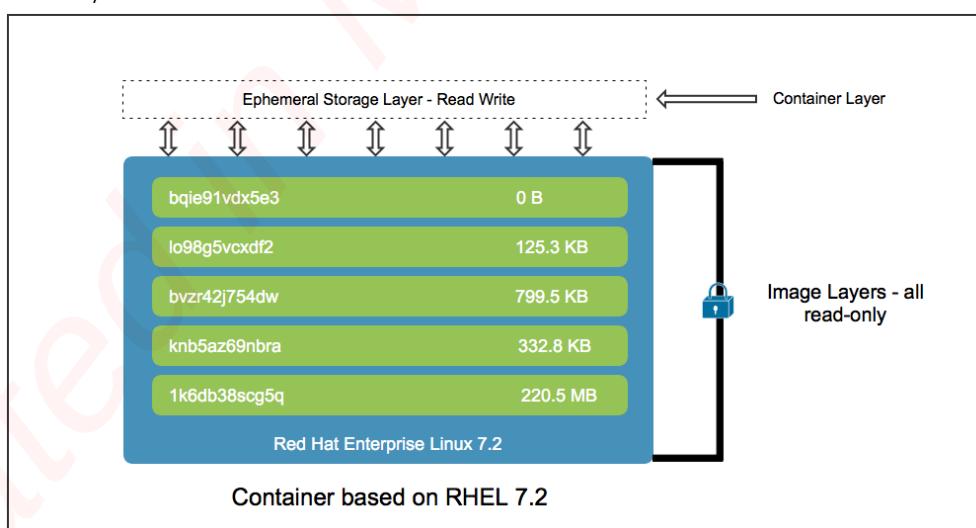


Figure 3.3: Container layers

Containerized applications should not try to use the container storage to store persistent data, as they cannot control how long its contents will be preserved. Even if it were possible to keep

container storage around for a long time, 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

Docker tries to keep old stopped container storage available for a while to be used by troubleshooting operations, such as reviewing a failed container logs for error messages. But this container storage can be reclaimed at any time to create new containers, including replacements for the old ones; for example, when the host is rebooted.

If the administrator needs to reclaim old containers storage sooner, the stopped container IDs can be found using `docker ps -a`, and the container then can be deleted using `docker rm container_id`. This last command also deletes the container storage.

Preparing the Host Directory

The Docker daemon can be requested to bind mount a host directory inside a running container. The host directory is seen by the containerized application as part of the container storage, much like a remote network volume is seen by applications as if it were part of the host file system. But these host directory contents are not reclaimed after the container is stopped, and it can be bind mounted to new containers whenever needed.

For example, a database container could be started using a host directory to store database files. If this database container dies, a new container can be created 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 `svirt_sandbox_file_t`.

One way to set up the host directory is:

1. Create a directory with owner and group `root` (notice the root prompt #):

```
# mkdir /var/dbfiles
```

2. The container user must be capable of writing files on the directory. If the host machine does not have the container user, the permission should be defined with the numeric user ID (UID) from the container. In case of the mysql service provided by Red Hat, the UID is 27:

```
# chown -R 27:27 /var/dbfiles
```

3. Allow containers (and also virtual machines) access to the directory:

```
# chcon -t svirt_sandbox_file_t /var/dbfiles
```

The host directory has to be configured **before** starting the container using it.

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 `docker 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:

```
# docker run -v /var/dbfiles:/var/lib/mysql mysql # other options required by the MySQL image omitted
```

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 lab, you will create a container that persists the MySQL database data into a host directory.

RESOURCES

Files:	NA
Application URL:	NA
Resources:	RHSCl MySQL 5.7 image (rhscl/mysql-57-rhel7)

OUTCOMES

You should be able to deploy a persistent database.

BEFORE YOU BEGIN

The workstation should not have any container images running. To achieve this goal, run from a terminal window the command:

```
[student@workstation ~]$ lab persist-mysqldb setup
```

- 1. Create a directory with the correct permissions.
 - 1.1. Open a terminal window from the **workstation** VM (Applications → System Tools → Terminal) and run the following command:

```
[student@workstation ~]$ sudo mkdir -p /var/local/mysql
```

- 1.2. Apply the appropriate SELinux context to the mount point.

```
[student@workstation ~]$ sudo chcon -R -t svirt_sandbox_file_t /var/local/mysql
```

- 1.3. Change the owner of the mount point to the mysql user and mysql group:

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



NOTE

The container user must be capable of writing files in the directory. If the host machine does not have the container user, set the permission to the numeric user ID (UID) from the container. In case of the **mysql** service provided by Red Hat, the UID is 27.

- 2. Create a MySQL container instance with persistent storage.
 - 2.1. Pull the MySQL container image from the internal registry:

```
[student@workstation ~]$ docker pull rhel/mysql-5.7-rhel7
```

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

```
[student@workstation ~]$ docker run --name persist-mysqldb \
-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 \
rhel/mysql-5.7-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.

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

```
[student@workstation ~]$ docker ps
```

An output similar to the following will be listed:

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS	NAMES	
8d6acf5aa5	rhel/mysql-5.7-rhel7	"container-entrypoint"	11 seconds ago 9
seconds ago	3306/tcp	persist-mysqldb	

- 3. 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 persists 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.

- 4. Verify if the database was correctly set up. Run the following from a terminal window:

```
[student@workstation ~]$ lab persist-mysqldb grade
```

- 5. Delete the containers and resources created by this lab.

- 5.1. Stop the running container, by running the following commands:

```
[student@workstation ~]$ docker stop persist-mysqldb
```

- 5.2. Remove the container data by running the following commands:

```
[student@workstation ~]$ docker rm persist-mysqldb
```

- 5.3. Remove the container image by running the following command:

```
[student@workstation ~]$ docker rmi rhsc1/mysql-57-rhel7
```

- 5.4. Delete the volume directory created earlier in the exercise by running the following command:

```
[student@workstation ~]$ sudo rm -rf /var/local/mysql
```

This concludes the guided exercise.

ACCESSING DOCKER NETWORKS

OBJECTIVES

After completing this section, students should be able to:

- Describe the basics of networking with containers.
- Connect to services within a container remotely.

INTRODUCING NETWORKING WITH CONTAINERS

By default, the Docker engine uses a bridged network mode, which through the use of **iptables** and NAT, allows containers to connect to the host machines network.

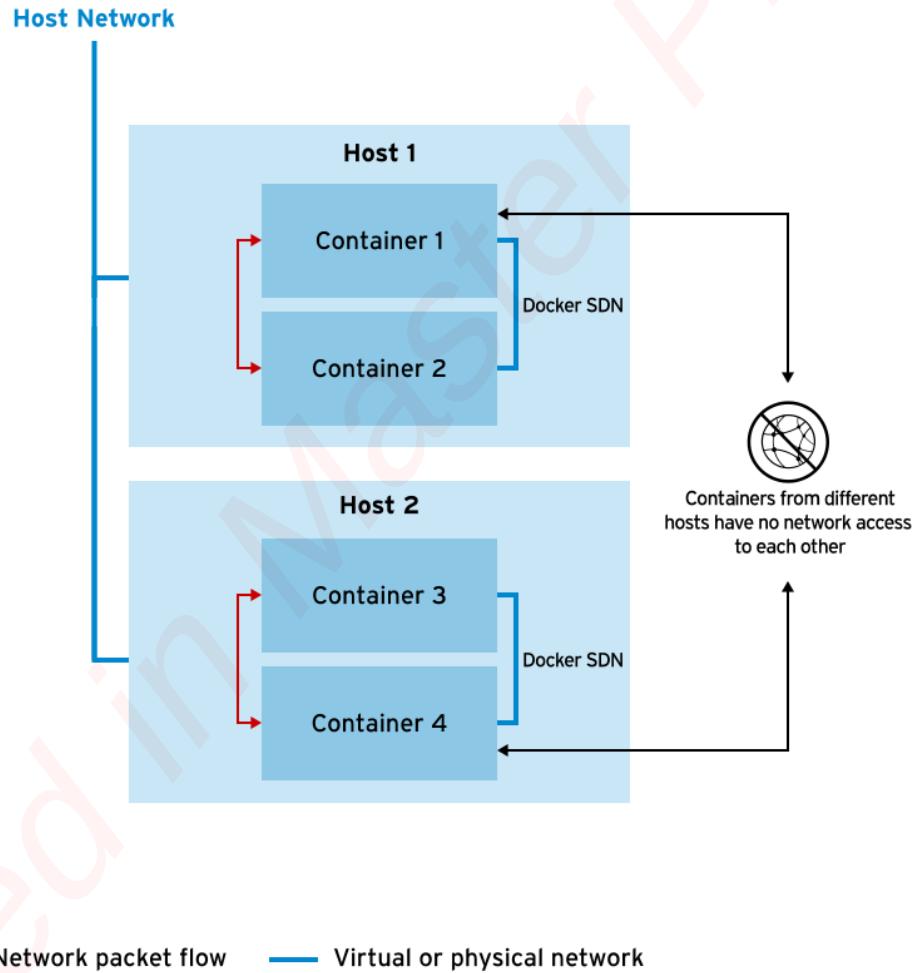


Figure 3.4: Basic Linux containers networking

Each container gets a networking stack, and Docker provides a bridge for these containers to communicate using a virtual switch. Containers running on a shared host each have a unique IP address and containers running on different Docker hosts can share an IP address. Moreover, all containers that connect to the same bridge on the same host can communicate with each other freely by IP address.

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

MAPPING NETWORK PORTS

Accessing the container from the external world can be a challenge. It is not possible to specify the IP address for the container that will be created, and the IP address changes for every new container. Another problem is that the container network is only accessible by the container host.

To solve these problems, it is possible to use the container host network model combined with network address translation (NAT) rules to allow the external access. To achieve this, use the **-p** option should be used:

```
# docker run -d --name httpd -p 8080:80 do276/httpd
```

In the previous example, requests received by the container host on port 8080 from any IP address will be forwarded to port 80 in the container.

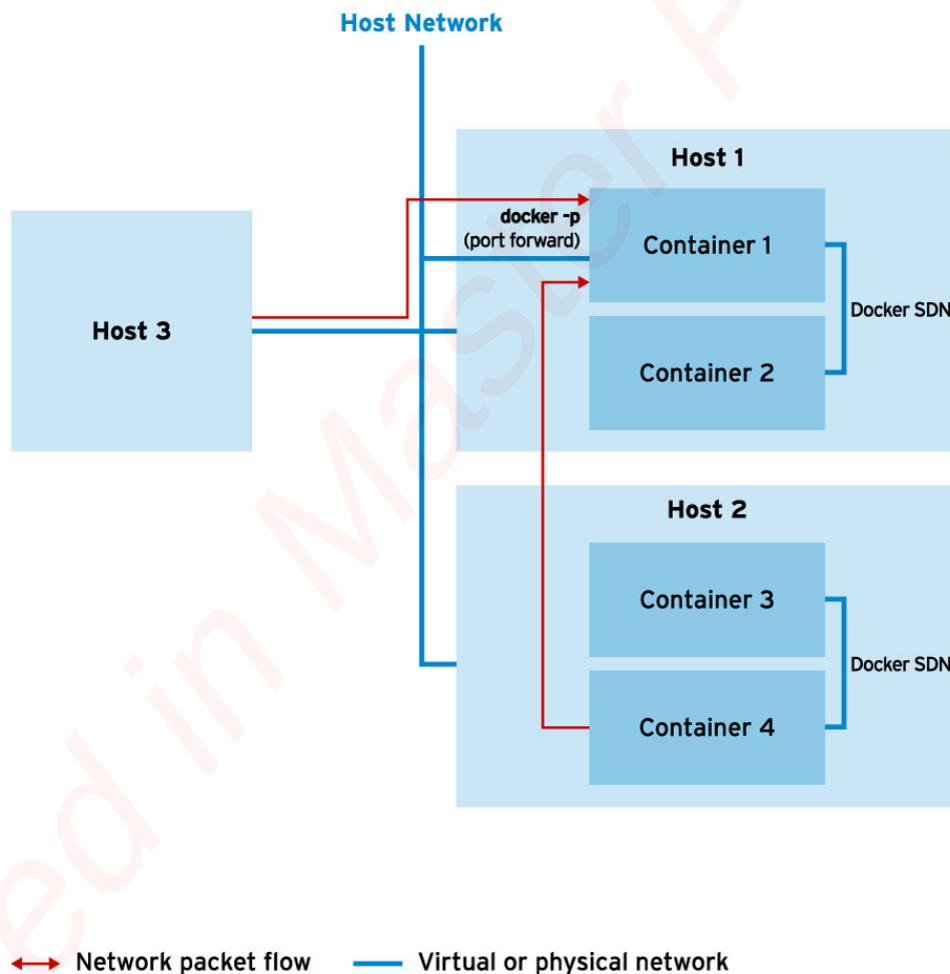


Figure 3.5: Allowing external accesses to Linux containers

It is also possible to specify an IP address for the port forward:

```
# docker run -d --name httpd -p 192.168.1.5:8080:80 do276/httpd
```

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

```
# docker run -d --name httpd -p 192.168.1.5::80 do276/httpd
```

Finally, it is possible to listen on all interfaces and have an available port picked automatically:

```
# docker run -d --name httpd -p 80 do276/httpd
```

► GUIDED EXERCISE

LOADING THE DATABASE

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

RESOURCES

Files:	NA
Application URL:	NA
Resources:	RHSCl MySQL 5.7 image (rhsc1/mysql-57-rhel7)

OUTCOMES

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

BEFORE YOU BEGIN

The workstation should have a directory to persist data from the database container. To ensure these requirements are supported by the workstation, the setup script creates the necessary directory. Run the following command from a terminal window:

```
[student@workstation ~]$ lab load-mysqldb setup
```

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

```
[student@workstation ~]$ docker run --name mysqldb-port \
-d -v /var/local/mysql:/var/lib/mysql/data \
-p 13306:3306 \
-e MYSQL_USER=user1 -e MYSQL_PASSWORD=mypa55 \
-e MYSQL_DATABASE=items -e MYSQL_ROOT_PASSWORD=r00tpa55 \
rhsc1/mysql-57-rhel7
```

The **-p** parameter is responsible for the port forward. In this case, every connection on the host IP using the port 13306 is forwarded to this container in port 3306.



NOTE

The **/var/local/mysql** directory was created and configured by the setup script to have the permissions required by the containerized database.

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

```
[student@workstation ~]$ docker ps
```

An output similar to the following will be listed. Look at the **PORTS** column and see the port forward.

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS	NAMES	
ad697775565b	rhscl/mysql-5.7-rhel7	"container-entrypoint"	4 seconds ago
Up 2 seconds	0.0.0.0:13306->3306/tcp	mysqlldb-port	

- ▶ 3. Load the database:

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

- ▶ 4. Verify that the database was successfully loaded:

```
[student@workstation ~]$ mysql -uuser1 -h 127.0.0.1 -pmypa55 \
-P13306 items -e "SELECT * FROM Item"
```

An output similar to the following will be listed:

id	description	done
1	Pick up newspaper	0
2	Buy groceries	1

- ▶ 5. Another way to verify that the database was successfully loaded is by running the **mysql** command inside the container. To do that, access the container **bash**:

```
[student@workstation ~]$ docker exec -it mysql ldb-port /bin/bash
```

- ▶ 6. Verify that the database contains data:

```
bash-4.2$ mysql -uroot items -e "SELECT * FROM Item"
```

- ▶ 7. Exit from the **bash** shell inside the container:

```
bash-4.2$ exit
```

- ▶ 8. There is a third option to verify that the database was successfully loaded. It is possible to inject a process into the container to check if the database contains data:

```
[student@workstation ~]$ docker exec -it mysql ldb-port \
```

```
/opt/rh/rh-mysql57/root/usr/bin/mysql -uroot items -e "SELECT * FROM Item"
```

**NOTE**

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

- ▶ 9. Verify that the databases were correctly set up. Run the following from a terminal window:

```
[student@workstation ~]$ lab load-mysqldb grade
```

- ▶ 10. Delete the container and volume created by this lab.

10.1. To stop the container, run the following command:

```
[student@workstation ~]$ docker stop mysqldb-port
```

10.2. To remove the data stored by the stopped container, run the following command:

```
[student@workstation ~]$ docker rm mysqldb-port
```

10.3. To remove the container image, run the following command:

```
[student@workstation ~]$ docker rmi rhsc1/mysql-57-rhel7
```

10.4. To remove the directory with the database data, run the following command:

```
[student@workstation ~]$ sudo rm -rf /var/local/mysql
```

This concludes the guided exercise.

▶ LAB

MANAGING CONTAINERS

PERFORMANCE CHECKLIST

In this lab, you will deploy a container that persists the MySQL database data into a host folder, load the database, and manage the container.

RESOURCES	
Files:	/home/student/D0180/labs/work-containers
Application URL:	NA
Resources:	RHSCL MySQL 5.7 image (rhscl/mysql-57-rhel7)

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 as they are using the same directory on the host to store the MySQL data.

The workstation should have Docker running already. To verify this and download the necessary files for the lab, run the following command from a terminal window:

```
[student@workstation ~]$ lab work-containers setup
```

1. Create the **/var/local/mysql** directory with the correct permission.
 - 1.1. Create the host folder to store the MySQL database data.
 - 1.2. Apply the appropriate SELinux context to the host folder.
 - 1.3. Change the owner of the host folder to the mysql user (**uid=27**) and mysql group (**gid = 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.
 - 2.2. Verify that the container was started correctly.
3. Load the **items** database using the **/home/student/D0180/labs/work-containers/db.sql** script.
 - 3.1. Get the container IP.
 - 3.2. Load the database.
 - 3.3. Verify that the database was loaded.
4. Stop the container gracefully.

**NOTE**

This step is very important since 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:** **rhscl/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.
 - 5.2. Verify that the container was started correctly.
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 the port forward, insert a new **Finished lab** row in the **Item** table.
 - 8.1. Connect to the MySQL database.
 - 8.2. Insert the new **Finished lab** row.
 - 8.3. Exit from the MySQL client.
9. Since the first container is not required any more, remove it from the Docker daemon to release resources.
10. Verify that the lab was correctly executed. Run the following from a terminal window:

```
[student@workstation ~]$ lab work-containers grade
```

11. Delete the containers and resources created by this lab.
 - 11.1. Stop the running container.
 - 11.2. Remove the container storage.
 - 11.3. Remove the container image.
 - 11.4. Remove the file created to store the information about the containers.
 - 11.5. Remove the host directory used by the container volumes.

Cleanup

From **workstation**, run the **lab work-containers cleanup** command to clean up the environment.

```
[student@workstation ~]$ lab work-containers cleanup
```

This concludes the lab.

► SOLUTION

MANAGING CONTAINERS

PERFORMANCE CHECKLIST

In this lab, you will deploy a container that persists the MySQL database data into a host folder, load the database, and manage the container.

RESOURCES	
Files:	/home/student/D0180/labs/work-containers
Application URL:	NA
Resources:	RHSCl MySQL 5.7 image (rhscl/mysql-57-rhel7)

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 as they are using the same directory on the host to store the MySQL data.

The workstation should have Docker running already. To verify this and download the necessary files for the lab, run the following command from a terminal window:

```
[student@workstation ~]$ lab work-containers setup
```

1. Create the **/var/local/mysql** directory with the correct permission.
 - 1.1. Create the host folder to store the MySQL database data.
Open a terminal window from the workstation VM (Applications → Utilities → Terminal) and run the following command:


```
[student@workstation ~]$ sudo mkdir -p /var/local/mysql
```
 - 1.2. Apply the appropriate SELinux context to the host folder.


```
[student@workstation ~]$ sudo chcon -R -t svirt_sandbox_file_t /var/local/mysql
```
- 1.3. Change the owner of the host folder to the mysql user (**uid=27**) and mysql group (**gid = 27**).


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

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 ~]$ docker 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
```

2.2. Verify that the container was started correctly.

```
[student@workstation ~]$ docker ps
```

An output similar to the following will be listed:

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS	NAMES	
616azfaa55x8	rhscl/mysql-57-rhel7	"container-entrypoint"	11 seconds ago
9 seconds ago	3306/tcp	mysql-1	

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

3.1. Get the container IP.

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

3.2. Load the database.

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

Where `CONTAINER_IP` is the IP address returned by the previous command.

3.3. Verify that the database was loaded.

```
[student@workstation ~]$ mysql -uuser1 -h CONTAINER_IP -pmypa55 items \
-e "SELECT * FROM Item"
```

- Stop the container gracefully.

**NOTE**

This step is very important since 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.

Stop the container using the following command:

```
[student@workstation ~]$ docker stop mysql-1
```

- 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:** **rhscl/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

- Create and start the container.

```
[student@workstation ~]$ docker run --name mysql-2 \
-d -v /var/local/mysql:/var/lib/mysql/data \
-p 13306:3306 \
-e MYSQL_USER=user1 -e MYSQL_PASSWORD=mypa55 \
-e MYSQL_DATABASE=items -e MYSQL_ROOT_PASSWORD=r00tpa55 \
rhscl/mysql-57-rhel7
```

- Verify that the container was started correctly.

```
[student@workstation ~]$ docker ps
```

An output similar to the following will be listed:

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS	NAMES	
281c0e2790e5	rhscl/mysql-57-rhel7	"container-entrypoint"	14 seconds ago
11 seconds ago	0.0.0.0:13306->3306/tcp	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 ~]$ docker 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 ~]$ docker 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;
```

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

```
mysql> use items;
mysql> show tables;
```

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
bash-4.2$ exit
```

8. Using the port forward, insert a new `Finished lab` row in the `Item` table.

8.1. Connect to the MySQL database.

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

8.2. Insert the new `Finished lab` row.

```
MySQL[items]> insert into Item (description, done) values ('Finished lab', 1);
```

8.3. Exit from the MySQL client.

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

9. Since the first container is not required any more, remove it from the Docker daemon to release resources.

Remove the container with the following command:

```
[student@workstation ~]$ docker rm mysql-1
```

10. Verify that the lab was correctly executed. Run the following from a terminal window:

```
[student@workstation ~]$ lab work-containers grade
```

11. Delete the containers and resources created by this lab.

11.1. Stop the running container.

```
[student@workstation ~]$ docker stop mysql-2
```

11.2. Remove the container storage.

```
[student@workstation ~]$ docker rm mysql-2
```

11.3. Remove the container image.

```
[student@workstation ~]$ docker rmi rhsc1/mysql-57-rhel7
```

11.4. Remove the file created to store the information about the containers.

```
[student@workstation ~]$ rm /tmp/my-containers
```

11.5. Remove the host directory used by the container volumes.

```
[student@workstation ~]$ sudo rm -rf /var/local/mysql
```

Cleanup

From **workstation**, run the **lab work-containers cleanup** command to clean up the environment.

```
[student@workstation ~]$ lab work-containers cleanup
```

This concludes the lab.

SUMMARY

In this chapter, you learned:

- A set of commands are provided to create and manage containers.
 - **docker run**: Create a new container.
 - **docker ps**: List containers.
 - **docker inspect**: List metadata about a container.
 - **docker stop**: Stop a container.
 - **docker kill**: Stop a container forcefully.
 - **docker restart**: Restart a stopped container.
 - **docker rm**: Delete a container.
- Container storage is said to be ephemeral, meaning its contents are not preserved after the container is removed.
- To work with persistent data, a folder from the host can be used.
- It is possible to mount a volume with the **-v** option in the **docker run** command.
- The **docker exec** command starts an additional process inside a running container.
- A port mapping can be used with the **-p** option in the **docker run** command.

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 .
- List the advantages of using a certified public registry to download secure images.
- Customize the **docker** daemon to access alternative container image registries.
- Search for container images using **docker** command and the REST API.
- Pull images from a registry.
- List images downloaded from a registry to the daemon cache.
- Manage tags to pull tagged images.

PUBLIC REGISTRIES

The **docker** daemon searches for and downloads container images from a public registry provided by Docker. Docker Hub is the public registry managed by Docker, and it hosts a large set of container images, including those provided by major open source projects, such as Apache, MySQL, and Jenkins. It also hosts customized container images developed by the community.

Some images provided by the community do not take security concerns into consideration, and might put data or the application running in a production environment at risk, because anyone can get an account and publish custom images.

For example, root-based access containers and security-flawed tools (such as the Bash with ShellShock security vulnerability) might be some of the issues encountered in such containers.

Alternatively, Red Hat also has a public registry where certified and tested container images are available for consumption by customers with a valid Red Hat subscription.

Red Hat container images provide the following benefits:

- *Trusted source*: All container images are built from a known source 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.
- *Red Hat Enterprise Linux (RHEL) compatible*: Container images are compatible with all Red Hat Enterprise Linux platforms, from bare metal to cloud.
- *Red Hat support*: The complete stack is commercially supported by Red Hat.

PRIVATE REGISTRY

Some teams might need to distribute custom container images for internal use. Even though it is possible to use a public registry to make them available for download, a better approach would be to publish them to a private registry. A private registry can be installed as a service on a host for

development purposes. To use this internal registry, developers only need to add it to the list of registries to use by updating the **ADD_REGISTRY** variable.



NOTE

The **docker-registry** service installation and customization process is beyond the scope of this course.

To configure extra registries for the **docker** daemon, you need to update the **/etc/sysconfig/docker** file. On a RHEL host, add the following extra parameter:

```
ADD_REGISTRY='--add-registry registry.access.redhat.com \
--add-registry① services.lab.example.com:5000②'
```

- ① The **--add-registry** parameter requires the registry FQDN host and port.
- ② The FQDN host and port number where the **docker-registry** service is running.



NOTE

The **docker** daemon requires a full restart to make them effective by running **systemctl restart docker.service** command.

To access a registry, a secure connection is needed with a certificate. For a closed environment where only known hosts are allowed, the **/etc/sysconfig/docker** file can be customized to support insecure connections from a RHEL host:

```
ADD_REGISTRY='--add-registry① registry.lab.example.com②'
```

- ① The **--add-registry** parameter requires the FQDN of the registry. Specify the port if it is different than port 5000.
- ② The FQDN host and port number that the **docker-registry** service uses.

ACCESSING REGISTRIES

A container image registry is accessed via the Docker daemon service from the **docker** command. Because the **docker** command line uses a RESTful API to request process execution by the daemon, most of the commands from the client are translated into an HTTP request, and can be sent using **curl**.



NOTE

This capability can be used to get additional information from the registries and troubleshoot **docker** client problems that are not clearly stated by the logs.

Searching for Images in Registries

The subcommand **search** is provided by the **docker** command to find images by image name, user name, or description from all the registries listed in the **/etc/sysconfig/docker** configuration file. The syntax for the verb is:

```
# docker search [OPTIONS] <term>
```

The following table shows the options available for the **search** verb to limit the output from the command:

OPTION	DESCRIPTION
--automated=true	List only automated builds, where the image files are built using a Dockerfile.
--no-trunc=true	Do not truncate the output.
--stars=N	Display images with at least N stars provided by users from the registry.

**NOTE**

The command returns up to 25 results from the registry and does not display which tags are available for download.

To overcome the limitations of the **search** verb, the RESTful API can be used instead.

**NOTE**

To send an HTTP request to a container registry, a tool with HTTP support should be used, such as **curl** or a web browser.

**NOTE**

Images are stored in collections, known as a *repositories*, as seen throughout the API specification. Registry instances may contain several repositories.

To customize the number of container images listed from a registry, a parameter called **n** is used to return a different number of images:

```
GET /v2/_catalog?n=<number>
```

For example, to get the list of images present in the registry, run the following **curl** command:

```
# curl https://registry.lab.example.com/v2/_catalog?n=4 | python -m json.tool
```

```
{
  "repositories": [
    "jboss-eap-7/eap70-openshift",
    "jboss-fuse-6/fis-java-openshift",
    "jboss-fuse-6/fis-karaf-openshift",
    "jboss-webserver-3/webserver31-tomcat8-openshift"
  ]
}
... output omitted ...
```

Searching for Image Tags in Registries

To get the tags from any image, use the RESTful API. The HTTP request must be similar to:

```
GET /v2/repository name/tags/list
```

The following procedure describes the required steps for authenticating with the V2 API of the official Docker registry.

1. Define variables for your user name and password.

```
# UNAME="username"
# UPASS="password"
```

2. Retrieve the authentication token. Pass the user name and password with the **POST** method.

```
# curl -s -H "Content-Type: application/json" \
-X POST -d '{"username": "'${UNAME}'", "password": "'${UPASS}'"}' \
https://hub.docker.com/v2/users/login/ | jq -r .token
```

3. Retrieve the list of repositories that you have access to. The **Authorization: JWT** header is used for authenticating the request.

```
# curl -s -H "Authorization: JWT token" \
https://hub.docker.com/v2/repositories/user name/?page_size=10000 | \
jq -r '.results|.[].name'
```

4. You can also retrieve a list of all images and tags by iterating on the output of the previous command.

```
# repository=$(curl -s -H "Authorization: JWT token" \
https://hub.docker.com/v2/repositories/user name/?page_size=10000 | \
jq -r '.results|.[].name')

# for image in ${repository}; do \
curl -s -H "Authorization: JWT token" \
https://hub.docker.com/v2/repositories/user name/${image}/tags/?page_size=100 | \
jq -r '.results|.[].name')
```

Searching for Images in Private Registries

You can use the **docker** command with the **search** option to browse a Docker registry. Public registries, such as `registry.access.redhat.com` support the **search** verb, as they are exposing the version 1 of the API. The **search** verb does not work in this classroom because the registry exposes the version 2 of the API.

To use the search feature in registries running the version 2 of the API, a Python script can be used. The script is provided as an open source project hosted at <https://github.com> and it is referred in the References section.

In the classroom environment, the script is provided as a bash script named **docker-registry-cli**. The script can list and search all the repositories available. It also supports HTTPS and basic authentication. To get all the images available at a private registry, use the following syntax:

```
# docker-registry-cli <docker-registry-host>:<port> <list|search> [options] [ssl]
```

For example, to get the list of all images available at `registry.lab.example.com`:

```
# docker-registry-cli registry.lab.example.com list all ssl
-----
1) Name: rhscl/postgresql-95-rhel7
Tags: latest
-----
2) Name: openshift3/container-engine
Tags: latest v3.9 v3.9.14
-----
... output omitted ...
```

To search for a specific string, use the following command:

```
# docker-registry-cli registry.lab.example.com search mysql ssl
-----
1) Name: rhscl/mysql-57-rhel7
Tags: 5.7-3.14 latest
-----
2) Name: openshift3/mysql-55-rhel7
Tags: latest
-----
3) Name: rhscl/mysql-56-rhel7
Tags: latest
... output omitted ...
```

Pulling Images

To pull container images from a registry, use the **docker** command with the **pull** option.

```
# docker pull [OPTIONS] NAME[:TAG] | [REGISTRY_HOST[:REGISTRY_PORT]/]NAME[:TAG]
```

The following table shows the options available for the **pull** verb:

OPTION	DESCRIPTION
--all-tags=true	Download all tagged images in the repository.
--disable-content-trust=true	Skip image verification.

To pull an image from a registry, **docker pull** will use the image name obtained from the **search** verb. The **docker pull** command supports the fully qualified domain name to identify from which registry the image should be pulled. This option is supported because multiple registries can be used by **docker** for searching purposes, and the same image name can be used by multiple registries for different images.

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

```
# docker pull docker.io/nginx
```



NOTE

If no registry is provided, the first registry listed in the **/etc/sysconfig/docker** configuration file from the **ADD_REGISTRY** line is used.

Listing Cached Copies of Images

Any image files pulled from a registry are stored on the same host where the **docker** daemon is running to avoid multiple downloads, and to minimize the deployment time for a container. Moreover, any custom container image built by a developer is saved to the same cache. To list all the container images cached by the daemon, **docker** provides a verb called **images**.

```
# docker images
REPOSITORY      TAG      IMAGE ID      CREATED      SIZE
docker.io/httpd  latest  1c0076966428  4 weeks ago  193.4 MB
```



NOTE

The image files are stored in the **/var/lib/docker** directory from the **docker** daemon's host if the default configuration is used. If LVM thin storage is used to store images, the LVM volume mount point is used instead.

Image Tags

An image tag is a mechanism from the **docker-registry** service to support multiple releases of the same project. This facility 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 operation where a container image is requested from a registry accepts a tag parameter to differentiate between multiple tags. If no tag is provided, then **latest** is used, which indicates the latest build of the image that has been published. For example, to pull an image with the tag **5.5** from **mysql**, use the following command:

```
# docker pull mysql:5.6
```

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

```
# docker run mysql:5.6
```



REFERENCES

Docker Hub website

<https://hub.docker.com>

The Docker Registry API documentation

https://docs.docker.com/v1.6/reference/api/registry_api/#search

Docker registry v2 CLI

https://github.com/vivekjuneja/docker_registry_cli/

Docker remote API documentation

https://docs.docker.com/engine/reference/api/docker_remote_api/

Red Hat Container Catalog

<https://registry.access.redhat.com>

Red Hat container certification program website

<https://connect.redhat.com/zones/containers/why-certify-containers>

Setting up a docker-registry container

<https://docs.docker.com/registry/deploying/>

► QUIZ

WORKING WITH REGISTRIES

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

A docker daemon is installed on a RHEL host with the following `/etc/sysconfig/docker` file:

```
ADD_REGISTRY="--add-registry registry.access.redhat.com --add-registry
docker.io"
```

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

- `registry.access.redhat.com`:

image names/tags:

- `nginx/1.0`
- `mysql/5.6`
- `httpd/2.2`

- `docker.io`:

image names/tags:

- `mysql/5.5`
- `httpd/2.4`

No images were downloaded by the daemon.

► 1. Which two commands search for the `mysql` image available for download from `registry.access.redhat.com`? (Select two.)

- a. `docker search registry.access.redhat.com/mysql`
- b. `docker images`
- c. `docker pull mysql`
- d. `docker search mysql`

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

- a. `docker search httpd`
- b. `docker images httpd`
- c. `docker pull --all-tags=true httpd`
- d. There is no docker command available to search for tags.

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

- a. `docker pull httpd:2.2`
- b. `docker pull httpd:latest`
- c. `docker pull docker.io/httpd`
- d. `docker pull registry.access.redhat.com/httpd:2.2`

► 4. After running the following commands, what is the output of the `docker images` command?

```
docker pull registry.access.redhat.com/httpd:2.2  
docker pull docker.io/mysql:5.6
```

a. Option 1:

REPOSITORY	TAG
docker.io/httpd	2.2
registry.access.redhat.com/mysql	5.6

b. Option 2:

REPOSITORY	TAG
registry.access.redhat.com/httpd	2.2
registry.access.redhat.com/mysql	5.6

c. Option 3:

REPOSITORY	TAG
registry.access.redhat.com/httpd	2.2

d. Option 4:

REPOSITORY	TAG
docker.io/httpd	2.2

► SOLUTION

WORKING WITH REGISTRIES

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

A docker daemon is installed on a RHEL host with the following `/etc/sysconfig/docker` file:

```
ADD_REGISTRY="--add-registry registry.access.redhat.com --add-registry
docker.io"
```

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

- `registry.access.redhat.com`:

image names/tags:

- `nginx/1.0`
- `mysql/5.6`
- `httpd/2.2`

- `docker.io`:

image names/tags:

- `mysql/5.5`
- `httpd/2.4`

No images were downloaded by the daemon.

► 1. Which two commands search for the `mysql` image available for download from `registry.access.redhat.com`? (Select two.)

- a. `docker search registry.access.redhat.com/mysql`
- b. `docker images`
- c. `docker pull mysql`
- d. `docker search mysql`

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

- a. `docker search httpd`
- b. `docker images httpd`
- c. `docker pull --all-tags=true httpd`
- d. There is no docker command available to search for tags.

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

- a. `docker pull httpd:2.2`
- b. `docker pull httpd:latest`
- c. `docker pull docker.io/httpd`
- d. `docker pull registry.access.redhat.com/httpd:2.2`

► 4. After running the following commands, what is the output of the `docker images` command?

```
docker pull registry.access.redhat.com/httpd:2.2
docker pull docker.io/mysql:5.6
```

a. Option 1:

REPOSITORY	TAG
docker.io/httpd	2.2
registry.access.redhat.com/mysql	5.6

b. Option 2:

REPOSITORY	TAG
registry.access.redhat.com/httpd	2.2
registry.access.redhat.com/mysql	5.6

c. Option 3:

REPOSITORY	TAG
registry.access.redhat.com/httpd	2.2

d. Option 4:

REPOSITORY	TAG
docker.io/httpd	2.2

MANIPULATING CONTAINER IMAGES

OBJECTIVES

After completing this section, students should be able to:

- Export, import, and manage container images locally and in a registry.
- Create a new container image using the **commit** command.
- Identify the changed artifacts in a container.
- Manage image tags for distribution purposes.

INTRODUCTION

There are various ways to manage image containers in a devops fashion. For example, a developer finished 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 accomplish 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 (**docker commit**). However, the recommended way to do so, that is, using **Dockerfiles** is discussed in next chapters.

SAVING AND LOADING IMAGES

Existing images from the Docker cache can be saved to a **.tar** file using the **docker save** command. The generated file is not a regular **tar** file: it contains image metadata and preserves the original image layers. By doing so, the original image can be later recreated exactly as it was.

The general syntax of the **docker** command **save** verb is as follows::

```
# docker save [-o FILE_NAME] IMAGE_NAME[:TAG]
```

If the **-o** option is not used the generated image is sent to the standard output as binary data.

In the following example, the MySQL container image from the Red Hat Container Catalog is saved to the **mysql.tar** file:

```
# docker save -o mysql.tar registry.access.redhat.com/rhscl/mysql-57-rhel7
```

.tar files generated using the **save** verb can be used for backup purposes. To restore the container image, use the **docker load** command. The general syntax of the command is as follows:

```
# docker load [-i FILE_NAME]
```

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

An image previously saved can be restored to the Docker cache by using the **load** verb.

```
# docker load -i mysql.tar
```



NOTE

To save disk space, the file generated by the **save** verb can be compressed as a Gzip file. The **load** verb uses the **gunzip** command before importing the file to the cache directory.

PUBLISHING IMAGES TO A REGISTRY

To publish an image to the registry, it must be stored in the Docker's cache cache, and should be tagged for identification purposes. To tag an image, use the **tag** verb, as follows:

```
# docker tag IMAGE[:TAG] [REGISTRYHOST/]USERNAME/]NAME[:TAG]
```

For example, to tag the **nginx** image with the **latest** tag, run the following command:

```
# docker tag nginx nginx
```

To push the image to the registry, use the **push** verb, as follows:

```
# docker push nginx
```

DELETING IMAGES

Any image downloaded to the Docker cache is kept there even if it is not used 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 cache has the latest version of an image.

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

```
# docker rmi [OPTIONS] IMAGE [IMAGE...]
```

The major option available for the **rmi** subcommand is **--force=true**. The option forces the removal of an image. An image can be referenced using its name or its ID for removal purposes.

A single image might use more than one tag, and using the **docker rmi** command with the image ID will fail in this case. To avoid removing each tag individually, the simplest approach is to use the **--force** option.

Any container using the image will block any attempt to delete an image. All the containers using that image must be stopped and removed before it can be deleted.

DELETING ALL IMAGES

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

```
# docker rmi $(docker images -q)
```

The command returns all the image IDs available in the cache, and passes them as a parameter to the **docker 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 container images may be provided as they are, without any **Dockerfile**. As an alternative approach to creating new images, a running container can be changed in place and its layers saved to create a new container image. This feature is provided by the **docker commit** command.



WARNING

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

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

```
# docker commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]
```

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

OPTION	DESCRIPTION
--author=""	Identifies who created the container image.
--message=""	Includes a commit message to the registry.

To identify a running container in Docker, run the **docker ps** command:

```
# docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS
NAMES
87bdfcc7c656 mysql "/entrypoint.sh mysql" 14 seconds ago Up 13 seconds 3306/tcp
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** verb. The verb only requires the container name or container ID:

```
# docker diff mysql-basic
C /run
C /run/mysqld
A /run/mysqld/mysqld.pid
A /run/mysqld/mysqld.sock
A /run/mysqld/mysqld.sock.lock
A /run/secrets
```

Any added file is tagged with an **A**, and any changed file is tagged with a **C**.

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

```
# docker 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; however, this approach requires more maintenance for managing and deploying the images to the correct locations.

Container image registries support tags in order 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 easily identify a release. On the official MySQL container image website, the version is used as the tag's name (**5.5.16**). In addition, the same image has a second tag with the minor version, for example 5.5, to minimize the need to get the latest release for a certain version.

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

```
# docker tag [OPTIONS] IMAGE[:TAG] [REGISTRYHOST/]USERNAME/]NAME[:TAG]
```

The **IMAGE** argument is the image name with an optional tag which is managed by Docker. The following argument refers to alternative names for the image that is stored locally. If the tag value is not provided, Docker assumes the latest version, as indicated by the **latest** tag. For example, to tag an image, use the following command:

```
# docker tag mysql-custom devops/mysql
```

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

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

```
# docker tag mysql-custom devops/mysql:snapshot
```

Removing Tags from Images

To associate multiple tags with a single image, use the **docker tag** command. Tags can be removed by using the **docker rmi** command, as mentioned earlier:

```
# docker rmi devops/mysql:snapshot
```



NOTE

Because multiple tags can point to the same image, to remove an image referred to by multiple tags, each tag should be individually removed first. Alternatively, use the **docker rmi --force** command.

BEST PRACTICES FOR TAGGING IMAGES

The **latest** tag is automatically added by Docker if you do not specify any tag, as Docker consider the image to be the latest build. However, this may not be true depending on how the tags are used. 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 certain 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 and pointed to the same image from the **2.1.10** release. This simplifies how the image is pulled from the registry.



REFERENCES

Docker documentation

<https://docs.docker.com/>

► GUIDED EXERCISE

CREATING A CUSTOM APACHE CONTAINER IMAGE

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

RESOURCES

Application URL:	http://127.0.0.1:8180/do180.html , http://127.0.0.1:8280/do180.html
Resources:	RHSCl HTTPD image (rhscl/httpd-24-rhel7)

OUTCOMES

You should be able to create a custom container image.

BEFORE YOU BEGIN

To set up the environment for the exercise, open a terminal and run the following command. The script ensures that the Docker daemon is running on the **workstation** VM.

```
[student@workstation ~]$ lab container-images-manipulating setup
```

- 1. Open a terminal on **workstation** (Applications → System Tools → Terminal) and start a container by using the image available at **rhscl/httpd-24-rhel7**. The **-p** option allows you to specify a redirect port. In this case, Docker forwards incoming request on TCP port 8180 to the TCP port 8080.

```
[student@workstation ~]$ docker run -d \
--name official-httdp -p 8180:8080 \
rhscl/httpd-24-rhel7
... output omitted ...
Digest: sha256:35f2b43891a7ebfa5330ef4c736e171c42380aec95329d863dcde0e608ffff1e
Status: Downloaded newer image for registry.lab.example.com/rhscl/httpd-24-
rhel7:latest
e37d6532638fe85b8d92ef8a60dcfb22bf154d6f6852dd4c4baef670f3d11f4a
```

- 2. Create an HTML page on the **official-httdp** container.
- 2.1. Access the shell of the container by using the **exec** option and create an HTML page.

```
[student@workstation ~]$ docker exec -it official-httdp /bin/bash
bash-4.2$ echo "D0180 Page" > /var/www/html/do180.html
```

- 2.2. Exit the container.

```
bash-4.2$ 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:

```
DO180 Page
```

- 3. Examine the differences in the container between the image and the new layer created by the container. To do so, use the **diff** option.

```
[student@workstation ~]$ docker diff official-httdp
```

```
C /etc
C /etc/httdp
C /etc/httdp/conf
C /etc/httdp/conf/httdp.conf
C /etc/httdp/conf.d
C /etc/httdp/conf.d/ssl.conf
C /opt
C /opt/app-root
C /opt/app-root/etc
A /opt/app-root/etc/passwd
C /opt/app-root/src
A /opt/app-root/src/.bash_history
C /run/httdp
A /run/httdp/httdp.pid
C /tmp
C /var
C /var/www
C /var/www/html
A /var/www/html/do180.html
```

The previous output lists the directories and files that were changed or added to the **official-httdp** container. Remember that these changes are only for this container.

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

- 4.1. Stop the **official-httdp** container.

```
[student@workstation ~]$ docker stop official-httdp
official-httdp
```

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

```
[student@workstation ~]$ docker commit \
-a 'Your Name' \
-m 'Added do180.html page'
```

official-httdp

sha256:6ad5919cb2cf89843c2c15b429c2eab29358887421d97f5e35a6b7fa35576888

- 4.3. List the available container images.

```
[student@workstation ~]$ docker images
```

The expected output is similar to the following:

REPOSITORY	TAG	
<none>	<none>	
registry.lab.example.com/rhscl/httpd-24-rhel7	latest	
IMAGE ID	CREATED	SIZE
6ad5919cb2cf	25 seconds ago	305 MB
1c0961bdb0f3	4 weeks ago	305 MB

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

- 4.4. The new container image has neither a name, as listed in the **REPOSITORY** column, nor a tag. Tag the image with a custom name of **do180/custom-httdp**.

```
[student@workstation ~]$ docker tag 6ad5919cb2cf do180/custom-httdp
```

**NOTE**

The container image ID, **6ad5919cb2cf**, is the truncated version of the ID returned by the previous step.

- 4.5. List the available container images again to ensure that the name and tag were applied to the correct image.

```
[student@workstation ~]$ docker images
```

The expected output is similar to the following:

REPOSITORY	TAG	
do180/custom-httdp	latest	
registry.lab.example.com/rhscl/httpd-24-rhel7	latest	
IMAGE ID	CREATED	SIZE
6ad5919cb2cf	6 minutes ago	305 MB
1c0961bdb0f3	4 weeks ago	305 MB

- 5. Publish the saved container image to the Docker registry.

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

```
[student@workstation ~]$ docker tag do180/custom-httdp \
```

```
registry.lab.example.com/do180/custom-httd:v1.0
```

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

```
[student@workstation ~]$ docker images
```

REPOSITORY	TAG
do180/custom-httd	latest
registry.lab.example.com/do180/custom-httd	v1.0
registry.lab.example.com/rhscl/httpd-24-rhel7	latest
IMAGE ID	CREATED
6ad5919cb2cf	8 minutes ago
6ad5919cb2cf	8 minutes ago
1c0961bdb0f3	4 weeks ago
	SIZE
6ad5919cb2cf	305 MB
6ad5919cb2cf	305 MB
1c0961bdb0f3	305 MB

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

```
[student@workstation ~]$ docker push \
registry.lab.example.com/do180/custom-httd:v1.0
... output omitted ...
8049b1db64b5: Mounted from rhscl/httpd-24-rhel7
v1.0: digest:
sha256:804109820ce13f540f916fabc742f0a241f6a3647723bcd0f4a83c39b26580f7 size:
1368
```



NOTE

Each student only has access to their own private registry, it is therefore impossible that they will interfere with each. 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 **docker-registry-cli** command.

```
[student@workstation ~]$ docker-registry-cli \
registry.lab.example.com search custom-httd ssl
```

```
available options:-
```

```
-----
1) Name: do180/custom-httd
Tags: v1.0
```

```
1 images found !
```

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

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

```
[student@workstation ~]$ docker run -d --name test-httd -p 8280:8080 \
```

```
do180/custom-httdp:v1.0
```

```
5ada5f7e53d1562796b4848928576c1c35df120faee38805a7d1e7e6c43f8e3f
```

- 6.2. Use the **curl** to access the HTML page. Make sure to use the port 8280.

The HTML page created in the previous step should be displayed.

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

- 7. Grade your work. From the **workstation** VM, run the **lab container-images-manipulating grade** command.

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

- 8. Delete the containers and images created in this exercise.

- 8.1. Use the **docker stop** command to stop the running containers.

```
[student@workstation ~]$ docker stop test-httdp
test-httdp
```

- 8.2. Remove the container images.

```
[student@workstation ~]$ docker rm official-httdp test-httdp
official-httdp
test-httdp
```

- 8.3. Delete the exported container image.

```
[student@workstation ~]$ docker rmi do180/custom-httdp
Untagged: do180/custom-httdp:latest
Untagged: registry.lab.example.com/do180/custom-httdp@sha256:(...)
```

Remove the committed container image.

```
[student@workstation ~]$ docker rmi \
registry.lab.example.com/do180/custom-httdp:v1.0
Untagged: registry.lab.example.com/do180/custom-httdp:v1.0
... output omitted ...
```

- 8.4. Remove the **rhscl/httpd-24-rhel7** container image:

```
[student@workstation ~]$ docker rmi rhscl/httpd-24-rhel7
Untagged: rhscl/httpd-24-rhel7:latest
... output omitted ...
```

Cleanup

Clean your exercise by running the following command.

```
[student@workstation ~]$ lab container-images-manipulating gradeclean
```

This concludes the guided exercise.

▶ LAB

MANAGING IMAGES

PERFORMANCE CHECKLIST

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

RESOURCES	
Application URL:	http://127.0.0.1:8080 , http://127.0.0.1:8280
Resources	Nginx image

OUTCOMES

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

To set the environment for this lab, run the following command. The script ensures the Docker daemon is running on **workstation**.

```
[student@workstation ~]$ lab container-images-lab setup
```

- From **workstation**, open a terminal and use the **docker-registry-cli** command to locate the **nginx** image and pull it into your local Docker cache.



NOTE

There are three Nginx images in the repository, and two of them were retrieved from the Red Hat Software Collection Library (*rh scl*). The Nginx container image for this lab is the same as the one that is available at [docker .io](https://hub.docker.com/_/nginx).

Ensure that the image has been successfully retrieved.

- Start a new container using the Nginx image, according to the specifications listed in the following list. Forward the server port 80 to port 8080 from the host using the **-p** option.
 - Name:** `official-nginx`
 - Run as daemon:** yes
 - Container image:** `nginx`
 - Port forward:** from host port 8080 to container port 80.
- Log in to the container using the **exec** verb and update the content **index.html** file with **DO180 Page**. The web server directory is located at **/usr/share/nginx/html**. Once the page is updated, exit the container and use the **curl** command to access the web page.
- 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**. Use the following specifications:

- Image name: **do180/mynginx**
 - Image tag: **v1.0**
 - Author name: *your name*
 - Commit message: **Changed index.html page**
5. 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
- From **workstation**, use the **curl** command to access the web server, accessible from the port 8280.
6. Grade your work. From the **workstation** VM, run the **lab container-images-manipulating grade** command.

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

7. Stop the running container and delete the two containers by using the **rm** option.

Cleanup

Clean your lab by running the following command.

```
[student@workstation ~]$ lab container-images-lab cleanup
```

This concludes the lab.

► SOLUTION

MANAGING IMAGES

PERFORMANCE CHECKLIST

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

RESOURCES	
Application URL:	http://127.0.0.1:8080 , http://127.0.0.1:8280
Resources	Nginx image

OUTCOMES

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

To set the environment for this lab, run the following command. The script ensures the Docker daemon is running on **workstation**.

```
[student@workstation ~]$ lab container-images-lab setup
```

- From **workstation**, open a terminal and use the **docker-registry-cli** command to locate the **nginx** image and pull it into your local Docker cache.



NOTE

There are three Nginx images in the repository, and two of them were retrieved from the Red Hat Software Collection Library (*rhscl*). The Nginx container image for this lab is the same as the one that is available at [docker .io](https://docker.io).

Ensure that the image has been successfully retrieved.

- Open a terminal on the workstation VM (Applications → System Tools → Terminal) and pull the HTTPd container image.
- Use the **docker-registry-cli** command to search for the Nginx container image **nginx**.

```
[student@workstation ~]$ docker-registry-cli \
registry.lab.example.com search nginx ssl
-----
1) Name: nginx
Tags: latest

1 images found !
```

- Pull the Nginx container image by using the **docker pull** command.

```
[student@workstation ~]$ docker pull nginx
... output omitted ...
```

```
Status: Downloaded newer image for registry.lab.example.com/nginx:latest
```

- 1.4. Ensure that the container image is available in the cache by running the **docker images** command.

```
[student@workstation ~]$ docker images
```

This command produces output similar to the following:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
registry.lab.example.com/nginx	latest	cd5239a0906a	8 days ago	109 MB

2. Start a new container using the Nginx image, according to the specifications listed in the following list. Forward the server port 80 to port 8080 from the host using the **-p** option.

- **Name:** **official-nginx**
- **Run as daemon:** yes
- **Container image:** **nginx**
- **Port forward:** from host port 8080 to container port 80.

- 2.1. From **workstation**, use the **docker run** command to create a container. Give the container a name of **official-nginx**.

```
[student@workstation ~]$ docker run \
--name official-nginx \
-d -p 8080:80 \
registry.lab.example.com/nginx
02dbc348c7dcf8560604a44b11926712f018b0ac44063d34b05704fb8447316f
```

3. Log in to the container using the **exec** verb and update the content **index.html** file with **D0180 Page**. The web server directory is located at **/usr/share/nginx/html**.

Once the page is updated, exit the container and use the **curl** command to access the web page.

- 3.1. Log in to the container by using the **docker exec** command.

```
[student@workstation ~]$ docker 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 Page**.

```
root@f22c60d901fa:/# echo 'D0180 Page' > /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 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**. Use the following specifications:

- Image name: **do180/mynginx**
- Image tag: **v1.0**
- Author name: *your name*
- Commit message: **Changed index.html page**

4.1. Use the **docker stop** to stop the **official-nginx** container.

```
[student@workstation ~]$ docker stop official-nginx
official-nginx
```

4.2. Commit your changes to a new container image. Use your name as the author of the changes, with a commit message of **Changed index.html page**.

```
[student@workstation ~]$ docker commit -a 'Your Name' \
-m 'Changed index.html page' \
official-nginx
sha256:d77b234dec1cc96df8d6ce95d395c8fc69664ab70f412557a54fffffebd180d6
```

**NOTE**

Note the container ID that is returned. This ID will be required for the image tagging.

4.3. List the available container images in order to locate your newly created image.

```
[student@workstation ~]$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
<none>	<none>	d77b234dec1c	7 seconds ago	109 MB
registry.lab.example.com/nginx	latest	cd5239a0906a	8 days ago	109 MB

4.4. Set the name and tag for the new image. Pass the ID to the **docker tag** command.

```
[student@workstation ~]$ docker tag d77b234dec1c \
do180/mynginx:v1.0
```

**NOTE**

The *d77b234dec1c* ID corresponds to the truncated version of the ID returned by ???.

5. 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

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

- 5.1. Use the **docker run** command to create the **my-nginx** container, according to the specifications.

```
[student@workstation ~]$ docker run -d \
--name my-nginx \
-p 8280:80 \
do180/mynginx:v1.0
c1cba44fa67bf532d6e661fc5e1918314b35a8d46424e502c151c48fb5fe6923
```

- 5.2. 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
```

6. Grade your work. From the **workstation** VM, run the **lab container-images-manipulating grade** command.

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

7. Stop the running container and delete the two containers by using the **rm** option.

- 7.1. Stop the **my-nginx** container.

```
[student@workstation ~]$ docker stop my-nginx
```

- 7.2. Delete the two containers.

```
[student@workstation ~]$ docker rm my-nginx official-nginx
```

- 7.3. Delete the container images.

```
[student@workstation ~]$ docker rmi do180/mynginx:v1.0
```

Cleanup

Clean your lab by running the following command.

```
[student@workstation ~]$ lab container-images-lab cleanup
```

This concludes the lab.

SUMMARY

In this chapter, you learned:

- Registries must be used to pull and push container images to private registries for internal use, or to public registries for outside consumption.
 - The Red Hat Software Collections Library (RHSCL) provides tested and certified images at `registry.access.redhat.com`.
 - Docker daemons support extra registries by editing the `/etc/sysconfig/docker` file and by adding new registries to the `ADD_REGISTRY` variable.
 - In order to support registries with self-signed certificates, add the registry to the `INSECURE_REGISTRY` variable of the `/etc/sysconfig/docker` configuration file.
 - Registries implement a RESTful API to pull, push, and manipulate objects. The API is used by the Docker daemon, but it can also be queried via tools such as `curl`.
 - To search for an image in a public registry, use the `docker search` command.
 - To search for an image in a private registry, use the `docker-registry-cli` command.
 - To pull an image from a registry, use the `docker pull` command.
 - Registries use tags as a mechanism to support multiple image releases.
- The Docker daemon supports export and import procedures for image files using the `docker export`, `docker import`, `docker save`, and `docker load` commands.
 - For most scenarios, using `docker save` and `docker load` command is the preferred approach.
 - The Docker daemon cache can be used as a staging area to customize and push images to a registry.
 - Docker also supports container image publication to a registry using the `docker push` command.
 - Container images from a daemon cache can be removed using the `docker rmi` command.

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

- Design Considerations for Custom Container Images (and Quiz)
- Building Custom Container Images with Dockerfile (and Guided Exercise)

LAB

- Creating Custom Container Images

DESIGN CONSIDERATIONS FOR 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

Two common ways of building a new container image are as follows:

1. Run operating system commands inside a container and then commit the image.
2. Use a Dockerfile that invokes operating system commands and uses an operating system image as the parent.

Both methods involve extra effort when the same runtimes and libraries are required by different application images. There is not much to be done to improve the first option, but the second option can be improved by selecting a better parent image.

Many popular application platforms are already available in public image registries like Docker Hub. 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 to create a Dockerfile for building a child image from an existing container image include:

- Add new runtime libraries, such as database connectors.
- Include organization-wide customizations 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 libraries).
- Lock either the parent image or some included software package to a specific release to lower the risk related to future software updates.

Two sources of container images to use either as parent images or for changing their Dockerfiles are the Docker Hub and the Red Hat Software Collections Library (RHSCL).

WORKING WITH THE RED HAT SOFTWARE COLLECTIONS LIBRARY

The *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, and which 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.

The RHSCl allows software developers to use the latest version without impacting RHEL, because the 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

The 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 the RHSCl Dockerfiles and related sources in the *rhscl-dockerfiles* package available from the RHSCl repository. Community users can get the Dockerfiles for CentOS-based equivalent container images from <https://github.com/sclorg?q=-container>.

Many RHSCl container images include support for *Source-to-Image (S2I)* which is best known as an OpenShift Container Platform feature. Having support for S2I does not affect the usage 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. The 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, you can access information about each image, such as its version, contents, and usage.

FINDING DOCKERFILES ON DOCKER HUB

The Docker Hub web site is a popular search site for container images. 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 5.5/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 5.5 **Dockerfile** tree is <https://github.com/docker-library/mysql/blob/master/5.5/>.

USING 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 that follows a standard directory structure and provides 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 MacOS, 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>

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 Docker community? (Choose one.)

- a. Run commands inside basic OS containers, 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 `docker 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 Docker 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 the typical scenarios for creating a Dockerfile to build a child image from an existing image (Choose three):

- a. Adding new runtime libraries.
- b. Setting constraints to the container's access to the host machine's CPU.
- c. Including organization-wide customizations, for example, SSL certificates and authentication providers.
- d. 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 Docker community? (Choose one.)**

- a. Run commands inside basic OS containers, 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 `docker 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 Docker 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 the typical scenarios for creating a Dockerfile to build a child image from an existing image (Choose three):**

- a. Adding new runtime libraries.
- b. Setting constraints to the container's access to the host machine's CPU.
- c. Including organization-wide customizations, for example, SSL certificates and authentication providers.
- d. Adding internal libraries, to be shared as a single image layer by multiple container images for different applications.

BUILDING CUSTOM CONTAINER IMAGES WITH DOCKERFILE

OBJECTIVES

After completing this section, students should be able to create a container image using common Dockerfile commands.

BASE CONTAINERS

A **Dockerfile** is the mechanism that the Docker packaging model provides 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** specification.
3. Build the image with the **docker** command.

Create a Working Directory

The **docker** command can use the files in a working directory to build an image. An empty working directory should be created to keep from 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 should exist in the working directory. The basic syntax of a **Dockerfile** is shown below:

```
# Comment  
INSTRUCTION arguments
```

Lines that begin with a pound sign (#) are comments. Inline comments are not supported. **INSTRUCTION** is a **Dockerfile** keyword. Keywords are not case-sensitive, but a common convention is to make instructions all uppercase to improve visibility.

Instructions in a Dockerfile are executed in the order they appear. The first non-comment instruction must be a **FROM** instruction to specify the base image to build upon. Each Dockerfile instruction is run independently (so **RUN cd /var/tmp** will not have an effect on the commands that follow).

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 pound sign (#) are comments.
- ② The new container image will be constructed using **rhel7:7.5** container base image. You 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. Red Hat highly recommends that you use these container images as a base.
- ③ **LABEL** is responsible for adding generic metadata to an image. A **LABEL** is a simple key/value pair.
- ④ **MAINTAINER** is responsible for setting the **Author** field of the generated container image. You can use the **docker inspect** command to view image metadata.
- ⑤ **RUN** executes commands in a new layer on top of the current image, then commits the results. The committed result is used in the next step in the **Dockerfile**. 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 **docker run** command exposes a port from the host and the port does not need to be listed in an **EXPOSE** instruction.
- ⑦ **ENV** is responsible for defining environment variables that will be available to 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** copies files from local or remote source and adds them to the container's file system.
- ⑨ **COPY** also copies files from local source 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 in the **Dockerfile**. 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 container is created. By default, the command that is executed is **/bin/sh -c** unless an **ENTRYPOINT** is specified.
- ⑫ **CMD** provides the default arguments for the **ENTRYPOINT** instruction.

USING CMD AND ENTRYPOINT INSTRUCTIONS IN THE DOCKERFILE

The **ENTRYPOINT** and **CMD** instructions have two formats:

- Using a **JSON** array:

```
ENTRYPOINT ["command", "param1", "param2"]
```

```
CMD ["param1", "param2"]
```

This is the preferred form.

- Using a shell form:

```
ENTRYPOINT command param1 param2
```

```
CMD param1 param2
```

The **Dockerfile** should contain at most one **ENTRYPOINT** and one **CMD** instruction. If more than one of each is written, then only the last instruction takes effect. Because the default **ENTRYPOINT** is **/bin/sh -c**, a **CMD** can be passed in without specifying an **ENTRYPOINT**.

The **CMD** instruction can be overridden when starting a container. For example, the following instruction causes any container that is run to display the current time:

```
ENTRYPOINT ["/bin/date", "+%H:%M"]
```

The following example provides the same functionality, with the added benefit of being overwritable when a container is started:

```
ENTRYPOINT ["/bin/date"]
CMD ["+%H:%M"]
```

When a container is started without providing a parameter, the current time is displayed:

```
[student@workstation ~]$ docker run -it do180/rhel
11:41
```

If a parameter is provided after the image name in the **docker run** command, it overwrites the **CMD** instruction. For example, the following command will display the current day of the week instead of the time:

```
[student@workstation demo-basic]$ docker run -it do180/rhel +%A
Tuesday
```

As previously mentioned, because the default **ENTRYPOINT** is **/bin/sh -c**, the following instruction also displays the current time, with the added benefit of being able to be overridden at run time.

```
CMD ["date", "+%H:%M"]
```

USING ADD AND COPY INSTRUCTIONS IN THE DOCKERFILE

The **ADD** and **COPY** instructions have two forms:

- Using a shell form:

```
ADD <source>... <destination>
```

```
COPY <source>... <destination>
```

- Using a **JSON** array:

```
ADD [<source>, ... <destination>]"
```

```
COPY ["<source>", ... "<destination>"]
```

The **source** path must be inside the same folder as the **Dockerfile**. The reason for this is that the first step of a **docker build** command is to send all files from the **Dockerfile** folder to the **docker** daemon, and the **docker** daemon cannot see folders or files that are in another folder.

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, 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 the permissions, and with **root** as the owner, even if the **USER** instruction is specified. Red Hat recommends that you use a **RUN** instruction after the copy to change the owner and avoid "permission denied" errors.

IMAGE LAYERING

Each instruction in a **Dockerfile** creates a new 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, because three layers are created for a single purpose. Red Hat recommends that you minimize the number of layers. You can achieve the same objective 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** is compromised, but it can be easily fixed:

```
RUN yum --disablerepo=* --enablerepo="rhel-7-server-rpms" && \  
yum update -y && \  
yum install -y httpd
```

The example creates only one layer and the readability is not compromised. This layering concept also applies to instructions such as **ENV** and **LABEL**. To specify multiple **LABEL** or **ENV** instructions, Red Hat recommends that you use only one instruction per line, and separate each key-value pair with an equals sign (=):

```
LABEL version="2.0" \  
description="This is an example container image" \  
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 THE DOCKER COMMAND

The **docker build** command processes the **Dockerfile** and builds a new image based on the instructions it contains. The syntax for this command is as follows:

```
$ docker build -t NAME:TAG DIR
```

DIR is the path to the working directory. It can be the current directory as designated by a period (.) if the working directory is the current directory of the **shell**. *NAME:TAG* is a name with a tag that is assigned to the new image. It is specified with the **-t** option. If the *TAG* is not specified, then the image is automatically tagged as **latest**.

DEMONSTRATION: BUILDING A SIMPLE CONTAINER

1. Briefly review the provided **Dockerfile** by opening it in a text editor:

```
[student@workstation ~]$ vim /home/student/DO180/labs/simple-container/Dockerfile
```

2. Observe the **FROM** instruction on the first line of the **Dockerfile**:

```
FROM rhel7:7.5
```

rhel7:7.5 is the base image from which the container is built and where subsequent instructions in the **Dockerfile** are executed.

3. Observe the **MAINTAINER** instruction:

```
MAINTAINER username <username@example.com>
```

The **MAINTAINER** instruction generally indicates the author of the **Dockerfile**. It sets the *Author* field of the generated image. You can run the **docker inspect** command on the generated image to view the *Author* field.

4. Observe the **LABEL** instruction, which sets multiple key-value pair labels as metadata for the image:

```
LABEL version="1.0" \
      description="This is a simple container image" \
      creationDate="31 March 2018"
```

You can run the **docker inspect** command to view the labels in the generated image.

5. Observe the **ENV** instruction, which injects environment variables into the container at runtime. These environment variables can be overridden in the **docker run** command:

```
ENV VAR1="hello" \
      VAR2="world"
```

- In the **ADD** instruction, the **training.repo** file that points to the classroom yum repository, is copied to **/etc/yum.repos.d/training.repo** from the current working directory:

```
ADD training.repo /etc/yum.repos.d/training.repo
```

**NOTE**

The **training.repo** file configures **yum** to use the local repository instead of relying on the default Red Hat Yum repositories.

- Observe the **RUN** instruction where the **yum update** command is executed, and the *bind-utils* package is installed in the container image:

```
RUN yum install -y bind-utils && \
yum clean all
```

The **yum update** command updates the RHEL 7.5 operating system, while the second command installs the DNS utility package *bind-utils*. Notice that both commands are executed with a single **RUN** instruction. Each **RUN** instruction in a **Dockerfile** creates a new image layer to execute the subsequent commands. Minimizing the number of **RUN** commands therefore makes for less overhead when actually running the container.

- Save the **Dockerfile** and run the following commands in the terminal to begin building the new image:

```
[student@workstation ~]$ cd /home/student/D0180/labs/simple-container
[student@workstation simple-container]$ docker build -t do180/rhel .
Sending build context to Docker daemon 3.584 kB
Step 1/6 : FROM rhel7:7.5
Trying to pull repository registry.lab.example.com/rhel7 ...
7.5: Pulling from registry.lab.example.com/rhel7
845d34b6bc6a: Pull complete
c7cbbc13fe2e: Pull complete
Digest: sha256:ca84777dbf5c4574f103c5e9f270193eb9bf8732198e86109c5aa26fdb6fdb0b
Status: Downloaded newer image for registry.lab.example.com/rhel7:7.5
--> 93bb76ddeb7a
Step 2/6 : MAINTAINER John Doe <jdoe@abc.com>
--> Running in a52aa1d5e94b
--> 78ea134831fc
Removing intermediate container a52aa1d5e94b
Step 3/6 : LABEL version "1.0" description "This is a simple container image"
creationDate "31 March 2017"
--> Running in 37871b3a4d69
--> a83e12fc5f8a
Removing intermediate container 37871b3a4d69
Step 4/6 : ENV VAR1 "hello" VAR2 "world"
--> Running in 83d6b5c8546f
--> cc0de1e2423a
Removing intermediate container 83d6b5c8546f
Step 5/6 : ADD training.repo /etc/yum.repos.d/training.repo
--> d6193393a573
Removing intermediate container f08a97a2851b
Step 6/6 : RUN yum install -y bind-utils && yum clean all
--> Running in dc57663d28b1
...
```

```

Installed:
bind-utils.x86_64 32:9.9.4-61.el7

Dependency Installed:
GeoIP.x86_64 0:1.5.0-11.el7           bind-libs.x86_64 32:9.9.4-61.el7
bind-license.noarch 32:9.9.4-61.el7

Complete!
Loaded plugins: ovl, product-id, search-disabled-repos, subscription-manager
This system is not receiving updates. You can use subscription-manager on the host
to register and assign subscriptions.
Cleaning repos: rhel-7-server-optional-rpms rhel-server-rhscl-7-rpms rhel_dvd
Cleaning up everything
Maybe you want: rm -rf /var/cache/yum, to also free up space taken by orphaned
data from disabled or removed repos
--> 93b8610f643e
Removing intermediate container dc57663d28b1
Successfully built 93b8610f643e

```

- After the build completes, run the **docker images** command to verify that the new image is built successfully.

```

[student@workstation simple-container]$ docker images
REPOSITORY      TAG      IMAGE ID      CREATED      SIZE
do180/rhel      latest   93b8610f643e   About a minute ago   381 MB
...

```

- Inspect the created image to verify that the **MAINTAINER** and **LABEL Dockerfile** instructions indeed manipulated the metadata of the image:

```

[student@workstation simple-container]$ docker inspect do180/rhel | grep Author
    "Author": "John Doe <jdoe@abc.com>",
[student@workstation simple-container]$ docker inspect do180/rhel \
| grep 'version\|description\|creationDate'
    "creationDate": "31 March 2018",
    "description": "This is a simple container image",
    "version": "1.0"
...

```

- Execute the following command to run a new container from the generated image with an interactive Bash terminal:

```

[student@workstation simple-container]$ docker run --name simple-container \
-it do180/rhel /bin/bash

```

- In the RHEL 7.5 container, verify that the environment variables set in the **Dockerfile** are injected into the container:

```

[root@5c62bfb50dc8 /]# env | grep 'VAR1\|VAR2'
VAR1=hello
VAR2=world

```

- Execute a **dig** command from the container to verify that the *bind-utils* package is installed and working correctly:

```
[root@5c62bfb50dc8 /]# dig materials.example.com
; <>> DiG 9.9.4-RedHat-9.9.4-37.el7 <>> materials.example.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 20867
;; flags: qr aa rd ra ad; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
;materials.example.com. IN A

;; ANSWER SECTION:
materials.example.com. 3600 IN A 172.25.254.254

;; Query time: 0 msec
;; SERVER: 172.25.250.254#53(172.25.250.254)
;; WHEN: Fri Mar 31 16:33:38 UTC 2017
;; MSG SIZE rcvd: 55
```

Examine the output to confirm that the answer section provides the IP address of the classroom materials server.

14. Exit from the Bash shell in the container:

```
[root@5c62bfb50dc8 thu /]# exit
```

Exiting Bash terminates the running container.

15. Remove the **simple-container** container:

```
[student@workstation simple-container]$ docker rm simple-container
```

16. Remove the **do180/rhel** container image:

```
[student@workstation simple-container]$ docker rmi do180/rhel
```

17. Remove the intermediate container images generated by the **docker build** command.

```
[student@workstation simple-container]$ docker rmi -f $(docker images -q)
```

This concludes the demonstration.



REFERENCES

OpenShift Container Platform documentation: Creating Images

https://docs.openshift.com/container-platform/3.9/creating_images

Dockerfile Reference Guide

<https://docs.docker.com/engine/reference/builder/>

Creating base images

<https://docs.docker.com/engine/userguide/eng-image/baseimages/>

Implementing a base image based on RHEL-based distros

<https://github.com/docker/docker/blob/master/contrib/mkimage-yum.sh>

► GUIDED EXERCISE

CREATING A BASIC APACHE CONTAINER IMAGE

In this exercise, you will create a basic Apache container image.

RESOURCES

Files:	/home/student/D0180/labs/basic-apache/ Dockerfile
Application URL:	http://127.0.0.1:10080

OUTCOMES

You should be able to create a basic Apache container image built on a RHEL 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 basic-apache setup
```

► 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/basic-apache/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 basic Apache container on RHEL 7"
```

- 1.5. Add a **RUN** instruction with a **yum install** command to install Apache on the new container:

```
ADD training.repo /etc/yum.repos.d/training.repo
RUN yum -y update && \
    yum install -y httpd && \
    yum clean all
```

NOTE

The **ADD** instruction configures **yum** to use the local repository instead of relying on the default Red Hat Yum repositories.

- 1.6. 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 more as metadata about which ports the container is listening to.

- 1.7. At the end of the file, use the following **CMD** instruction to set **httpd** as the default executable when the container is run:

```
CMD ["httpd", "-D", "FOREGROUND"]
```

- 1.8. 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 -y update && \
    yum install -y httpd && \
    yum clean all

EXPOSE 80

CMD ["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/DO180/labs/basic-apache
[student@workstation basic-apache]$ docker build -t do180/apache .
Sending build context to Docker daemon 3.584 kB
```

```

Step 1/7 : FROM rhel7:7.5
Trying to pull repository registry.lab.example.com/rhel7 ...
7.5: Pulling from registry.lab.example.com/rhel7
845d34b6bc6a: Pull complete
c7cfc13fe2e: Pull complete
Digest: sha256:ca84777dbf5c4574f103c5e9f270193eb9bf8732198e86109c5aa26fdb6fdb0b
Status: Downloaded newer image for registry.lab.example.com/rhel7:7.5
--> 93bb76ddeb7a
Step 2/7 : MAINTAINER Your Name <youremail>
--> Running in 3d4d201a3730
--> 6c0c7ca9d614
Removing intermediate container 3d4d201a3730
Step 3/7 : LABEL description "A basic Apache container on RHEL 7"
--> Running in 0e9ec16edf9b
--> 0541f6f3d393
Removing intermediate container 0e9ec16edf9b
Step 4/7 : ADD training.repo /etc/yum.repos.d/training.repo
--> 642dc842d870
Removing intermediate container 33e2dc33f198
Step 5/7 : RUN yum install -y httpd && yum clean all
--> Running in 50e68830492b
...
Installed:
httpd.x86_64 0:2.4.6-80.el7
...
Complete!
...
Maybe you want: rm -rf /var/cache/yum, to also free up space taken by orphaned
data from disabled or removed repos
--> 550f4c27c866
Removing intermediate container 50e68830492b
Step 6/7 : EXPOSE 80
--> Running in 63c30a79a047
--> 517c0dd6f2ae
Removing intermediate container 63c30a79a047
Step 7/7 : CMD httpd -D FOREGROUND
--> Running in 612a9802c872
--> 6b291a377825
Removing intermediate container 612a9802c872
Successfully built 6b291a377825

```

- 2.2. After the build process has finished, run **docker images** to see the new image in the image repository:

```
[student@workstation basic-apache]$ docker images
REPOSITORY      TAG      IMAGE ID      CREATED      SIZE
do180/apache    latest   6b291a377825   4 minutes ago   393 MB
...
```

► 3. Run the Apache Container

- 3.1. Use the following command to run a container using the Apache image:

```
[student@workstation basic-apache]$ docker run --name lab-apache \
-d -p 10080:80 do180/apache
```

```
693da7820edb...
```

- 3.2. Run the **docker ps** command to see the running container:

```
[student@workstation basic-apache]$ docker ps
CONTAINER ID        IMAGE               COMMAND                  ...   PORTS
NAMES
693da7820edb      do180/apache     "httpd -D FOREGROUND"    0.0.0.0:10080->80/
tcp  lab-apache
```

- 3.3. Use **curl** command to verify that the server is running:

```
[student@workstation basic-apache]$ curl 127.0.0.1:10080
```

If the server is running, you should see HTML output for an Apache server test home page.

- 4. Run the following command to verify that the image was correctly built:

```
[student@workstation basic-apache]$ lab basic-apache grade
```

- 5. Stop and then remove the **lab-apache** container:

```
[student@workstation basic-apache]$ docker stop lab-apache
[student@workstation basic-apache]$ docker rm lab-apache
```

- 6. Remove the **do180/apache** container image:

```
[student@workstation basic-apache]$ docker rmi -f do180/apache
```

- 7. Remove the intermediate container images generated by the **docker build** command.

```
[student@workstation simple-container]$ docker rmi -f $(docker images -q)
```

This concludes the guided exercise.

► LAB

CREATING CUSTOM CONTAINER IMAGES

PERFORMANCE CHECKLIST

In this lab, you will create a Dockerfile to run an Apache Web Server container that hosts a static HTML file. The Apache image will be based on a base RHEL 7.5 image that serves a custom **index.html** page.

RESOURCES

Files: /home/student/D0180/labs/custom-images/

Application URL: http://127.0.0.1:20080

OUTCOMES

You should be able to create a custom Apache Web Server container that hosts static HTML files.

Run the following command to download the relevant lab files and to verify that there are no running or stopped containers that will interfere with completing the lab:

```
[student@workstation ~]$ lab custom-images setup
```

1. Review the provided Dockerfile stub in the **/home/student/D0180/labs/custom-images/** folder. Edit the **Dockerfile** and ensure that it meets the following specifications:
 - Base image: **rhel7:7.5**
 - Set the desired author name and email ID with **MAINTAINER** instruction.
 - Environment variable: **PORT** set to 8080
 - Update the RHEL packages and install Apache (**httpd** package) using the classroom Yum repository. Also ensure 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** (**/var/www/html/**) inside the container.

The **src** folder contains a single **index.html** file that prints a **Hello World!** message.

- Start Apache **httpd** 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/DO180/labs/custom-images/** folder.
- 1.2. Set the base image for the Dockerfile to **rhel7:7.5**.
- 1.3. Set your name and email with a **MAINTAINER** instruction.
- 1.4. Create an environment variable called **PORT** and set it to 8080.
- 1.5. Add the classroom Yum repositories configuration file. Update the RHEL packages and install Apache with a single **RUN** instruction.
- 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.
- 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.
- 1.8. Copy all the files from the **src** folder to the Apache **DocumentRoot** path at **/var/www/html**.
- 1.9. Finally, insert a **CMD** instruction to run **httpd** in the foreground, and then save the Dockerfile.
2. Build the custom Apache image with the name **do180/custom-apache**.
 - 2.1. Verify the Dockerfile for the custom Apache image.
 - 2.2. Run a **docker build** command to build the custom Apache image and name it **do180/custom-apache**.
 - 2.3. Run the **docker images** command to verify that the custom image is built successfully.
3. Create a new container in detached mode with the following characteristics:
 - **Name: lab-custom-images**
 - **Container image: do180/custom-apache**
 - **Port forward:** from host port 20080 to container port 8080
 - 3.1. Create and run the container.
 - 3.2. Verify that the container is ready and running.
4. Verify that the server is running and that it is serving the HTML file.
 - 4.1. Run a **curl** command on **127.0.0.1:20080**
5. Verify that the lab was correctly executed. Run the following from a terminal:

```
[student@workstation custom-images]$ lab custom-images grade
```

6. Stop and then remove the container started in this lab.
7. Remove the custom container image created in this lab.

8. Remove the intermediate container images generated by the **docker build** command.

```
[student@workstation simple-container]$ docker rmi -f $(docker images -q)
```

This concludes the lab.

► SOLUTION

CREATING CUSTOM CONTAINER IMAGES

PERFORMANCE CHECKLIST

In this lab, you will create a Dockerfile to run an Apache Web Server container that hosts a static HTML file. The Apache image will be based on a base RHEL 7.5 image that serves a custom `index.html` page.

RESOURCES

Files:	/home/student/D0180/labs/custom-images/
Application URL:	http://127.0.0.1:20080

OUTCOMES

You should be able to create a custom Apache Web Server container that hosts static HTML files.

Run the following command to download the relevant lab files and to verify that there are no running or stopped containers that will interfere with completing the lab:

```
[student@workstation ~]$ lab custom-images setup
```

1. Review the provided Dockerfile stub in the `/home/student/D0180/labs/custom-images/` folder. Edit the **Dockerfile** and ensure that it meets the following specifications:

- Base image: **rhel7:7.5**
- Set the desired author name and email ID with **MAINTAINER** instruction.
- Environment variable: **PORT** set to 8080
- Update the RHEL packages and install Apache (`httpd` package) using the classroom Yum repository. Also ensure 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** (`/var/www/html/`) inside the container.

The `src` folder contains a single `index.html` file that prints a **Hello World!** message.

- Start Apache `httpd` in the foreground using the following command:

```
httpd -D FOREGROUND
```

1. Open a terminal (Applications → System Tools → Terminal) and use your preferred editor to modify the Dockerfile located in the **/home/student/D0180/labs/custom-images/** folder.

```
[student@workstation ~]$ cd /home/student/D0180/labs/custom-images/
[student@workstation custom-images]$ 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. Update the RHEL packages and install Apache 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 ${PORT}
```

- 1.8. Copy all the files from the **src** folder to the Apache **DocumentRoot** path at **/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.

```
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 ${PORT}

COPY ./src/ /var/www/html/

CMD ["httpd", "-D", "FOREGROUND"]
```

2.2. Run a **docker build** command to build the custom Apache image and name it **do180/custom-apache**.

```
[student@workstation custom-images]$ docker build -t do180/custom-apache .
Sending build context to Docker daemon 5.632 kB
Step 1/10 : FROM rhel7:7.5
--> 93bb76ddeb7a
Step 2/10 : MAINTAINER Your Name <youremail>
--> Running in fd2701e2dc31
--> 2a0cb88ba0d9
Removing intermediate container fd2701e2dc31
Step 3/10 : ENV PORT 8080
--> Running in 5274fb5b5017
--> 4f347c7508e3
Removing intermediate container 5274fb5b5017
Step 4/10 : ADD training.repo /etc/yum.repos.d/training.repo
--> a99cd105c046
Removing intermediate container 0e8c11d7a1f6
Step 5/10 : RUN yum install -y httpd && yum clean all
--> Running in 594f9e7b594a
...
Installed:
httpd.x86_64 0:2.4.6-80.el7
...
Complete!
...
--> dd9ed43b1683
Removing intermediate container 92baa6734de3
```

```

Step 6/10 : 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/
--> Running in 24911b35ea8c

--> 0e556d9cb284
Removing intermediate container 24911b35ea8c
Step 7/10 : USER apache
--> Running in 7f769d2f9eda
--> 95e648d72347
Removing intermediate container 7f769d2f9eda
Step 8/10 : EXPOSE ${PORT}
--> Running in de11b4f17026
--> b3ad5660808c
Removing intermediate container de11b4f17026
Step 9/10 : COPY ./src/ /var/www/html
--> 804879072dc7
Removing intermediate container 51526a53cf1b
Step 10/10 : CMD httpd -D FOREGROUND
--> Running in 2dfffc816173
--> 35175850d37d
Removing intermediate container 2dfffc816173
Successfully built 35175850d37d

```

- 2.3. Run the **docker images** command to verify that the custom image is built successfully.

```
[student@workstation custom-images]$ docker images
REPOSITORY          TAG      IMAGE ID      ...
do180/custom-apache latest   35175850d37d ...
registry.lab.example.com/rhel7    7.5     41a4953dbf95 ...
```

3. Create a new container in detached mode with the following characteristics:

- **Name: lab-custom-images**
- **Container image: do180/custom-apache**
- **Port forward:** from host port 20080 to container port 8080

- 3.1. Create and run the container.

```
[student@workstation custom-images]$ docker run --name lab-custom-images -d \
-p 20080:8080 do180/custom-apache
367823e35c4a...
```

- 3.2. Verify that the container is ready and running.

```
[student@workstation custom-images]$ docker ps
...   IMAGE           COMMAND            ... PORTS
NAMES
...   do180/custom-apache "httpd -D FOREGROUND" ...  0.0.0.0:20080->8080/tcp
lab-custom-images
```

- 4.** Verify that the server is running and that it is serving the HTML file.

4.1. Run a **curl** command on **127.0.0.1:20080**

```
[student@workstation custom-images]$ curl 127.0.0.1:20080
```

The output should be as follows:

```
<html>
<header><title>DO180 Hello!</title></header>
<body>
Hello World! The custom-images lab works!
</body>
</html>
```

- 5.** Verify that the lab was correctly executed. Run the following from a terminal:

```
[student@workstation custom-images]$ lab custom-images grade
```

- 6.** Stop and then remove the container started in this lab.

```
[student@workstation custom-images]$ docker stop lab-custom-images
[student@workstation custom-images]$ docker rm lab-custom-images
```

- 7.** Remove the custom container image created in this lab.

```
[student@workstation custom-images]$ docker rmi -f do180/custom-apache
```

- 8.** Remove the intermediate container images generated by the **docker build** command.

```
[student@workstation simple-container]$ docker rmi -f $(docker images -q)
```

This concludes the lab.

SUMMARY

In this chapter, you learned:

- The usual method of creating container images is using Dockerfiles.
- Dockerfiles provided by Red Hat or Docker Hub are a good starting point for creating custom images using a specific language or technology.
- The Source-to-Image (S2I) process provides an alternative to Dockerfiles. S2I spares developers the need to learn low-level operating system commands and usually generates smaller images.
- Building an image from a Dockerfile is a three-step process:
 1. Create a working directory.
 2. Write the **Dockerfile** specification.
 3. Build the image using the **docker build** command.
- The **RUN** instruction is responsible for modifying image contents.
- The following instructions are responsible for adding metadata to images:
 - **LABEL**
 - **MAINTAINER**
 - **EXPOSE**
- The default command that runs when the container starts can be changed with the **RUN** and **ENTRYPOINT** instructions.
- The following instructions are responsible for managing the container environment:
 - **WORKDIR**
 - **ENV**
 - **USER**
- The **VOLUME** instruction creates a mount point in the container.
- The **Dockerfile** provides two instructions to include resources in the container image:
 - **ADD**
 - **COPY**

CHAPTER 6

DEPLOYING CONTAINERIZED APPLICATIONS ON OPENSHIFT

GOAL

Deploy single container applications on OpenShift Container Platform.

OBJECTIVES

- Install the OpenShift CLI tool and execute basic commands.
- Create standard Kubernetes resources.
- Build an application using the Source-to-Image facility of OpenShift.
- Create a route to a service.
- Create an application using the OpenShift web console.

SECTIONS

- Installing the OpenShift Command-line Tool (and Quiz)
- Creating Kubernetes Resources (and Guided Exercise)
- Creating Applications with the Source-to-Image Facility (and Guided Exercise)
- Creating Routes (and Guided Exercise)
- Creating Applications with the OpenShift Web Console (and Guided Exercise)

LAB

- Deploying Containerized Applications on OpenShift

INSTALLING THE OPENSHIFT COMMAND-LINE TOOL

OBJECTIVES

After completing this section, students should be able to install the OpenShift CLI tool and execute basic commands.

INTRODUCTION

OpenShift Container Platform (OCP) ships with a command-line tool that enables system administrators and developers to work with an OpenShift cluster. The **oc** command-line tool provides the ability to modify and manage resources throughout the delivery life cycle of a software development project. Common operations with this tool include deploying applications, scaling applications, checking the status of projects, and similar tasks.

INSTALLING THE **oc** COMMAND-LINE TOOL

During OpenShift installation, the **oc** command-line tool is installed on all master and node machines. You can install the **oc** client on systems that are not part of the OpenShift cluster, such as developer machines. Once installed, you can issue commands after authenticating against any master node with a user name and password.

There are several different methods available for installing the **oc** command-line tool, depending on the platform being used:

- On Red Hat Enterprise Linux (RHEL) systems with valid subscriptions, the tool is available directly as an RPM and installable using the **yum install** command.

```
[user@host ~]$ sudo yum install -y atomic-openshift-clients
```

- For alternative Linux distributions and other operating systems such as Windows and MacOS, native clients are available for download from the Red Hat Customer Portal. This also requires an active OpenShift subscription. These downloads are statically compiled to reduce incompatibility issues.

Once the **oc** CLI tool is installed, the **oc help** command displays help information. There are **oc** subcommands for tasks such as:

- Logging in and out of an OpenShift cluster.
- Creating, changing, and deleting projects.
- Creating applications inside a project. For example, creating a deployment configuration from a container image, or a build configuration from application source, and all associated resources.
- Creating, deleting, inspecting, editing, and exporting individual resources such as pods, services, and routes inside a project.
- Scaling applications.
- Starting new deployments and builds.
- Checking logs from application pods, deployments, and build operations.

CORE CLI COMMANDS

You can use the **oc login** command to log in interactively, which prompts you for a server name, a user name, and a password, or you can include the required information on the command line.

```
[student@workstation ~]$ oc login https://master.lab.example.com \
-u developer -p redhat
```



NOTE

Note that the backslash character (\) in the previous command is a command continuation character and should only be used if you are not entering the command as a single line.

After successful authentication from a client, OpenShift saves an authorization token in the user's home folder. This token is used for subsequent requests, negating the need to re-enter credentials or the full master URL.

To check your current credentials, run the **oc whoami** command:

```
[student@workstation ~]$ oc whoami
```

This command outputs the user name that you used when logging in.

```
developer
```

To create a new project, use the **oc new-project** command:

```
[student@workstation ~]$ oc new-project working
```

Use run the **oc status** command to verify the status of the project:

```
[student@workstation ~]$ oc status
```

Initially, the output from the status command reads:

```
In project working on server https://master.lab.example.com:443

You have no services, deployment configs, or build configs.
Run 'oc new-app' to create an application.
```

The output of the above command changes as you create new projects, and resources like **Services**, **DeploymentConfigs**, or **BuildConfigs** are added throughout this course.

To delete a project, use the **oc delete project** command:

```
[student@workstation ~]$ oc delete project working
```

To log out of the OpenShift cluster, use the **oc logout** command:

```
[student@workstation ~]$ oc logout
```

```
Logged "developer" out on "https://master.lab.example.com:443"
```



REFERENCES

Further information is available in the OpenShift Container Platform documentation:

CLI Reference

<https://access.redhat.com/documentation/en-us/>

► QUIZ

OPENSIFT CLI

Choose the correct answer or answers to the following questions:

► 1. Which of the following two statements are true regarding the **oc** CLI tool? (Choose two.)

- a. The **oc** command is used exclusively only to create new projects and applications. For other tasks, like deployment and scaling, you must use the OpenShift web console.
- b. The **oc** command is used exclusively to perform administrative and operational tasks, like deployment and scaling. To create new projects and applications, you must use the OpenShift web console.
- c. On RHEL based systems, the **oc** CLI tool is provided by the *atomic-openshift-clients* RPM package.
- d. You cannot install the **oc** tool on Windows or MacOS systems. Only RHEL based systems are supported.
- e. Pre-compiled native binaries of the **oc** CLI tool are available for Windows and MacOS systems.

► 2. Which of the following two statements are true regarding the OpenShift Container Platform (OCP) authentication mechanism? (Choose two.)

- a. The OpenShift master is accessible to all users without any authentication or authorization. Any user can issue any command from the **oc** client.
- b. OpenShift requires the user to authenticate with a user name and password prior to issuing any authorized command.
- c. For security reasons, OpenShift requires you to authenticate with a user name and password every single time you issue a command.
- d. OpenShift generates an authentication token upon successful login, and subsequent requests need not provide credentials again.

► 3. Which command below allows you to delete the project called **web-console** ?

- a. **oc web-console delete**
- b. **oc delete web-console**
- c. **oc delete project web-console**
- d. **oc delete web-console project**

► SOLUTION

OPENSIFT CLI

Choose the correct answer or answers to the following questions:

► 1. Which of the following two statements are true regarding the **oc** CLI tool? (Choose two.)

- a. The **oc** command is used exclusively only to create new projects and applications. For other tasks, like deployment and scaling, you must use the OpenShift web console.
- b. The **oc** command is used exclusively to perform administrative and operational tasks, like deployment and scaling. To create new projects and applications, you must use the OpenShift web console.
- c. On RHEL based systems, the **oc** CLI tool is provided by the *atomic-openshift-clients* RPM package.
- d. You cannot install the **oc** tool on Windows or MacOS systems. Only RHEL based systems are supported.
- e. Pre-compiled native binaries of the **oc** CLI tool are available for Windows and MacOS systems.

► 2. Which of the following two statements are true regarding the OpenShift Container Platform (OCP) authentication mechanism? (Choose two.)

- a. The OpenShift master is accessible to all users without any authentication or authorization. Any user can issue any command from the **oc** client.
- b. OpenShift requires the user to authenticate with a user name and password prior to issuing any authorized command.
- c. For security reasons, OpenShift requires you to authenticate with a user name and password every single time you issue a command.
- d. OpenShift generates an authentication token upon successful login, and subsequent requests need not provide credentials again.

► 3. Which command below allows you to delete the project called **web-console** ?

- a. **oc web-console delete**
- b. **oc delete web-console**
- c. **oc delete project web-console**
- d. **oc delete web-console project**

CREATING KUBERNETES RESOURCES

OBJECTIVES

After completing this section, students should be able to create standard Kubernetes resources.

OPENSHIFT CONTAINER PLATFORM (OCP) RESOURCES

The OpenShift Container Platform organizes entities in the OpenShift cluster as objects stored on the master node. These are collectively known as *resources*. The most common ones are:

Pod

A set of one or more containers that run in a node and share a unique IP address and volumes (persistent storage). Pods also define the security and runtime policy for each container.

Label

Labels are key-value pairs that can be assigned to any resource in the system for grouping and selection. Many resources use labels to identify sets of other resources.

Persistent Volume (PV)

Containers' data are ephemeral. Their contents are lost when they are removed. Persistent Volumes represent storage that can be accessed by pods to persist data and are mounted as a file system inside a pod container.

Regular OpenShift users cannot create persistent volumes. They need to be created and provisioned by a cluster administrator.

Persistent Volume Claim (PVC)

A Persistent Volume Claim is a request for storage from a project. The claim specifies desired characteristics of the storage, such as size, and the OpenShift cluster matches it to one of the available Persistent Volumes created by the administrator.

If the claim is satisfied, any pod in the same project that references the claim by name gets the associated PV mounted as a volume by containers inside the pod.

Pods can, alternatively, reference volumes of type **EmptyDir**, which is a temporary directory on the node machine and its contents are lost when the pod is stopped.

Service (SVC)

A name representing a set of pods (or external servers) that are accessed by other pods. A service is assigned an IP address and a DNS name, and can be exposed externally to the cluster via a port or a route. It is also easy to consume services from pods, because an environment variable with the name **SERVICE_HOST** is automatically injected into other pods.

Route

A route is an external DNS entry (either a top-level domain or a dynamically allocated name) that is created to point to a service so that it can be accessed outside the OpenShift cluster. Administrators configure one or more routers to handle those routes.

Replication Controller (RC)

A replication controller ensures that a specific number of pods (or replicas) are running. These pods are created from a template, which is part of the replication controller definition. If pods are lost by any reason, for example, a cluster node failure, then the controller creates new pods to replace the lost ones.

Deployment Configuration (DC)

Manages replication controllers to keep a set of pods updated regarding container image changes. A single deployment configuration is usually analogous to a single microservice. A DC can support many different deployment patterns, including full restart, customizable rolling updates, and fully custom behaviors, as well as hooks for integration with external Continuous Integration (CI) and Continuous Development (CD) systems.

Build Configuration (BC)

Manages building a container image from source code stored in a Git server. Builds can be based on Source-to-Image (S2I) process or Dockerfile. Build configurations also support hooks for integration with external CI and CD systems.

Project

Projects have a list of members and their roles. Most of the previous terms in this list exist inside of an OpenShift project, in Kubernetes terminology, *namespace*. Projects have a list of members and their roles, such as viewer, editor, or admin, as well as a set of security controls on the running pods, and limits on how many resources the project can use. Resource names are unique within a project. Developers may request projects to be created, but administrators control the resource quotas allocated to projects.

Regardless of the type of resource that the administrator is managing, the OpenShift command-line tool (**oc**) provides a unified and consistent way to update, modify, delete, and otherwise administer these resources, as well as helpers for working with the most frequently used resource types.

The **oc types** command provides a quick refresher on all the resource types available in OpenShift.

POD RESOURCE DEFINITION SYNTAX

OpenShift Container Platform 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 that are grouped together. An example of a WildFly application server pod definition in YAML format is as follows:

```
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 pod Kubernetes 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 called **name** that can be used to be found by other resources, usually a service, from Kubernetes.
- ➍ A container-dependent attribute identifying which port from 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** is declared internally by the Kubernetes runtime with a value called **user1**, and is forwarded to the container image definition. Since 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.

SERVICE RESOURCE DEFINITION SYNTAX

Kubernetes provides a virtual network that allows pods from different nodes to connect. But Kubernetes provides no easy way for a pod to learn the IP addresses of other pods.

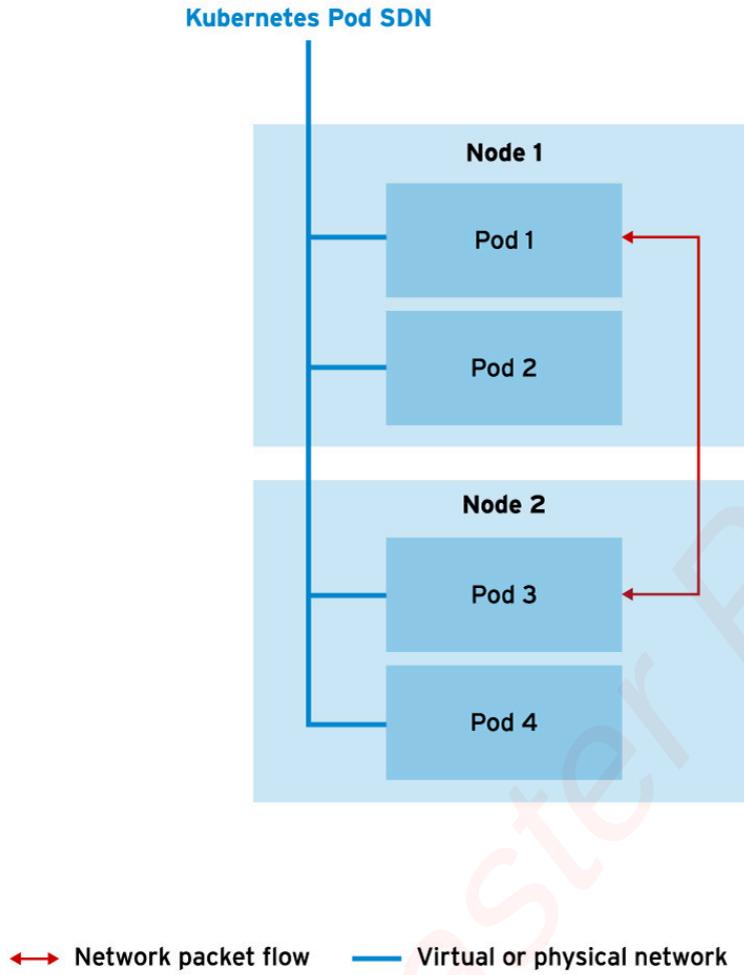


Figure 6.1: Kubernetes basic networking

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 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 node runs the pod after each restart.

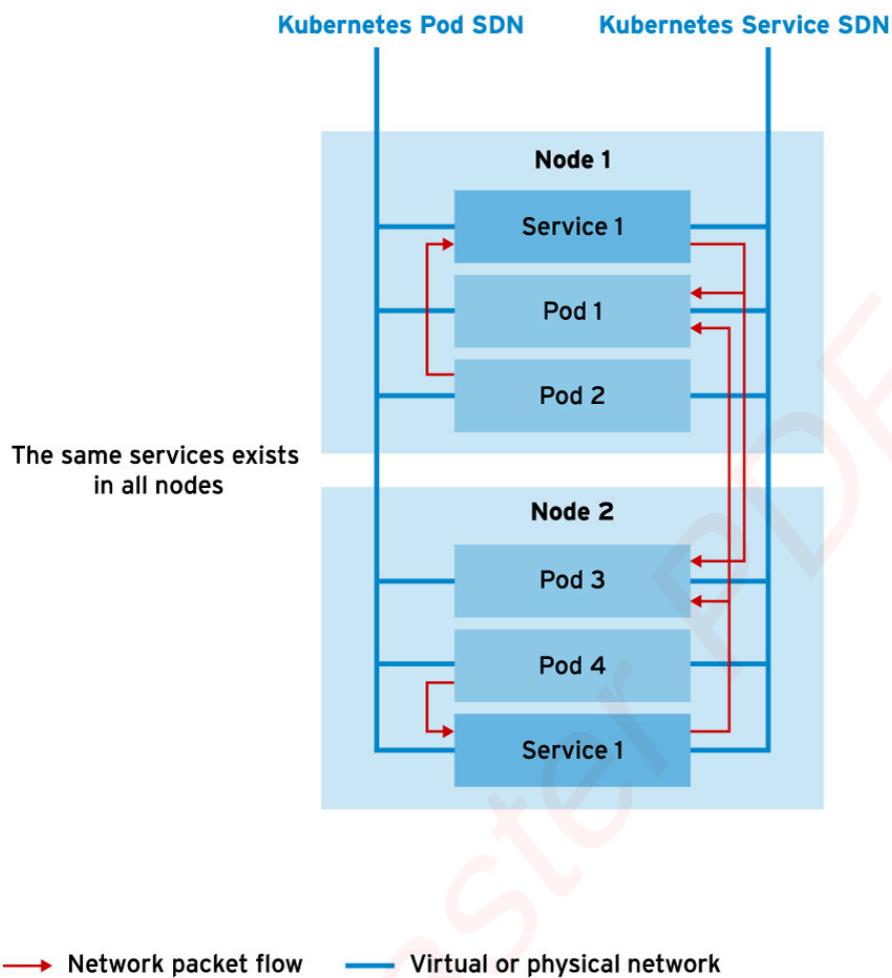


Figure 6.2: Kubernetes services networking

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 a 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 some of them fail. **DeploymentConfig** and **ReplicationControllers** are explained later in this chapter.

The following listing shows a minimal service definition in JSON syntax:

```
{
  "kind": "Service", ①
  "apiVersion": "v1",
  "metadata": {
    "name": "quotedb" ②
  },
  "spec": {
    "ports": [ ③
      {
        "port": 80,
        "targetPort": 8080
      }
    ]
  }
}
```

```

        "port": 3306,
        "targetPort": 3306
    },
],
"selector": {
    "name": "mysqlDb" ④
}
}
}

```

- ① The kind of Kubernetes resource. In this case, a *Service*.
- ② A unique name for the service.
- ③ *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, while 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*.
- ④ *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 among them.

Each service is assigned a unique IP address for clients to connect. 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 end point.

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** 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 a 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. But if the application was written to discover services using DNS queries, 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 node host, which then proxies connections to the service IP address. You can use the **oc edit svc** command to edit service attributes and specify **NodePort** as the type and provide a port value for the **nodePort** attribute.

OpenShift then proxies connections to the service via the public IP address of the 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.

The following figure shows how **NodePort** services allows external access to Kubernetes services. Later this course will present OpenShift routes in more detail.

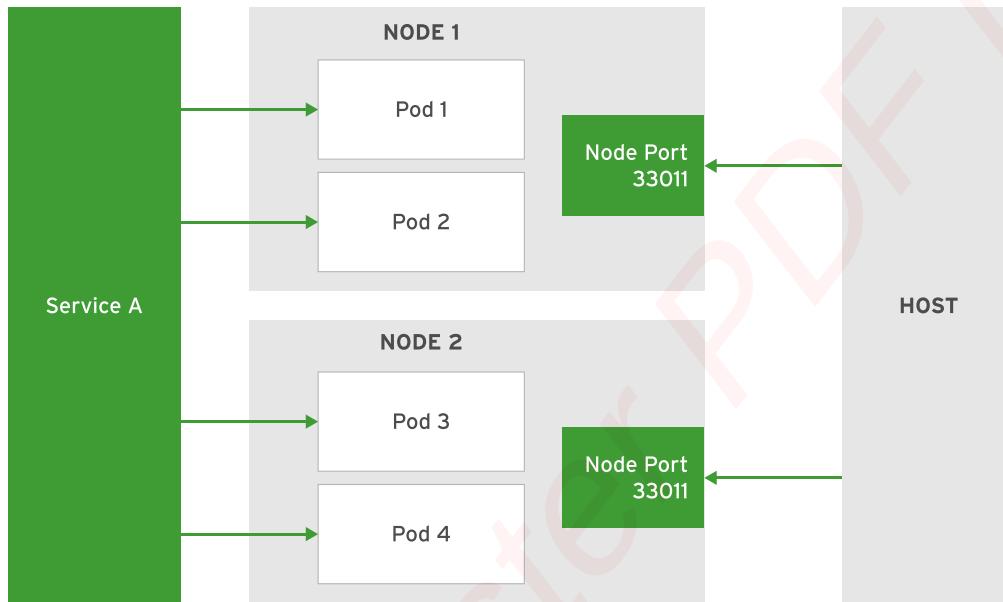


Figure 6.3: Kubernetes NodePort services

CREATING APPLICATIONS USING **oc new-app**

Simple applications, complex multi-tier 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 option **-o json** or **-o yaml**, 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 briefly 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:

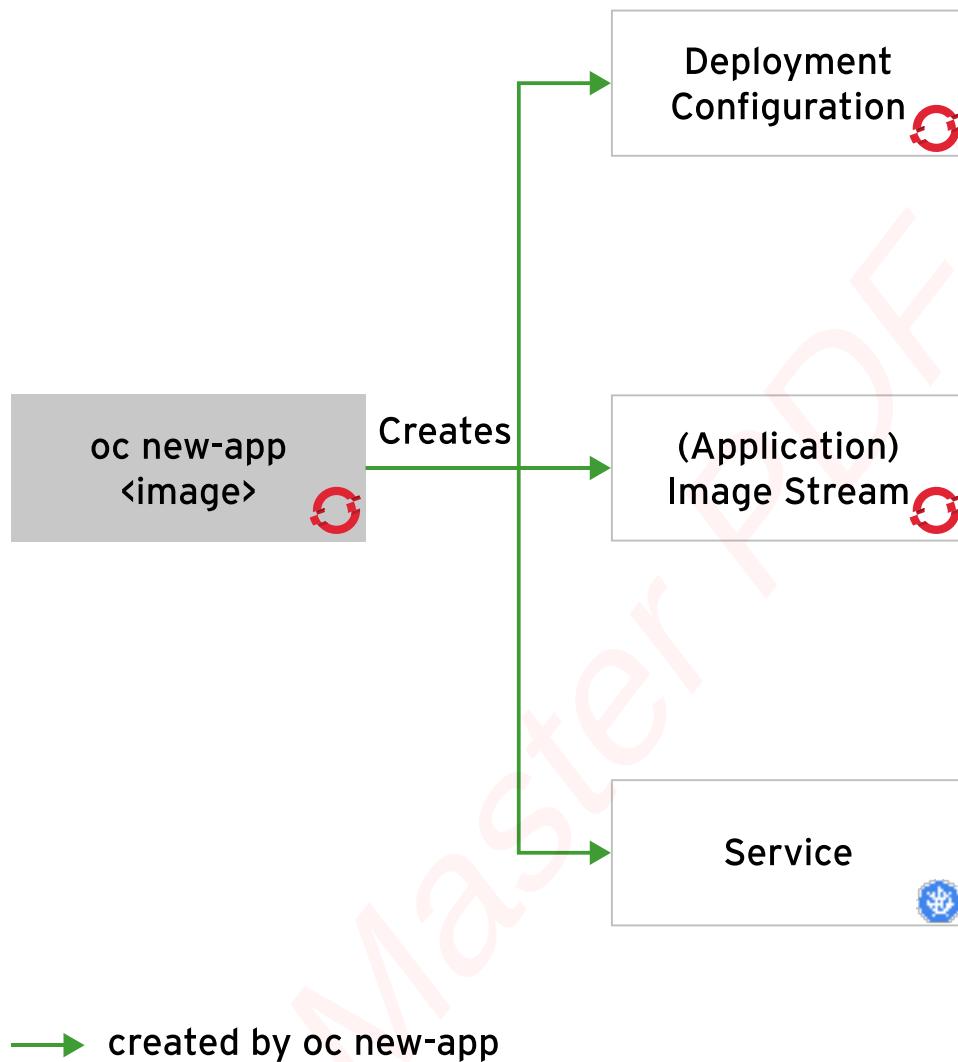


Figure 6.4: Resources created by the **oc new-app command**

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.

USEFUL COMMANDS TO MANAGE OPENSHIFT RESOURCES

There are several essential commands used to manage OpenShift resources as described below.

Typically, as an administrator, you will most likely use **oc get** command. This retrieves information about resources in the cluster. Generally, this command outputs only the most important characteristics of the resources and omits more detailed information.

If the **RESOURCE_NAME** parameter is omitted, then all resources of the specified **RESOURCE_TYPE** are summarized. The following output is a sample of an execution of **oc get pods** command.

NAME	READY	STATUS	RESTARTS	AGE
nginx-1-5r583	1/1	Running	0	1h
myapp-1-l44m7	1/1	Running	0	1h

oc get all

If the administrator wants a summary of all the most important components of a cluster, the **oc get all** command can be executed. This command iterates through the major resource types for the current project and prints out a summary of their information. For example:

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	
rc/nginx-1	1	1	1	
<hr/>				
NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
svc/nginx	172.30.72.75	<none>	80/TCP, 443/TCP	1h
<hr/>				
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, additional information about the resource can be retrieved by using the **oc describe** command. Unlike the **oc get** command, there is no way to simply 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:	docker-registry-4-ku34r
Namespace:	default
Security Policy:	restricted
Node:	node.lab.example.com/172.25.250.11
Start Time:	Mon, 23 Jan 2017 12:17:28 -0500
Labels:	deployment=docker-registry-4 deploymentconfig=docker-registry docker-registry=default
Status:	Running
<hr/>	

```
No events
```

oc export

This command can be used to export the a resource definition. Typical use cases include creating a backup, or to aid in 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 allows the user to create 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 up a **vi** buffer for editing the resource definition.

oc delete RESOURCE_TYPE name

The **oc delete** command allows the user to remove a resource from an OpenShift cluster. Note that a fundamental understanding of the OpenShift architecture is needed here, because deletion of managed resources like pods result in newer instances of those resources being automatically recreated. 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 allows the user to execute commands inside a container. You can use this command to run interactive as well as noninteractive batch commands as part of a script.

DEMONSTRATION: CREATING BASIC KUBERNETES RESOURCES

1. Log in to OpenShift as the **developer** user on the **workstation** VM:

```
[student@workstation ~]$ oc login -u developer -p redhat \
https://master.lab.example.com
```

2. Create a new project for the resources you will create during this demonstration:

```
[student@workstation ~]$ oc new-project basic-kubernetes
```

3. Relax the default cluster security policy.

The **nginx** image from Docker Hub runs as root, but this is not allowed by the default OpenShift security policy.

As the **admin** user, change the default security policy to allow containers to run as root:

```
[student@workstation ~]$ oc login -u admin -p redhat https://
master.lab.example.com
Login successful.
...
[student@workstation ~]$ oc adm policy add-scc-to-user anyuid -z default
scc "anyuid" added to: ["system:serviceaccount:basic-kubernetes:default"]
```

4. Log in back to OpenShift as the **developer** user and create a new application from the **nginx** container image using the **oc new-app** command.

Use the **--docker-image** option with **oc new-app** to specify the classroom private registry URI so that OpenShift does not try and pull the image from the Internet:

```
[student@workstation ~]$ oc login -u developer -p redhat \
https://master.lab.example.com
Login successful.

[student@workstation ~]$ oc new-app \
--docker-image=registry.lab.example.com/nginx:latest \
--name=nginx
--> Found Docker image ae513a4 (7 weeks old) from registry.lab.example.com for
"registry.lab.example.com/nginx:latest"

      * An image stream will be created as "nginx:latest" that will track this image
      * This image will be deployed in deployment config "nginx"
      * Port 80/tcp will be load balanced by service "nginx"
        * Other containers can access this service through the hostname "nginx"
      * WARNING: Image "registry.lab.example.com/nginx:latest" runs as the 'root'
user which may not be permitted by your cluster administrator

--> Creating resources ...
  imagestream "nginx" created
  deploymentconfig "nginx" created
  service "nginx" 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/nginx'
  Run 'oc status' to view your app.
```

5. Run the **oc status** command to view the status of the new application, and verify if the deployment of the Nginx image was successful:

```
[student@workstation ~]$ oc status
In project basic-kubernetes on server https://master.lab.example.com:443

svc/nginx - 172.30.149.93:80
dc/nginx deploys istag/nginx:latest
  deployment #1 deployed 50 seconds ago - 1 pod

2 infos identified, use 'oc status -v' to see details.
```

6. List the pods in this project to verify if the Nginx pod is ready and running:

```
[student@workstation ~]$ oc get pods
NAME          READY     STATUS    RESTARTS   AGE
nginx-1-ypp8t 1/1       Running   0          25m
```

7. Use the **oc describe** command to view more details about the pod:

```
[student@workstation ~]$ oc describe pod nginx-1-ypp8t
Name:           nginx-1-ypp8t
Namespace:      basic-kubernetes
Node:          node2.lab.example.com/172.25.250.12
Start Time:    Tue, 19 Jun 2018 10:46:50 +0000
Labels:         app=nginx
                deployment=nginx-1
                deploymentconfig=nginx
Annotations:   openshift.io/deployment-config.latest-version=1
                openshift.io/deployment-config.name=nginx
                openshift.io/deployment.name=nginx-1
                openshift.io/generated-by=OpenShiftNewApp
                openshift.io/scc=anyuid
Status:        Running
IP:            10.129.0.44
...
```

8. List the services in this project and verify if a service to access the Nginx pod was created:

```
[student@workstation ~]$ oc get svc
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
nginx    ClusterIP  172.30.149.93  <none>          80/TCP       3m
```

9. Retrieve the details of **nginx** service and note the Service IP through which the Nginx pod can be accessed:

```
[student@workstation ~]$ oc describe service nginx
Name:           nginx
Namespace:      basic-kubernetes
Labels:         app=nginx
Annotations:   openshift.io/generated-by=OpenShiftNewApp
Selector:       app=nginx,deploymentconfig=nginx
Type:          ClusterIP
IP:            172.30.149.93
Port:          80-tcp  80/TCP
TargetPort:    80/TCP
Endpoints:    10.129.0.44:80
Session Affinity: None
Events:        <none>
```

10. Open an SSH session to the **master** machine and access the default Nginx home page using the service IP:

```
[student@workstation ~]$ ssh master curl -s http://172.30.149.93
<!DOCTYPE html>
<html>
...
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>
...
```

Accessing the Service IP and port directly works in this scenario because **master** is the machine that manages the entire OpenShift cluster in the classroom environment.

11. Delete the project, to remove all the resources in the project:

```
[student@workstation ~]$ oc delete project basic-kubernetes
```

This concludes the demonstration.



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/3.9/html/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/3.9/html/creating_images/

► 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.

RESOURCES	
Files:	/home/student/D0180/labs/mysql Openshift
Resources:	Red Hat Software Collections official MySQL 5.7 image (rhscl/mysql-57-rhel7)

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 mysql-openshift setup
```

- 1. Log in to OpenShift as a developer user and create a new project for this exercise.

- 1.1. From the **workstation** VM, log in to OpenShift as the **developer** user with **redhat** as password:

```
[student@workstation ~]$ oc login -u developer -p redhat \
https://master.lab.example.com
```

If the **oc login** command prompts about using insecure connections, answer **y** (yes).



WARNING

Be careful with the user name. If a different user is used to run the following steps, the grading script fails.

- 1.2. Create a new project for the resources you will create during this exercise:

```
[student@workstation ~]$ oc new-project mysql-openshift
```

- 2. Create a new application from the **rhscl/mysql-57-rhel7** container image using the **oc new-app** command.

This image requires several environment variables (**MYSQL_USER**, **MYSQL_PASSWORD**, **MYSQL_DATABASE**, and **MYSQL_ROOT_PASSWORD**) to be fed using multiple instances of **-e** option.

Use the **--docker-image** option with **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
--> Found Docker image 63d6bb0 (2 months old) from registry.lab.example.com for
"registry.lab.example.com/rhscl/mysql-57-rhel7:latest"
...
--> Creating resources ...
imagestream "mysql-openshift" created
deploymentconfig "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 if the MySQL pod was created successfully and view details about the pod and it's service.
 - 3.1. Run the **oc status** command to view the status of the new application and verify if the deployment of the MySQL image was successful:

```
[student@workstation ~]$ oc status
In project mysql-openshift on server https://master.lab.example.com:443

svc/mysql-openshift - 172.30.205.27:3306
dc/mysql-openshift deploys istag/mysql-openshift:latest
  deployment #1 running for 11 seconds - 0/1 pods
...
```

- 3.2. List the pods in this project to verify if the MySQL pod is ready and running:

```
[student@workstation ~]$ oc get pods -o=wide
NAME          READY   STATUS    RESTARTS   AGE     IP
mysql-openshift-1-flknk   1/1     Running   0          1m     10.129.0.25
node2.lab.example.com
```



NOTE

Notice the node 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-ct78z
```

```
Name: mysql-openshift-1-ct78z
Namespace: mysql-openshift
Node: node2.lab.example.com/172.25.250.12
Start Time: Thu, 14 Jun 2018 09:01:19 +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: 10.129.0.23
...
```

- 3.4. List the services in this project and verify if a 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  172.30.205.27  <none>           3306/TCP   6m
```

- 3.5. Retrieve the details of **mysql-openshift** service using **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: 172.30.205.27
Port: 3306-tcp 3306/TCP
TargetPort: 3306/TCP
Endpoints: 10.129.0.23:3306
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
Labels: app=mysql-openshift
...
Deployment #1 (latest):
Name: mysql-openshift-1
Created: 10 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
```

...

- ▶ 4. Connect to the MySQL database server and verify if the database was created successfully.

- 4.1. From the **master** machine, connect to the MySQL server using the MySQL client. Use **mysql.openshift.mysql.openshift.svc.cluster.local** as the Database server's hostname:

```
[student@workstation ~]$ ssh master

[student@master ~]$ mysql -P3306 -uuser1 -pmypa55 \
-hmysql.openshift.mysql.openshift.svc.cluster.local

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)]>
```



NOTE

The services can only be accessed on the cluster servers themselves.

- 4.2. Verify if the **testdb** database has been created:

```
MySQL [(none)]> show databases;
+-----+
| Database      |
+-----+
| information_schema |
| testdb         |
+-----+
2 rows in set (0.00 sec)
```

- 4.3. Exit from the MySQL prompt:

```
MySQL [(none)]> exit
Bye
```

- 4.4. Exit from **master** node.

```
[student@master ~]$ exit
```

- ▶ 5. Verify that the database was correctly set up. Run the following command from a terminal window:

```
[student@workstation ~]$ lab mysql.openshift grade
```

- **6.** Delete the project which in turn all the resources in the project:

```
[student@workstation ~]$ oc delete project mysql-openshift
```

This concludes the 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 facility that makes it easy to build a container image from application source code. This facility takes an application's source code from a Git server, 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.

The following figure shows the resources created by the `oc new-app <source>` command when the argument is an application source code repository. Notice that a Deployment Configuration and all its dependent resources are also created.

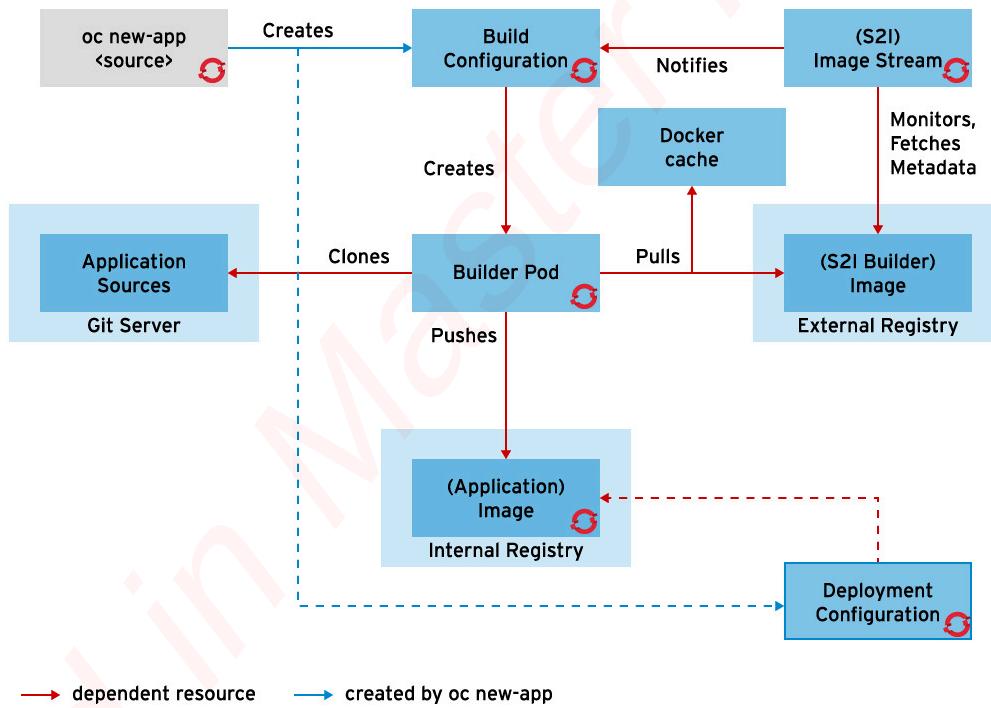


Figure 6.5: Deployment Configuration and dependent resources

S2I is the major 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.

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, a new container image is created. Pods created using the older container image are replaced by pods using the new image.

While it is obvious that the container image needs to be updated when application code changes, it may not be obvious 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. An application is built against an image stream. The OpenShift installer populates several image streams by default during installation. To check available image streams, use the **oc get** command, as follows:

\$ oc get is -n openshift		
NAME	DOCKER REPO	TAGS
jenkins	172.30.0.103:5000/openshift/jenkins	2,1
mariadb	172.30.0.103:5000/openshift/mariadb	10.1
mongodb	172.30.0.103:5000/openshift/mongodb	3.2,2.6,2.4
mysql	172.30.0.103:5000/openshift/mysql	5.5,5.6
nodejs	172.30.0.103:5000/openshift/nodejs	0.10,4
perl	172.30.0.103:5000/openshift/perl	5.20,5.16
php	172.30.0.103:5000/openshift/php	5.5,5.6
postgresql	172.30.0.103:5000/openshift/postgresql	9.5,9.4,9.2
python	172.30.0.103:5000/openshift/python	3.5,3.4,3.3
ruby	172.30.0.103:5000/openshift/ruby	2.3,2.2,2.0



NOTE

Your OpenShift instance may have more or fewer image streams depending on local additions and OpenShift point releases.

OpenShift has the ability to detect when an image stream changes and to take action based on that change. If a security issue is found 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.

An organization will likely choose 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
```

- ➊ The image stream used in the process appears to the left of the tilde (~).
- ➋ The URL after the tilde indicates the location of the source code's Git repository.
- ➌ Sets the application name.

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 at the current directory.

```
$ oc new-app .
```



NOTE

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
Java EE	pom.xml
nodejs	app.json, package.json
php	index.php, composer.json
python	requirements.txt, config.py
perl	index.pl, cpanfile

After a language is detected, **new-app** 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
```

A list of resources will be created by this JSON definition file. The first resource is the image stream:

```
...
{
    "kind": "ImageStream", ①
    "apiVersion": "v1",
    "metadata": {
        "name": "myapp", ②
        "creationTimestamp": null
        "labels": {
            "app": "myapp"
        },
        "annotations": {
            "openshift.io/generated-by": "OpenShiftNewApp"
        }
    },
    "spec": {},
    "status": {
        "dockerImageRepository": ""
    }
},
...

```

- ①** Define a resource type of image stream.
- ②** Name the image stream as **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 that are used by OpenShift to create a runnable image.

```
...
{
    "kind": "BuildConfig", ①
    "apiVersion": "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": "3q8K8JND0Rzhjoz1KgMz"
            }
        },
        {
            "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:5.5" ⑤
            }
        }
    },
    "output": {
        "to": {
            "kind": "ImageStreamTag",
            "name": "myapp:latest" ⑥
        }
    },
    "resources": {},
    "postCommit": {},
    "nodeSelector": null
},
"status": {
    "lastVersion": 0
}
},
...

```

- ①** Define a resource type of **BuildConfig**.
- ②** Name the **BuildConfig** as **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:5.5** image stream.
- ⑥** Name the output image stream as **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 DeploymentConfigs are:

- User customizable strategies to transition from the existing deployments to new deployments.
- Rollbacks to a previous deployment.
- Manual replication scaling.

```

...
{
    "kind": "DeploymentConfig", ❶
    "apiVersion": "v1",
    "metadata": {
        "name": "myapp", ❷
        "creationTimestamp": null,
        "labels": {
            "app": "myapp"
        },
        "annotations": {
            "openshift.io/generated-by": "OpenShiftNewApp"
        }
    },
    "spec": {
        "strategy": {
            "resources": {}
        },
        "triggers": [
            {
                "type": "ConfigChange" ❸
            },
            {
                "type": "ImageChange", ❹
                "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/container.myapp.image.entrypoint":
            "[\"container-entrypoint\", \"/bin/sh\", \"-c\", \"$STI_SCRIPTS_PATH/usage\"]",
            "openshift.io/generated-by": "OpenShiftNewApp"
        }
    },
    "spec": {
        "containers": [
            {
                "name": "myapp",
                "image": "myapp:latest", ⑤
                "ports": [ ⑥
                    {
                        "containerPort": 8080,
                        "protocol": "TCP"
                    }
                ],
                "resources": {}
            }
        ]
    },
    "status": {}
},
...

```

- ①** Define a resource type of **DeploymentConfig**.
- ②** Name the **DeploymentConfig** as **myapp**.
- ③** A configuration change trigger causes a new deployment to be created any time the replication controller template changes.
- ④** An image change trigger causes a new deployment to be created each time a new version of the **myapp:latest** image is available in the repository.
- ⑤** Define that the **library/myapp:latest** container image should be deployed.
- ⑥** Specifies the container ports.

The last item is the service, already covered in previous chapters:

```

...
{
    "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
        }
    ],
    "selector": {
        "app": "myapp",
        "deploymentconfig": "myapp"
    }
},
"status": {
    "loadBalancer": {}
}
}
]
}
}

```

**NOTE**

By default, no route is created by the **oc new-app** command. A route can be created after the application creation. 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. See a list of application builds with **oc get builds**:

```
$ oc get builds
NAME      TYPE      FROM      STATUS      STARTED      DURATION
myapp-1   Source    Git@59d3066  Complete   3 days ago  2m13s
```

OpenShift allows viewing of the build logs. The following command shows the last few lines of the build log:

```
$ oc logs build/myapp-1
```

**NOTE**

If the build is not **Running** yet, or the **s2i-build** pod has not been deployed yet, the above command throws an error. Just wait a few moments and retry it.

Trigger a new builder 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 BUILDCONFIG AND DEPLOYMENTCONFIG

The BuildConfig pod is responsible for creating the images in OpenShift and pushing them to the internal Docker registry. Any source code or content update normally requires a new build to guarantee the image is updated.

The DeploymentConfig pod is responsible for deploying pods into OpenShift. The outcome from a DeploymentConfig pod execution is the creation of pods with the images deployed in the internal docker registry. Any existing running pod may be destroyed, depending on how the DeploymentConfig is set.

The BuildConfig and DeploymentConfig resources do not interact directly. The BuildConfig 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

Additional information about S2I builds is available in the *Core Concepts* section of the OpenShift Container Platform documentation:

Architecture

<https://access.redhat.com/documentation/en-us/>

Additional information about the S2I build process is available in the OpenShift Container Platform documentation:

Developer Guide

<https://access.redhat.com/documentation/en-us/>

► 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 on an OpenShift cluster.

RESOURCES	
Files:	/home/student/D0180/labs/s2i
Application URL:	N/A

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

Retrieve the lab files and verify that Docker and OpenShift are up and running, by executing the lab script:

```
[student@workstation ~]$ lab s2i setup
[student@workstation ~]$ cd ~/D0180/labs/s2i
```

- 1. Examine the source code for the PHP version 5.6 Source-to-Image container.
- 1.1. Use the **tree** command to review the files that make up the container image:

```
[student@workstation 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

```

- 1.2. 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 that is provided when the application is built using the `oc new-app` command or the web console. The remainder of the script supports retrieving PHP packages that your application names as requirements, if any.
- 1.3. Review the **s2i-php-container/5.6/s2i/bin/run** script. This script is executed as the command (**CMD**) for the PHP container built by the Source-to-Image process. It is responsible for setting up and running the Apache HTTP service which executes the PHP code in response to HTTP requests.
- 1.4. 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. Login to OpenShift:

```
[student@workstation s2i]$ oc login -u developer -p redhat https://master.lab.example.com
Login successful.

You don't have any projects. You can try to create a new project, by running
  oc new-project <projectname>
```

► 3. Create a new project named **s2i**:

```
[student@workstation s2i]$ oc new-project s2i
Now using project "s2i" on server "https://172.25.250.11:8443".
```

...

Your current project may differ. You may also be asked to login. If so, the password is **redhat**.

- ▶ 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 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 s2i]$ oc new-app \
php:5.6~http://services.lab.example.com/php-helloworld
```

- 4.2. Wait for the build to complete and the application to deploy. Review the resources that were built for you by the **oc new-app** command. Examine the **BuildConfig** resource created using **oc describe**:

```
[student@workstation s2i]$ oc get pods -w # wait for build/deploy
...
Ctrl+C
[student@workstation s2i]$ oc describe bc/php-helloworld
Name:   php-helloworld
Namespace: s2i
Created: 2 minutes ago
Labels:   app=php-helloworld
Annotations: openshift.io/generated-by=OpenShiftNewApp
Latest Version: 1

Strategy: Source
URL:   http://services.lab.example.com/php-helloworld
From Image: ImageStreamTag openshift/php:5.6
Output to: ImageStreamTag php-helloworld:latest
...

Build Run Policy: Serial
Triggered by: Config, ImageChange
Webhook GitHub:
  URL: https://master.lab.example.com:443/apis/build.openshift.io/v1/namespaces/
s2i/buildconfigs/php-helloworld/webhooks/<secret>/github
Webhook Generic:
  URL: https://master.lab.example.com:443/apis/build.openshift.io/v1/namespaces/
s2i/buildconfigs/php-helloworld/webhooks/<secret>/generic
  AllowEnv: false

Build Status Duration Creation Time
```

```
php-helloworld-1 complete 1m46s 2018-06-15 10:34:10 +0000 UTC
```

The last part of the display gives the build history for this application. So far, there has been only one build and that build completed successfully.

- 4.3. Examine the build log for this build. Use the build name given in the listing above and add **-build** to the name to create the pod name for this build, **php-helloworld-1-build**.

```
[student@workstation s2i]$ oc logs php-helloworld-1-build
--> Installing application source...
=> sourcing 20-copy-config.sh ...
--> 10:35:12      Processing additional arbitrary httpd configuration provided by
s2i ...
=> sourcing 00-documentroot.conf ...
=> sourcing 50-mpm-tuning.conf ...
=> sourcing 40-ssl-certs.sh ...

Pushing image docker-registry.default.svc:5000/s2i/php-helloworld:latest ...
Pushed 0/6 layers, 1% complete
Pushed 1/6 layers, 20% complete
Pushed 2/6 layers, 40% complete
Pushed 3/6 layers, 59% complete
Pushed 4/6 layers, 81% complete
Pushed 5/6 layers, 99% complete
Pushed 6/6 layers, 100% complete
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 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
...
  Containers:
    php-helloworld:
      Image:  docker-registry.default.svc:5000/s2i/php-
helloworld@sha256:6d274e9d6e3d4ba11e5dc3fc25e1326c8170b1562c2e3330595ed21c7dfb983
      Ports:  8443/TCP, 8080/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
...

```

- 4.5. Review the **service** for this application:

```

[student@workstation s2i]$ oc describe svc/php-helloworld
Name:           php-helloworld
Namespace:      s2i
Labels:          app=php-helloworld
Annotations:    openshift.io/generated-by=OpenShiftNewApp
Selector:        app=php-helloworld,deploymentconfig=php-helloworld
Type:           ClusterIP
IP:             172.30.18.13
Port:           8080-tcp  8080/TCP
TargetPort:     8080/TCP
Endpoints:      10.129.0.26:8080
Port:           8443-tcp  8443/TCP
TargetPort:     8443/TCP
Endpoints:      10.129.0.26:8443
Session Affinity: None
Events:         <none>

```

- 4.6. Test the application by sending it an HTTP GET request (replace this IP address with the one shown in your service listing):

```

[student@workstation s2i]$ ssh student@master curl -s 172.30.18.13:8080
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 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.

```

```
[student@workstation s2i]$ cd php-helloworld
```

- 5.2. Edit the **index.php** file and make the contents look like this:

```
<?php
print "Hello, World! php version is " . PHP_VERSION . "\n";
print "A change is a coming!\n";
?>
```

Save the file.

- 5.3. 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
...
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
  ecb93d1..eb438ba  master -> master
[student@workstation php-helloworld]$ cd ..
```

- 5.4. Start a new Source-to-Image build process and wait for it to build and deploy:

```
[student@workstation s2i]$ oc start-build php-helloworld
build "php-helloworld-2" started
[student@workstation s2i]$ oc get pods -w
...
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
```

^C

- 5.5. Test that your changes are served by the application:

```
[student@workstation s2i]$ ssh student@master curl -s 172.30.18.13:8080
Hello, World! php version is 5.6.25
A change is a coming!
```

- 6. Grade the lab:

```
[student@workstation s2i]$ lab s2i grade
Accessing PHP web application..... SUCCESS
```

- 7. Clean up the lab by deleting the OpenShift project, which in turn deletes all the Kubernetes and OpenShift resources:

```
[student@workstation s2i]$ oc delete project s2i
project "s2i" deleted
```

This concludes this guided exercise.

CREATING ROUTES

OBJECTIVES

After completing this section, students should be able to create a route to a service.

WORKING WITH ROUTES

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.

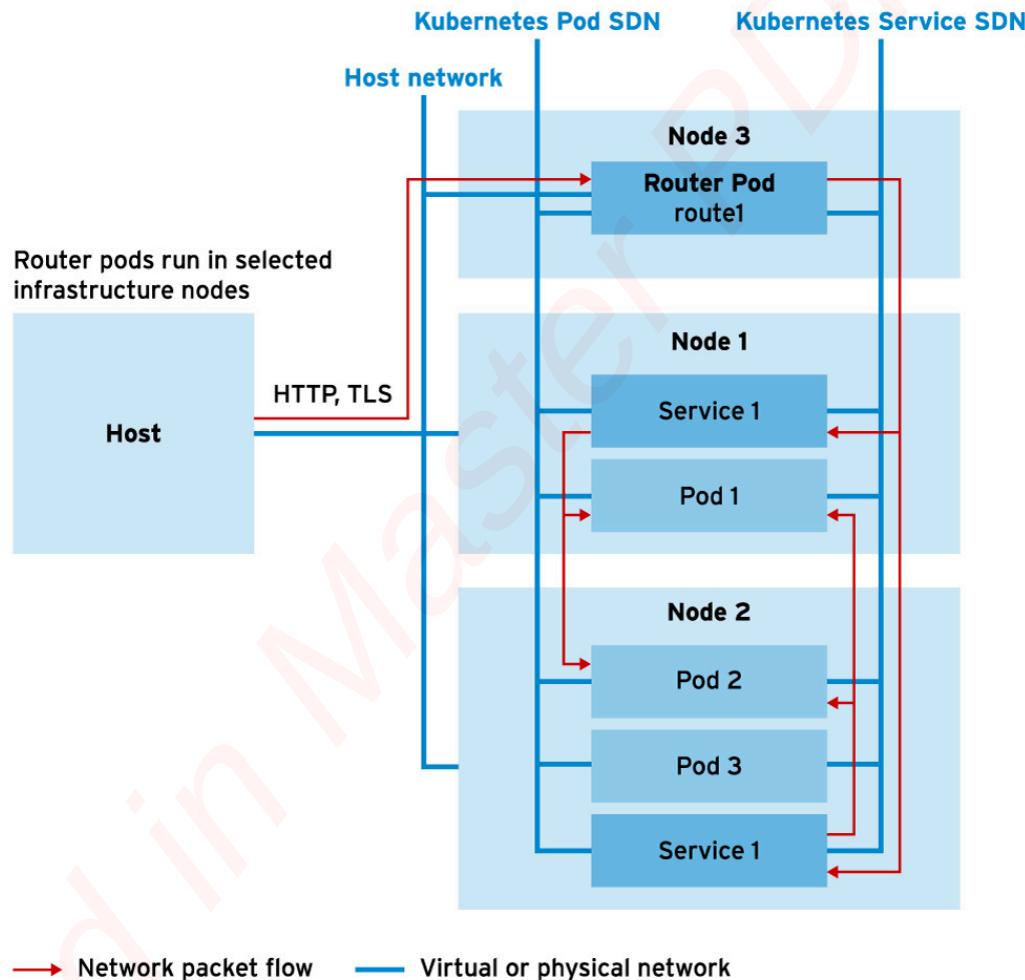


Figure 6.6: OpenShift routes and Kubernetes services

A route connects a public-facing IP address and DNS host name to an internal-facing service IP. At least, this is the concept. In practice, to improve performance and reduce latency, the OpenShift router connects directly to the pods using the internal pod software-defined network (SDN). Use the service to find the end points, that is, the pods exposed by the service.

OpenShift routes are implemented by a shared router service, which runs as pods inside the OpenShift instance and can be scaled and replicated like any other regular pod. This router service is based on the open source software *HAProxy*.

An important consideration for the OpenShift administrator 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 listings 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"
    }
  }
}
```

Starting the route resource, there are the standard, `apiVersion`, `kind`, and `metadata` attributes. The `Route` value for `kind` shows that this is a resource attribute, and the `metadata.name` attribute gives this particular route, the identifier, `quoteapp`.

As with pods and services, the interesting part is the `spec` attribute, which is an object containing the following attributes:

- `host` is a string containing the FQDN name associated with the route. It has to be preconfigured to resolve to the OpenShift router IP address.
- `to` is an object stating the `kind` of resource this route points to, which in this case is an OpenShift Service with the `name` set to `quoteapp`.



NOTE

Remember the 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 uses selectors to link to pod resources containing specific labels, a route links directly to the service resource name.

CREATING ROUTES

Route resources can be created like any other OpenShift resource by providing `oc create` with a JSON or YAML resource definition file.

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. When `oc new-app` creates a group of pods from a template, nothing prevents the template from including a route resource as part of the application. The same is true for the web console.

Another way to create a route is to use the **oc expose** 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
```

Routes created from templates or from **oc expose** generate DNS names of the form:

route-name-project-name.default-domain

Where:

- *route-name* is the name explicitly assigned to the route, or the name of the originating resource (template for **oc new-app** and service for **oc expose** or from the **--name** option).
- *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 prerequisite for installing OpenShift.

For example, creating route **quote** in project **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 simply 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. Invalid wildcard domain host names, that is, host names that do not correspond to any route, will be blocked by the OpenShift router and result in an HTTP 404 error.

Finding the Default Domain

The subdomain or, default domain, is defined in the OpenShift configuration file **master-config.yaml** in the **routingConfig** section with the keyword **subdomain**. For example:

```
routingConfig:
  subdomain: 172.25.250.254.xip.io
```



REFERENCES

Additional information about the architecture of routes in OpenShift is available in the *Routes* section of the OpenShift Container Platform documentation:

Architecture

<https://access.redhat.com/documentation/en-us/>

Additional developer information about routes is available in the *Routes* section of the OpenShift Container Platform documentation:

Developer Guide

<https://access.redhat.com/documentation/en-us/>

► 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.

RESOURCES	
Files:	N/A
Application URL:	http://xyz-route.apps.lab.example.com

OUTCOMES

You should be able to expose a service as a route for a deployed OpenShift application.

BEFORE YOU BEGIN

Retrieve the lab files and verify that Docker and OpenShift are up and running, by executing the lab script:

```
[student@workstation ~]$ lab route setup
```

- 1. On **workstation** open a terminal and login to OpenShift.

```
[student@workstation ~]$ oc login -u developer https://master.lab.example.com
Logged into "https://master.lab.example.com:443" as "developer" using existing
credentials.
...
```

You may have to enter credentials for the **developer** account. The password is **redhat**. The current project in your command output may differ from the listing above.

- 2. Create a new project named **route**:

```
[student@workstation ~]$ oc new-project route
Now using project "route" on server "https://master.lab.example.com:443".
```

You can add applications to this project with the 'new-app' command. For example, try:

```
oc new-app centos/ruby-22-centos7~https://github.com/openshift/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 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
```

- 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
...
^C
```

- 3.3. Review the **service** for this application using **oc describe**:

```
[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 creating a route with a default name and fully qualified domain name (FQDN):

```
[student@workstation ~]$ oc expose svc/php-helloworld
route "php-helloworld" exposed
[student@workstation ~]$ oc get route
NAME            HOST/PORT                               PATH      SERVICES
PORT
php-helloworld  php-helloworld-route.apps.lab.example.com   php-
helloworld     8080-tcp
[student@workstation ~]$ curl php-helloworld-route.apps.lab.example.com
```

```
Hello, World! php version is 5.6.25
```

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. Use **curl** command to verify that the application can be accessed with its route address.

► 5. Replace this route with a route named **xyz**.

5.1. Delete the current route:

```
[student@workstation ~]$ oc delete route/php-helloworld
route "php-helloworld" deleted
```

5.2. Expose the service creating a route named **xyz**:

```
[student@workstation ~]$ oc expose svc/php-helloworld --name=xyz
route "xyz" exposed
[student@workstation ~]$ oc get route
NAME          HOST/PORT           PATH      SERVICES
PORT          TERMINATION
xyz           xyz-route.apps.lab.example.com   php-helloworld
8080-tcp
```

Note the new FQDN that was generated.

5.3. Make an HTTP request using the FQDN on port 80:

```
[student@workstation ~]$ curl xyz-route.apps.lab.example.com
Hello, World! php version is 5.6.25
A change is a coming!
```



NOTE

The output of the PHP application will be different if you did not complete the previous exercise in this chapter.

► 6. Grade your lab progress:

```
[student@workstation ~]$ lab route grade
Accessing PHP web application..... SUCCESS
```

► 7. Clean up the lab environment by deleting the project:

```
[student@workstation ~]$ oc delete project route
project "route" deleted
```

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.
- Examine resources for an application.
- Manage and monitor the build cycle of an application.

ACCESSING THE OPENSHIFT WEB CONSOLE

The OpenShift web console allows a user to execute many of the same tasks as the OpenShift command line. Projects can be created, applications can be created within those projects, and application resources can be examined and manipulated as needed. The OpenShift web console runs as a pod on the master.

Accessing the Web Console

The web console runs in a web browser. The URL is of the format `https://{{hostname of OpenShift master}}/console`. By default, OpenShift generates a self-signed certificate for the web console. The user must trust this certificate in order to gain access. The console requires authentication.

Managing Projects

Upon successful login, the user may select, edit, delete, and create projects on the home page. Once a project is selected, the user is taken to the Overview page which shows all of the applications created within that project space.

Application Overview Page

The application overview page is the heart of the web console.

The screenshot shows the OpenShift Application Overview page for the 'php-helloworld' application. At the top, it displays the application name 'php-helloworld' and its route URL 'http://php-helloworld-console.apps.lab.example.com'. Below this, under 'DEPLOYMENT CONFIG', it shows 'php-helloworld, #1'. In the 'CONTAINERS' section, there is one pod listed. The 'Scale tool' (indicated by arrows) is shown next to the pod count. Under 'NETWORKING', it shows a 'Service' named 'php-helloworld' and a 'Route' with the URL 'http://php-helloworld-console.apps.lab.example.com'. Other networking details like internal traffic and ports are also listed.

Figure 6.7: Application overview page

From this page, the user can view the route, build, service, and deployment information. The scale tool (arrows) can be used to increase and decrease the number of replicas of the application that are running in the cluster. All of the hyperlinks lead to detailed information about that particular application resource including the ability to manipulate that resource. For example, clicking on the link for the build allows the user to start a new build.

CREATING NEW APPLICATIONS

The user can select the Add to Project link to create a new application. The user can create an application using a template (Source-to-Image), deploy an existing image, and define a new application by importing YAML or JSON formatted resources. Once an application has been created with one of these three methods, it can be managed on the overview page.

OTHER WEB CONSOLE FEATURES

The web console allows the user 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.

► GUIDED EXERCISE

CREATING AN APPLICATION WITH THE WEB CONSOLE

In this exercise, you will create, build, and deploy an application on an OpenShift cluster using the OpenShift Web Console.

RESOURCES

Files:	N/A
Application URL:	http://php-helloworld-console.apps.lab.example.com

OUTCOMES

You should be able to create, build, and deploy an application on an OpenShift cluster using the web console.

BEFORE YOU BEGIN

Get the lab files by executing the lab script:

```
[student@workstation ~]$ lab console setup
```

The lab setup script makes sure that the OpenShift cluster is running.

- 1. Go to the <https://master.lab.example.com> address to access the OpenShift Web Console via a browser. Log in and create a new project.
 - 1.1. Enter the web console URL in the browser and trust the self-signed certificate generated by OpenShift.
 - 1.2. Login with **developer** as the user name and **redhat** as the password.

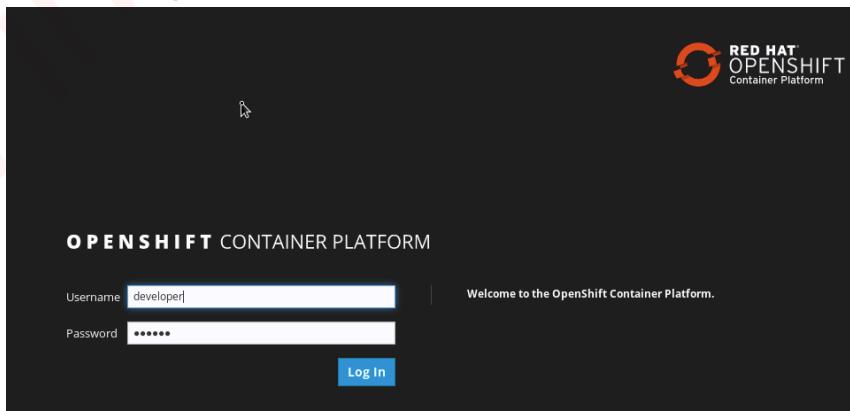


Figure 6.8: Web console login

- 1.3. Create a new project named **console**. You may type any values you wish in the other fields.

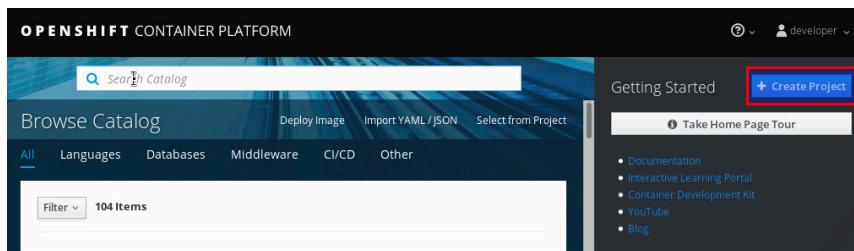


Figure 6.9: Create a new project - step 1

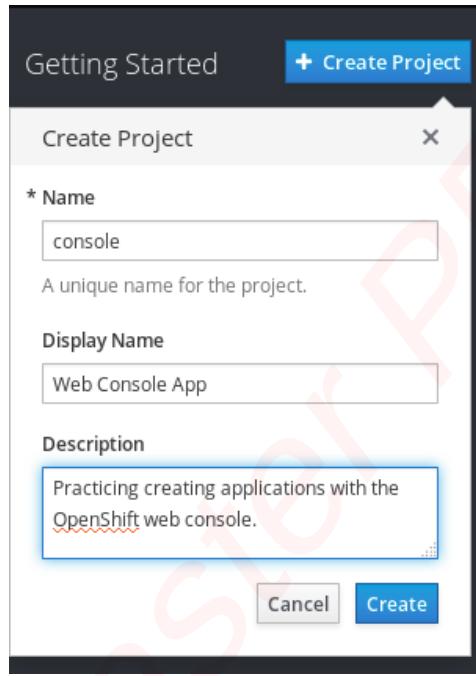


Figure 6.10: Create a new project - step 2

- 1.4. After filling in the details of the appropriate fields, click on the **Create** button of the **Create Project** dialog box.

► 2. Create the new **php-helloworld** application with a PHP template.

2.1. Select the PHP template from the catalog.

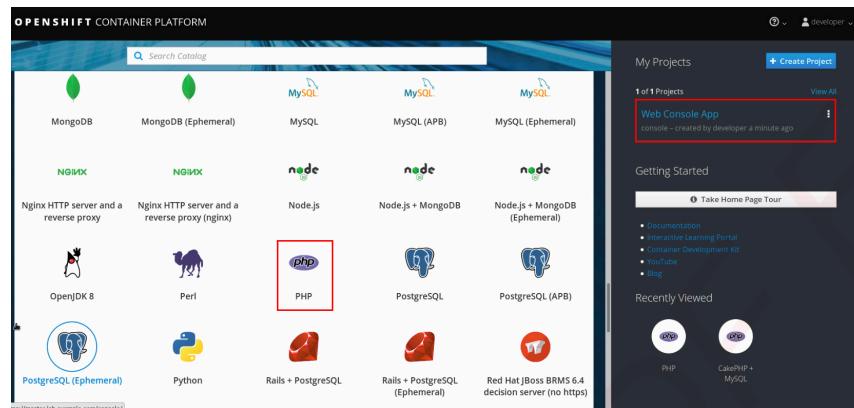


Figure 6.11: Select the PHP template

- 2.2. A PHP dialog box appears. Click on the **Next >** button. Then, select PHP version 7.0 from the drop-down menu Version.

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>

Click on the **Create** button of the **PHP** dialog box.

The screenshot shows the 'PHP' application creation dialog box. It has three tabs at the top: 'Information' (marked with a red box), 'Configuration' (marked with a blue circle), and 'Results' (marked with a green circle). The 'Information' tab contains fields for 'Add to Project' (set to 'Web Console App'), 'Version' (set to '7.0'), 'Application Name' (set to 'php-helloworld'), and 'Git Repository' (set to 'http://services.lab.example.com/php-helloworld'). At the bottom right of the dialog, there are buttons for 'Cancel', '< Back', and a large blue 'Create' button.

Figure 6.12: PHP application details

- 2.3. On the confirmation page, click on the Continue to the project overview link.

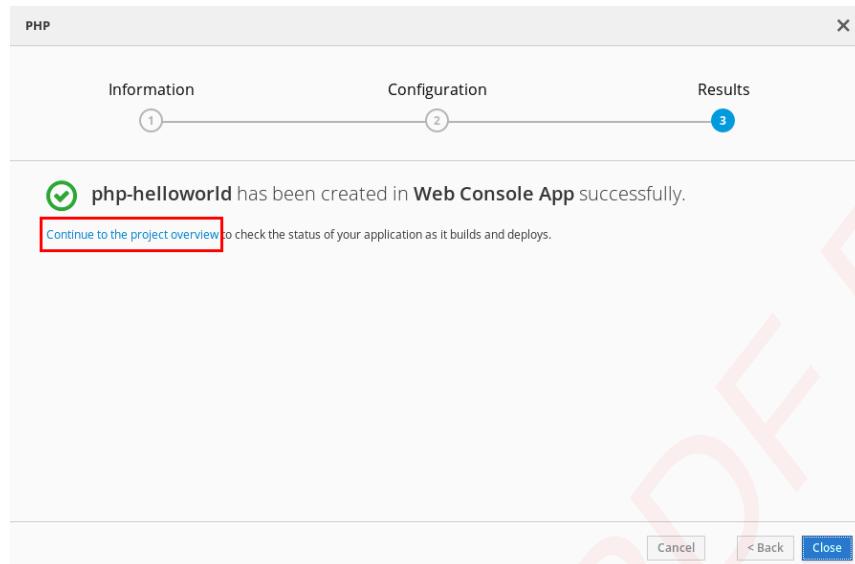


Figure 6.13: Application created confirmation page

- 3. Explore application components from the Overview page. The build may still be running when you reach this page, so the build section may look slightly different from the image below.

The screenshot shows the OpenShift Application Overview page for the 'php-helloworld' application. At the top, it displays the application name 'php-helloworld' and its route URL, <http://php-helloworld-console.apps.lab.example.com>. Below this, under 'DEPLOYMENT CONFIG', it shows 'php-helloworld, #1'. In the 'CONTAINERS' section, there is one pod listed with the image 'console/php-helloworld' and the build 'php-helloworld, #1'. It also shows the source as 'updated app ad6d963' and ports as '8080/TCP'. Under 'NETWORKING', it lists a service named 'php-helloworld' with internal traffic on port 8080/TCP (8080-tcp) and external traffic on port 8080. To the right, there is a 'Scale tool' icon with up and down arrows, and a note indicating 1 pod. A red box highlights the 'Route' URL, another highlights the 'Build and Deployment' section, a third highlights the 'Service' link, and a fourth highlights the 'Scale tool' icon.

Figure 6.14: Application overview page

- 3.1. Identify the components of the application and their corresponding OpenShift and Kubernetes resources.
 - Route URL
Clicking on this link opens a browser tab to view your application.
 - Build
Clicking on the relevant links show the build configuration, specific build information, and build log.
 - Service
Clicking on the relevant link shows the service configuration.
 - Deployment Configuration
Clicking on the relevant links show the deployment configuration and current deployment information.
 - Scale Tool
Clicking on the up arrow will increase the number of running pods. Clicking on the down arrow decreases the number of running pods.
- 3.2. Examine the build log. Under the BUILDS section of the **Overview** page, click on the php-helloworld link. Click the View Log link to view the build log. Return to the **Overview** page by clicking Overview in the left-hand menu.
- 3.3. Examine the deployment configuration. Click on the php-helloworld link just below the label DEPLOYMENT CONFIG in the **Overview** page. Examine the information and features on this page and return to the **Overview** page.

- 3.4. Examine the service configuration. Click on the **php-helloworld** link in the **NETWORKING** section of the **Overview** page. Examine the service configuration page and return to the **Overview** page.
 - 3.5. Click on the route link to view the application output in a new browser tab. This is the URL that appears on the right side at the same line as the title of the application (near the top).
- 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
 1 file changed, 1 insertion(+), 1 deletion(-)
[student@workstation console]$ 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
 eb438ba..d198fb5  master -> master
```

- 4.3. Trigger an application build manually from the web console.

Navigate to the build page from the **Overview** page by clicking on **php-helloworld** link in the **BUILDS** section. Click the **Start Build** button on the upper-right of the

page. Wait for the build to complete. Examine the build log on the build page or the **Overview** page to determine when the build completes.

The screenshot shows the OpenShift web console interface for the 'php-helloworld' application. At the top, it displays the application name and its URL: <http://php-helloworld-console.apps.lab.example.com>. Below this, under 'DEPLOYMENT CONFIG', there is a section for 'php-helloworld, #1'. The 'CONTAINERS' section lists the 'php-helloworld' container with details like the image, build ID, source, and ports. Under 'NETWORKING', it shows a service named 'php-helloworld' with port 8080/TCP mapping to 8080. To the right of the main content, a context menu is open over the deployment config, with the 'Start Build' option highlighted and circled in blue.

Figure 6.15: Start a new application build

- 4.4. Use the route link on the **Overview** page to verify that your code change was deployed.

► 5. Grade your work:

```
[student@workstation php-helloworld]$ lab console grade
Accessing PHP web application..... SUCCESS
```

- 6. Delete the project. Click the OPENSHIFT CONTAINER PLATFORM icon in the web console to go back to the list of projects. Click the menu icon, next to your project name. From the list choose **Delete Project** and enter the name of the project to confirm its deletion.

The screenshot shows the OpenShift Container Platform interface with the title 'OPENSHIFT CONTAINER PLATFORM'. It displays a list of projects under 'My Projects', including 'Web Console App'. A context menu is open over the 'Web Console App' project, with the 'Delete Project' option highlighted and circled in red.

Figure 6.16: Delete the project

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.

RESOURCES	
Files:	N/A
Application URL:	http://temps-ocp.apps.lab.example.com

OUTCOMES

You should be able to create an OpenShift application and access it through a web browser.

Get the lab files by executing the lab script:

```
[student@workstation ~]$ lab openshift setup
```

1. Login as **developer** and create the project **ocp**.
If the **oc login** command prompts about using insecure connections, answer **y** (yes).
2. Create a temperature converter application written in PHP using the **php:5.6** image stream tag. The source code is in the Git repository at <http://services.lab.example.com/temps>. You may use the OpenShift command line interface or the web console to create the application. Make sure a route is created so that you can access the application from a web browser.
3. Test the application in a web browser using the route URL.
4. Grade your work:

```
[student@workstation openshift]$ lab openshift grade  
Accessing PHP web application..... SUCCESS
```

5. Delete the project.

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.

RESOURCES	
Files:	N/A
Application URL:	http://temps-ocp.apps.lab.example.com

OUTCOMES

You should be able to create an OpenShift application and access it through a web browser.

Get the lab files by executing the lab script:

```
[student@workstation ~]$ lab openshift setup
```

1. Login as **developer** and create the project **ocp**.
If the **oc login** command prompts about using insecure connections, answer **y** (yes).
Issue the following commands:

```
[student@workstation openshift]$ oc login -u developer \
https://master.lab.example.com
Login successful.
```

You don't have any projects. You can try to create a new project, by running

```
oc new-project <projectname>
```

```
[student@workstation openshift]$ oc new-project ocp
Now using project "ocp" on server "https://master.lab.example.com:443".
```

You can add applications to this project with the 'new-app' command. For example, try:

```
oc new-app centos/ruby-22-centos7-https://github.com/openshift/ruby-ex.git
```

to build a new example application in Ruby.

2. Create a temperature converter application written in PHP using the **php:5.6** image stream tag. The source code is in the Git repository at <http://services.lab.example.com/temps>. You may use the OpenShift command line interface or the web console to create the application. Make sure a route is created so that you can access the application from a web browser.

If using the command line interface, issue the following commands:

```
[student@workstation openshift]$ oc new-app \
  php:5.6-http://services.lab.example.com/temps
[student@workstation openshift]$ 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
---> Installing application source...
...output omitted...
Pushed 6/6 layers, 100% complete
Push successful
[student@workstation openshift]$ oc get pods -w
NAME      READY     STATUS    RESTARTS   AGE
temps-1-build  0/1      Completed  0          5m
temps-1-h76lt  1/1      Running   0          5m
Ctrl+C
[student@workstation openshift]$ oc expose svc/temps
route "temps" exposed
```

3. Test the application in a web browser using the route URL.

3.1. Discover the URL for the route.

```
[student@workstation openshift]$ oc get route/temps
NAME      HOST/PORT           PATH      SERVICES   PORT
TERMINATION  WILDCARD
temps      temps-ocp.apps.lab.example.com      temps      8080-tcp
None
```

3.2. Open Firefox with `http://temps-ocp.apps.lab.example.com` URL to verify that the temperature converter application works.

4. Grade your work:

```
[student@workstation openshift]$ lab openshift grade
Accessing PHP web application..... SUCCESS
```

5. Delete the project.

Delete the project by running the `oc delete project` command.

```
[student@workstation openshift]$ oc delete project ocp
```

This concludes the lab.

SUMMARY

In this chapter, you learned:

- The OpenShift command line client **oc** is used to perform the following tasks in an OpenShift cluster:
 - Logging in and out of an OpenShift cluster.
 - Creating, changing, and deleting projects.
 - Creating applications inside a project, including creating a deployment configuration from a container image, or a build configuration from application source and all associated resources.
 - Creating, deleting, inspecting, editing, and exporting individual resources such as pods, services, and routes inside a project.
 - Scaling applications.
 - Starting new deployments and builds.
 - Checking logs from application pods, deployments, and build operations.
- The OpenShift Container Platform organizes entities in the OpenShift cluster as objects stored on the master node. These are collectively known as **resources**. The most common ones are:
 - Pod
 - Label
 - Persistent Volume (PV)
 - Persistent Volume Claim (PVC)
 - Service
 - Route
 - Replication Controller (RC)
 - Deployment Configuration (DC)
 - Build Configuration (BC)
 - Project
- 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.
- **Source-to-Image (S2I)** is a facility that makes it easy to build a container image from application source code. This facility takes an application's source code from a Git server, 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.

- 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 the considerations for containerizing applications with multiple container images.
- Deploy a multi-container application with user-defined Docker network.
- Deploy a multi-container application on OpenShift using a template.

SECTIONS

- Considerations for Multi-Container Applications (and Quiz)
- Deploying a Multi-Container Application with Docker (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.
- Inject environment variables into a container.
- Describe the architecture of the To Do List application.

DISCOVERING SERVICES IN A MULTI-CONTAINER APPLICATION

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 in which each container must be able to communicate with other containers in order to use the services on which it depends.

For example, consider an application which is 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 is restarted, so a process is needed to ensure any change in IP triggers an update to existing containers.

Both Docker and Kubernetes 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 section.

INJECTING ENVIRONMENT VARIABLES INTO A DOCKER CONTAINER

It is a recommended practice to parameterize application connection information to outside services, and one common way to do that is the usage of operating system environment variables. The **docker** command provides a few options for interacting with container environment variables:

- The **docker run** command provides the **-e** option to define environment variables when starting a container, and this could be used to pass parameters to an application such as a database server IP address or user credentials. The **-e** option can be used multiple times to define more than one environment variable for the same container.
- The **docker inspect** command can be used to check a running container for environment variables specified either when starting the container or defined by the container image **Dockerfile** instructions. However, it does not show environment variables inherited by the container by the operating system or defined by shell scripts inside the image.
- The **docker exec** command can be used to inspect all environment variables known to a running container using regular shell commands. For example:

```
$ docker exec mysql env | grep MYSQL
```

Additionally, the **Dockerfile** for a container image may contain instructions related to the management of a container environment variables. Use the **ENV** instruction to expose an environment variable to the container:

```
ENV MYSQL_ROOT_PASSWORD="my_password"
ENV MYSQL_DATABASE="my_database"
```

It is recommended to declare all environment variables using only one **ENV** instruction to avoid creating multiple layers:

```
ENV MYSQL_ROOT_PASSWORD="my_password" \
    MYSQL_DATABASE="my_database"
```

COMPARING PLAIN DOCKER AND KUBERNETES

Using environment variables allows you to share information between containers with Docker, which makes service discovery technically possible. 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.
- **Words separated with underscore:** Any environment variable created by a service normally are created with multiple words and they are 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 are declared with 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 the **<SERVICE_NAME>_SERVICE_PORT**.

- <SERVICE_NAME>_PORT_<PORT_NUMBER>_<PROTOCOL>_ADDR: Defines an alias for the <SERVICE_NAME>_SERVICE_HOST.

Note that the variables following the convention <SERVICE_NAME>_PORT_* emulate the variables created by Docker for associated containers in the pod.

For instance, if the following service is deployed on Kubernetes:

```
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

On the basis of the protocol, IP address and port number as set against <SERVICE_NAME>_PORT environment variable, the other relevant <SERVICE_NAME>_PORT_* environment variables' names are set. For example, **MYSQL_PORT=tcp://10.0.0.11:3306** variable leads to the creation of the 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 is not set against the said environment variable, Kubernetes uses TCP protocol and assigns the variable names accordingly.

EXAMINING THE To Do List APPLICATION

Many labs from this course are based on a To Do List application. This application is divided in three tiers as illustrated by the following figure:

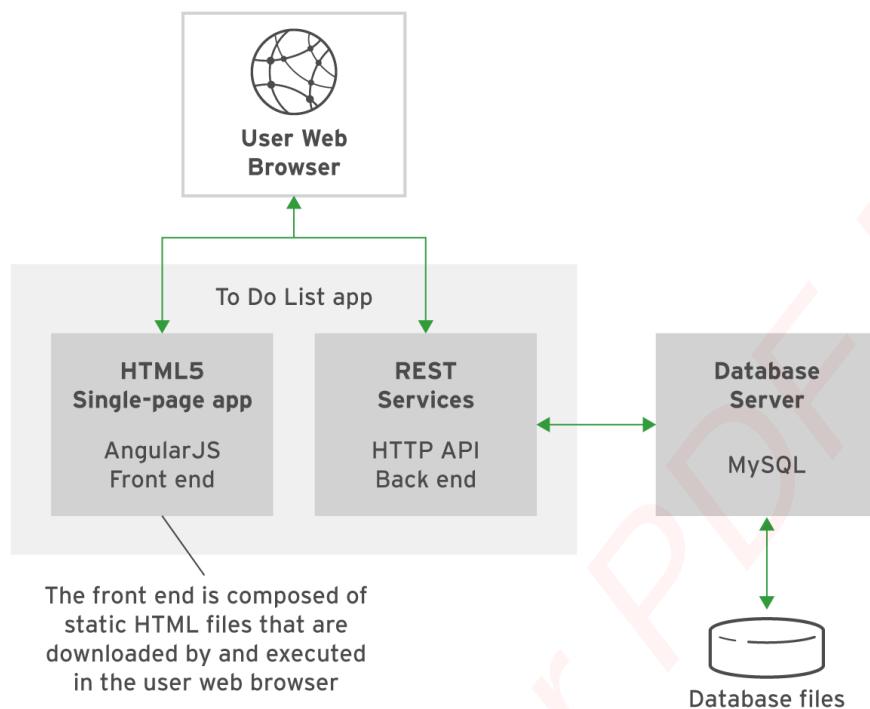


Figure 7.1: To Do List application logical architecture

- The presentation tier is built as a single-page HTML5 front-end using **AngularJS**.
- The business tier is built as an HTTP API back-end, with **Node.js**.
- Persistence tier is based on a **MySQL** database server.

The following figure is a screen capture of the application web interface.

To Do List Application

To Do List

Id	Description	Done	
1	Pick up new...	false	×
2	Buy groceries	true	×

[First](#) [Previous](#) [1](#) [Next](#) [Last](#)

Add Task

Description:

Add Description.

Completed:

[Clear](#) [Save](#)

Figure 7.2: 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 how 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 presentation and business tiers so they can be deployed into different containers.

The sources of both of these application versions are available under *todoapp* GIT repository at:
<http://services.lab.example.com/todoapp/apps/nodejs>

► QUIZ

MULTI-CONTAINER APPLICATION CONSIDERATIONS

Choose the correct answer(s) to the following questions:

► 1. **Why is it difficult for Docker containers to communicate with each other?**

- a. There is no connectivity between containers using the default Docker networking.
- b. The container's firewall must always be disabled in order to enable communication between containers.
- c. Containers use dynamic IP addresses and host names, so it is difficult for one container to reliably find another container's IP address without explicit configuration.
- d. Containers require a VPN client and server in order to connect to each other.

► 2. **What are three benefits of using a multi-container architecture? (Choose three.)**

- a. Keeping images as simple as possible by only including a single application or service in each container provides for easier deployment, maintenance, and administration.
- b. The more containers used in an application, the better the performance will be.
- c. When using multiple containers for an application, each layer of the application can be scaled independently from the others, thus conserving resources.
- d. Monitoring, troubleshooting, and debugging are simplified when a container only provides a single purpose.
- e. Applications using multiple containers typically contain fewer defects than those built inside a single container.

► 3. **How does Kubernetes solve the issue of service discovery using environment variables?**

- a. Kubernetes automatically propagates all environment variables to all pods ensuring the required environment variables are available for containers to leverage.
- b. Controllers are responsible for sharing environment variables between pods.
- c. Kubernetes maintains a list of all environment variables and pods can request them as needed.
- d. Kubernetes automatically injects environment variables for all services in a given namespace into all the pods running on that same namespace.

► SOLUTION

MULTI-CONTAINER APPLICATION CONSIDERATIONS

Choose the correct answer(s) to the following questions:

► 1. Why is it difficult for Docker containers to communicate with each other?

- a. There is no connectivity between containers using the default Docker networking.
- b. The container's firewall must always be disabled in order to enable communication between containers.
- c. Containers use dynamic IP addresses and host names, so it is difficult for one container to reliably find another container's IP address without explicit configuration.
- d. Containers require a VPN client and server in order to connect to each other.

► 2. What are three benefits of using a multi-container architecture? (Choose three.)

- a. Keeping images as simple as possible by only including a single application or service in each container provides for easier deployment, maintenance, and administration.
- b. The more containers used in an application, the better the performance will be.
- c. When using multiple containers for an application, each layer of the application can be scaled independently from the others, thus conserving resources.
- d. Monitoring, troubleshooting, and debugging are simplified when a container only provides a single purpose.
- e. Applications using multiple containers typically contain fewer defects than those built inside a single container.

► 3. How does Kubernetes solve the issue of service discovery using environment variables?

- a. Kubernetes automatically propagates all environment variables to all pods ensuring the required environment variables are available for containers to leverage.
- b. Controllers are responsible for sharing environment variables between pods.
- c. Kubernetes maintains a list of all environment variables and pods can request them as needed.
- d. Kubernetes automatically injects environment variables for all services in a given namespace into all the pods running on that same namespace.

DEPLOYING A MULTI-CONTAINER APPLICATION WITH DOCKER

OBJECTIVES

After completing this section, students should be able to deploy a multi-container application with a user-defined Docker network.

USER-DEFINED DOCKER NETWORK

Standalone containers in an application stack can communicate with each other using IP addresses. However, when a container restarts, there is no guarantee that the IP address of the container is unchanged. This may lead to broken communication between the containers of the application stack, as the destination IP address that the other container uses may no longer be valid.

One way Docker solves this issue is by implementing *user-defined* bridge networks. User-defined bridge networks allow multiple standalone containers to talk to each other using container names. The container names are unlikely to change in the event of a container restart.

All containers attached to the same user-defined bridge have access to the uniform DNS records maintained by the built-in DNS server of the Docker host. These entries are dynamically updated with a containers' IP addresses and host names (container names) as they go in and out of the bridge network. This allows the containers to communicate using static container names.



NOTE

A user-defined bridge network is one method of Docker networking. This course covers the user-defined bridge networks of Docker among other methods, which are beyond the scope of this course. Consult the reference list *References* for more information about Docker networks.

MANAGING DOCKER NETWORKS

To manage Docker networks, use the **docker network** command. Run the **docker network --help** command to get help on command usage.

- To list available networks, use **docker network ls** command.

```
$ docker network ls
NETWORK ID      NAME          DRIVER      SCOPE
...output omitted...
d531b5e4dbe   bridge        bridge      local
...output omitted...
```

By default, Docker creates a user-defined bridge network named **bridge**. This bridge can be used for testing temporary container connections. However, using the default bridge network in production is not a recommended practice.

- To examine a network, use **docker inspect network name** command.

```
$ docker inspect do180-bridge
[
```

```
{
  "Name": "do180-bridge❶",
  "Id": "7dae997d41f21b7a1b17cd5c3e84ad89c3a38fc05852d44eef9fe4cd6dbe61f2",
  "Created": "2018-06-21T19:09:25.41450065+05:30",
  "Scope": "local",
  "Driver": "bridge",
  "EnableIPv6": false,
  "IPAM": {
    "Driver": "default",
    "Options": {},
    "Config": [
      {
        "Subnet": "172.22.0.0/16❷",
        "Gateway": "172.22.0.1❸"
      }
    ]
  },
  "Internal": false,
  "Attachable": true❹,
  "Containers": {}❺,
  "Options": {},
  "Labels": {}
}
]
```

- ❶ Shows the name of the bridge network.
- ❷ Shows the subnet for the network that the containers use.
- ❸ Shows the gateway IP for all the containers in the bridge. The IP address is the address of the bridge itself.
- ❹ If set to **true**, containers can dynamically join the bridge.
- ❺ List the containers currently connected in the bridge.

- To create a network, use the **docker network create network name** command.

```
$ docker network create --attachable do180-bridge
```

The **--attachable** option allows containers to join the bridge dynamically. The default network type is set to **bridge**. Override this default behavior using **--type** option.

- Use **--network network name** option with **docker run** command to specify the intended Docker network for the container.
- To delete a network, use the **docker network rm network name** command.

```
$ docker network rm do180-bridge
```

SERVICE DISCOVERY WITH USER-DEFINED NETWORKING

Containers within the same user-defined bridge network can connect to each other using the supplied container names. These container names, which can be supplied by the user, are resolvable host names that other containers can resolve. For example, a container named

web_container is able to resolve **db_container**, and the other way around, if both belong to the same user-defined bridge network.

Each container exposes all ports to other containers in the same bridge. As such, services are readily accessible within the same user-defined bridge network. The containers expose ports to external networks only on the basis of explicit configuration.

The user-defined bridge network is particularly relevant in running a multi-container application. Its dynamic name resolution feature helps to resolve the issue of service discoverability between containers in a multi-container application.

The following procedure demonstrates the usage of user-defined networks.

- The following command produces a list of available Docker networks:

```
[student@workstation ~]$ docker network ls
NETWORK ID      NAME      DRIVER      SCOPE
...output omitted...
1876bcd67a0f    do180-bridge    bridge      local
...output omitted...
```

- The following command runs a database (MySQL) container in the background and attaches it to **do180-bridge** Docker network. The **-e** options specifies the required environment variables to run the database container image.

```
[student@workstation ~]$ docker run -d --name db_container \
-e MYSQL_USER=user1 -e MYSQL_PASSWORD=mypa55 \
-e MYSQL_DATABASE=items -e MYSQL_ROOT_PASSWORD=r00tpa55 \
--network do180-bridge \
test/db:1.0
```

- The following command runs another container in the background, using the standard **rhel7** image, with **mysql** client installed in it. This container is attached to the **do180-bridge** Docker network.

```
[student@workstation ~]$ docker run -d --name web_container \
--network do180-bridge test/web:1.0
```

- The following command executes an interactive shell inside the **web_container** container. From this shell, MySQL statements can be executed seamlessly.

```
[student@workstation ~]$ docker exec -it web_container /bin/bash
[root@cf36ce51a8b8 /]# mysql -uuser1 -pmypa55 -hdb_container -e 'show databases;'
+-----+
| Database      |
+-----+
| information_schema |
| items          |
+-----+
```

Notice that the hostname of the database server, **db_container**, resolves to the IP address of the database container. This indicates that the user-defined bridge network allows containers to reach each other using the static container names supplied by the user.



REFERENCES

Further information about Docker networks is available in the *Networking Overview* section of the Docker documentation at
<https://docs.docker.com/network/>

► 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.

RESOURCES	
Files:	/home/student/D0180/labs/networking-containers
Application URL:	http://127.0.0.1:30080/todo/
Resources:	RHSCl MySQL 5.7 image (rhscl/mysql-57-rhel7) Custom MySQL 5.7 image (do180/mysql-57-rhel7) RHEL 7.5 image (rhel7.5) Custom Node.js 4.0 image (do180/nodejs)

OUTCOMES

You should be able to network containers to create a multi-tiered application.

BEFORE YOU BEGIN

The workstation requires the To Do List application source code and lab files. To set up the environment for the exercise, run the following command:

```
[student@workstation ~]$ lab networking-containers setup
```

► 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/networking-containers/images/mysql/Dockerfile**.

1.2. Build the MySQL database image.

Examine the **/home/student/D0180/labs/networking-containers/images/mysql/build.sh** script to see how the image is built. To build the base image, run the **build.sh** script.

```
[student@workstation images]$ cd ~/D0180/labs/networking-containers/images/
```

```
[student@workstation images]$ cd mysql
[student@workstation mysql]$ ./build.sh
```

13. Wait for the build to complete, and then run the following command to verify that the image is built successfully:

```
[student@workstation mysql]$ docker images
REPOSITORY          TAG      IMAGE ID   CREATED        SIZE
do180/mysql-57-rhel7    latest   b0679ef6  8 seconds ago  405 MB
registry[...]com/rhscl/mysql-57-rhel7 latest   4ae3a3f4  9 months ago  405 MB
```

► 2. Build the To Do List application parent image using the Node.js Dockerfile.

- 2.1. Review the Dockerfile.

Using your preferred editor, open and examine the completed Dockerfile located at **/home/student/D0180/labs/networking-containers/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 is created to run the Node.js application along with the **app-root** directory using the **RUN** command.
- The **enable-rh-nodejs4.sh** script is added to **/etc/profile.d/** to run automatically on login using the **ADD** command.
- The **USER** command is used to switch to the newly created **appuser**
- The **WORKDIR** command is used to switch to the **\$HOME** directory for application execution.
- The **ONBUILD** command is used to define actions that run when any child container image is built from this image. In this case, the **COPY** command copies the **run.sh** file and the **build** directory into **\$HOME** and the **RUN** command runs **scl enable rh-nodejs4** as well as **npm install**. The **npm install** has a local Node.js module registry specified to override the default behavior of using **http://npmjs.registry.org** to download the dependent node modules, so that the node application can be built without accessing Internet access.

- 2.2. Build the parent image.

Examine the script located at **/home/student/D0180/labs/networking-containers/images/nodejs/build.sh** to see how the image is built. To build the base image, run the **build.sh** script.

```
[student@workstation images]$ cd ~/D0180/labs/networking-containers/images/
[student@workstation images]$ cd nodejs
```

```
[student@workstation nodejs]$ ./build.sh
```

- 2.3. Wait for the build to complete, and then run the following command to verify that the image is built successfully.

```
[student@workstation nodejs]$ docker images
REPOSITORY           TAG      IMAGE ID   CREATED        SIZE
do180/nodejs         latest   622e8a4d  5 minutes ago  407 MB
do180/mysql-57-rhel7 latest   b0679ef6  24 minutes ago  405 MB
registry[...]/rhscl/mysql-57-rhel7  latest   4ae3a3f4  9 months ago   405 MB
registry.lab.example.com/rhel7       7.3     93bb76dd  12 months ago  193 MB
```

- 3. Build the To Do application child image using the Node.js Dockerfile.

- 3.1. Review the Dockerfile.

Using your preferred editor, open and examine the completed Dockerfile located at **/home/student/D0180/labs/networking-containers/deploy/nodejs/Dockerfile**.

- 3.2. Build the child image.

Examine the **/home/student/D0180/labs/networking-containers/deploy/nodejs/build.sh** script to see how the image is built. Run the following commands in order to build the child image.

```
[student@workstation nodejs]$ cd ~/D0180/labs/networking-containers/deploy/
[student@workstation deploy]$ cd nodejs
[student@workstation nodejs]$ ./build.sh
```



NOTE

The **build.sh** script lowers restriction for write access to the build directory to allow the installation of dependencies by non-root users.

- 3.3. Wait for the build to complete and then run the following command to verify that the image is built successfully:

```
[student@workstation nodejs]$ docker images
REPOSITORY           TAG      IMAGE ID   CREATED        SIZE
do180/todonodejs    latest   21b7cdc6  6 seconds ago  435 MB
do180/nodejs         latest   622e8a4d  11 minutes ago  407 MB
do180/mysql-57-rhel7 latest   b0679ef6  29 minutes ago  405 MB
registry[...]/rhscl/mysql-57-rhel7  latest   4ae3a3f4  9 months ago   405 MB
registry.lab.example.com/rhel7       7.3     93bb76dd  12 months ago  193 MB
```

► 4. Explore the Environment Variables.

Inspect the environment variables that allow the Node.js REST API container to communicate with the MySQL container.

- 4.1. View the file `/home/student/D0180/labs/networking-containers/deploy/nodejs/nodejs-source/models/db.js` that holds the database configuration as given below.

```
module.exports.params = {
  dbname: process.env.MYSQL_DATABASE,
  username: process.env.MYSQL_USER,
  password: process.env.MYSQL_PASSWORD,
  params: {
    host: "mysql",
    port: "3306",
    dialect: 'mysql'
  }
};
```

- 4.2. Notice the environment variables being used by the REST API. These variables are exposed to the container using `-e` options with the `docker run` command in this guided exercise. The environment variables are described below.

MYSQL_DATABASE

This is the name of the MySQL database in the `mysql` container.

MYSQL_USER

The name of the database user that the `todoapi` container uses to run MySQL commands.

MYSQL_PASSWORD

The password of the database user that the `todoapi` container uses for authentication 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 the `db.js` file is the resolvable hostname of `mysql` container. Any container is reachable by its name from other containers that belong to the same user-defined Docker network.

► 5. Create and inspect an attachable user-defined bridge network called `do180-bridge`.

- 5.1. Run the following command to create the `do180-bridge` bridge.

```
[student@workstation nodejs]$ docker network create --attachable do180-bridge
6cf8659f66241bccbb5cf38cdcb052d0f7846e632c868680221ef6f818e04a9
```



NOTE

The `--attachable` option allows containers to join the bridge dynamically.

- 5.2. Verify the docker network `do180-bridge` using `docker inspect` command.

```
[student@workstation nodejs]$ docker inspect do180-bridge
```

```
[
  {
    "Name": "do180-bridge",
    "Id": "6cf8...04a9",
    "Created": "2018-06-20T21:02:48.165825856+05:30",
    "Scope": "local",
    "Driver": "bridge",
    "EnableIPv6": false,
    "IPAM": {
      "Driver": "default",
      "Options": {},
      "Config": [
        {
          "Subnet": "172.18.0.0/16",
          "Gateway": "172.18.0.1"
        }
      ]
    },
    "Internal": false,
    "Attachable": true,
    "Containers": {},
    "Options": {},
    "Labels": {}
  }
]
```

- 6. Modify the existing script to create containers in the same bridge, **do180-bridge**, created 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, giving enough time for each container to start.
- 6.1. Edit the **run.sh** file located at **/home/student/DO180/labs/networking-containers/deploy/nodejs/networked/** to insert the **docker run** command at the appropriate line for invoking **mysql** container. The following screen shows the exact **docker** command to insert into the file.

```
docker run -d --name mysql -e MYSQL_DATABASE=items -e MYSQL_USER=user1 \
-e MYSQL_PASSWORD=mypa55 -e MYSQL_ROOT_PASSWORD=r00tpass55 \
-v $PWD/work/data:/var/lib/mysql/data \
-v $PWD/work/init:/var/lib/mysql/init -p 30306:3306 \
--network do180-bridge 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.

- 6.2. In the same **run.sh** file, insert another **docker run** command at the appropriate line to run the **todoapi** container. The following screen shows the **docker** command to insert into the file.

```
docker run -d --name todoapi -e MYSQL_DATABASE=items -e MYSQL_USER=user1 \
-e MYSQL_PASSWORD=mypa55 -p 30080:30080 \
```

```
--network do180-bridge do180/todonodejs
```

**NOTE**

After each **docker 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 prior to re-running the **run.sh** script.

- 6.3. Verify that your **run.sh** script matches the solution script located at **/home/student/D0180/solutions/networking-containers/deploy/nodejs/networked/run.sh**.
- 6.4. Save the file and exit the editor.

▶ **7.** Run the containers.

- 7.1. Use the following command to execute the script that you updated to run the **mysql** and **todoapi** containers, connected to the **do180-bridge** network.

```
[student@workstation nodejs]$ cd \
/home/student/D0180/labs/networking-containers/deploy/nodejs/networked
[student@workstation networked]$ ./run.sh
```

- 7.2. Verify that the containers started successfully.

```
[student@workstation networked]$ docker ps
CONTAINER ID IMAGE COMMAND STATUS PORTS
NAMES
2c61a089105d do180/todonodejs "scl enab..." Up 7 minutes 0.0.0.0:30080->30080/tcp todoapi
0b9fa2821ba1 do180/mysql-57-rhel7 "container..." Up 7 minutes 0.0.0.0:30306->3306/tcp mysql
```

▶ **8.** Examine the environment variables of the API container.

Run the following command to explore the environment variables that exposed in the API container.

```
[student@workstation networked]$ docker 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
```

► 9. Test the application.

- 9.1. 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.

- 9.2. 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 (/).

- 9.3. Verify that the correct images were built and that the application is running.

```
[student@workstation networked]$ lab networking-containers grade
```

► 10. Clean up.

- 10.1. Navigate to the current user's home directory and stop the running containers:

```
[student@workstation networked]$ cd ~
[student@workstation ~]$ docker stop todoapi mysql
```

- 10.2. Delete the Docker bridge:

```
[student@workstation ~]$ docker network rm do180-bridge
```

- 10.3. Remove all containers and container images from the Docker builds:

```
[student@workstation ~]$ docker rm $(docker ps -aq)
[student@workstation ~]$ docker rmi $(docker images -q)
```

This concludes the guided exercise.

DEPLOYING A MULTI-CONTAINER APPLICATION ON OPENSHIFT

OBJECTIVE

After completing this section, students should be able to deploy a multi-container application on OpenShift using a template.

EXAMINING THE SKELETON OF A TEMPLATE

OpenShift Container Platform contains a facility to deploy applications using *templates*. A template can include any OpenShift resource, assuming users have permission to create them within a project.

A template describes a set of related resource definitions to be created together, as well as a set of parameters for those objects.

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:

NAME	DESCRIPTION
cakephp-mysql-persistent	An example CakePHP application...
dancer-mysql-persistent	An example Dancer application...
django-psql-persistent	An example Django application...
jenkins-ephemeral	Jenkins service, without persistent storage....
jenkins-persistent	Jenkins service, with persistent storage....
jenkins-pipeline-example	This example showcases the new Jenkins...
logging-deployer-account-template	Template for creating the deployer...
logging-deployer-template	Template for running the aggregated...
mariadb-persistent	MariaDB database service, with persistent storage...
mongodb-persistent	MongoDB database service, with persistent storage...
mysql-persistent	MySQL database service, with persistent storage...
nodejs-mongo-persistent	An example Node.js application with a MongoDB...
postgresql-persistent	PostgreSQL database service...
rails-pgsql-persistent	An example Rails application...

The following listing shows a template definition:

```
{
```

```

"kind": "Template",
"apiVersion": "v1",
"metadata": {
    "name": "mysql-persistent", ①
    "creationTimestamp": null,
    "annotations": {
        "description": "MySQL database service, with persistent storage...",
        "iconClass": "icon-mysql-database",
        "openshift.io/display-name": "MySQL (Persistent)",
        "tags": "database,mysql" ②
    }
},
"message": "The following service(s) have been created ...",
"objects": [
{
    "apiVersion": "v1",
    "kind": "Service",
    "metadata": {
        "name": "${DATABASE_SERVICE_NAME}" ③
    },
    ...
    Service attributes omitted ...
},
{
    "apiVersion": "v1",
    "kind": "PersistentVolumeClaim",
    "metadata": {
        "name": "${DATABASE_SERVICE_NAME}"
    },
    ...
    PVC attributes omitted ...
},
{
    "apiVersion": "v1",
    "kind": "DeploymentConfig",
    "metadata": {
        "name": "${DATABASE_SERVICE_NAME}"
    },
    "spec": {
        "replicas": 1,
        "selector": {
            "name": "${DATABASE_SERVICE_NAME}"
        },
        "strategy": {
            "type": "Recreate"
        },
        "template": {
            "metadata": {
                "labels": {
                    "name": "${DATABASE_SERVICE_NAME}"
                }
            },
            ...
            Other pod and Deployment attributes omitted
        }
    }
},
"parameters": [

```

```

...
{
    "name": "DATABASE_SERVICE_NAME",
    "displayName": "Database Service Name",
    "description": "The name of the OpenShift Service exposed for the
database.",
    "value": "mysql", ❸
    "required": true
},
{
    "name": "MYSQL_USER",
    "displayName": "MySQL Connection Username",
    "description": "Username for MySQL user that will be used for
accessing the database.",
    "generate": "expression",
    "from": "user[A-Z0-9]{3}",
    "required": true
},
{
    "name": "MYSQL_PASSWORD",
    "displayName": "MySQL Connection Password",
    "description": "Password for the MySQL connection user.",
    "generate": "expression",
    "from": "[a-zA-Z0-9]{16}", ❹
    "required": true
},
{
    "name": "MYSQL_DATABASE",
    "displayName": "MySQL Database Name",
    "description": "Name of the MySQL database accessed.",
    "value": "sampledb",
    "required": true
},
...
}

```

- ❶ Defines the template name.
- ❷ Defines a list of arbitrary tags that this template will have in the UI.
- ❸ Defines an entry for using the value assigned by the parameter **DATABASE_SERVICE_NAME**.
- ❹ Defines the default value for the parameter **DATABASE_SERVICE_NAME**.
- ❺ Defines an expression used to generate a random password if one is not specified.

It is also possible to upload new templates from a file to the OpenShift cluster so that other developers can build applications from the template. This can be done using the **oc create** command, as shown in the following example:

```
[student@workstation deploy-multicontainer]$ oc create -f todo-template.json
template "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.json -n openshift
```

**IMPORTANT**

Any template created under the **openshift** namespace (OpenShift project) is available in the web console under the dialog box accessible via Add to Project → Select from Project menu item. Moreover, any template that gets 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 values 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 the 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: 2 weeks ago
Labels: <none>
Description: MySQL database service, with persistent storage. For more
information ...
Annotations: iconClass=icon-mysql-database
            openshift.io/display-name=MySQL (Persistent)
            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
  ... SOME OUTPUT OMITTED ...
  Name: MYSQL_VERSION
  Display Name: Version of MySQL Image
  Description: Version of MySQL image to be used (5.5, 5.6 or latest).
  Required: true
  Value: 5.6

Object Labels: template=mysql-persistent-template
```

```
Message: The following service(s) have been created 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, including OpenShift
considerations, see https://github.com/sclorg/mysql-container/blob/master/5.6/README.md.

Objects:
Service ${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
MEMORY_LIMIT   Maximum amount of memory the container can use.
               512Mi
NAMESPACE      The OpenShift Namespace where the ImageStream resides.
               openshift
DATABASE_SERVICE_NAME The name of the OpenShift Service exposed for the
                     database.
                     mysql
MYSQL_USER     Username for MySQL user that will be used for accessing
                     the database.
                     expression user[A-Z0-9]{3}
MYSQL_PASSWORD Password for the MySQL connection user.
                     expression [a-zA-Z0-9]{16}
MYSQL_DATABASE Name of the MySQL database accessed.
VOLUME_CAPACITY Volume space available for data, e.g. 512Mi, 2Gi, 1Gi
MYSQL_VERSION  Version of MySQL image to be used (5.5, 5.6 or latest).
               5.6
```

Persistent Volume (PV) and Persistent Volume Claim (PVC)

Any data getting into a container while it is running, is lost with the deletion of the container. The reason behind this is the volatile storage space that the containers operate in. In many circumstances, a persistent storage is desired for the containers to avoid losing data. For example, often while running a database container, having the data entries of the database persist independently of the lifetime of the container is paramount. Both Docker and OpenShift Container Platform address the storage issue and provide solutions in different ways.

Docker uses its bind mount feature to define persistent storage for a container. Bind mounts may create an empty directory on the Docker host and map it to the container, or map an existing directory to the container. Bind-mounting an existing directory requires **svirt_sandbox_file_t** SELinux context to be set on it.

OpenShift Container Platform implements *Persistent Volumes* (PVs) and *Persistent Volume Claims* (PVCs) to offer persistent storage to pod containers. A *Persistent Volume* is a cluster-wide resource in an OpenShift cluster, usually backed by an external storage array. It supports various storage back-ends such as NFS, iSCSI, Red Hat Gluster Storage, Red Hat Ceph Storage through the use of plug-ins. As persistent volumes are cluster-wide objects, only users with *cluster-admin* role can

manage these. A *Persistent Volume Claim* is a user's request from the scope of an OpenShift project or Kubernetes namespace to use persistent volumes. It binds a persistent volume to a pod.

PROCESSING A TEMPLATE USING THE CLI

A template should be processed to generate a list of resources to create a new application. The **oc process** command is responsible for processing a template:

```
$ oc process -f filename
```

The previous command processes a template from a JSON or YAML resource definition file and returns the list of resources to standard output.

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 the list of resources to standard output. This output can be redirected to a file:

```
$ oc process -f filename -o json > myapp.json
```

Templates can generate different values based on the parameters. To override a parameter, use the **-p** option followed by a **<name>=<value>** pair.

```
$ oc process -f mysql.json \
-p MYSQL_USER=dev -p MYSQL_PASSWORD=$P4SSD -p MYSQL_DATABASE=bank \
-p VOLUME_CAPACITY=10Gi > mysqlProcessed.json
```

To create the application, use the generated JSON resource definition file:

```
$ oc create -f mysqlProcessed.json
```

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.json \
-p MYSQL_USER=dev -p MYSQL_PASSWORD=$P4SSD -p MYSQL_DATABASE=bank -p \
VOLUME_CAPACITY=10Gi \
| oc create -f -
```

It is not possible to process a template from the **openshift** project as a regular user using the **oc process** command.

```
$ oc process mysql-persistent -n openshift \
-p MYSQL_USER=dev -p MYSQL_PASSWORD=$P4SSD -p MYSQL_DATABASE=bank \
-p VOLUME_CAPACITY=10Gi | oc create -f -
```

The previous command returns an error:

```
error processing the template "mysql-persistent": User "regularUser" cannot create  
processed templates in project "openshift"
```

One way to solve this problem is to export the template to a file and then process the template using that file:

```
$ oc -o json export template mysql-persistent \  
-n openshift > mysql-persistent-template.json
```

```
$ oc process -f mysql-persistent-template.json \  
-p MYSQL_USER=dev -p MYSQL_PASSWORD=$P4SSD -p MYSQL_DATABASE=bank \  
-p VOLUME_CAPACITY=10Gi | oc create -f -
```

Another way to solve this problem is to use two slashes (>//) to provide the namespace as part of the template name:

```
$ oc process openshift//mysql-persistent-template \  
-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 -n openshift \  
-p MYSQL_USER=dev -p MYSQL_PASSWORD=$P4SSD -p MYSQL_DATABASE=bank \  
-p VOLUME_CAPACITY=10Gi
```



REFERENCES

Further information about templates can be found in the *Templates* section of the OpenShift Container Platform documentation:

Architecture

https://access.redhat.com/documentation/en-us/openshift_container_platform/3.5/html/architecture/

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/3.5/html/developer_guide/

► 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 the resources the application needs to run.

RESOURCES	
Files:	/home/student/D0180/labs/openshift-template
Application URL:	http://todoapi-template.apps.lab.example.com/todo/

OUTCOMES

You should be able to build and deploy an application in OpenShift Container Platform using a provided JSON template.

BEFORE YOU BEGIN

The workstation requires the To Do List application source code and lab files. Both Docker and the OpenShift cluster need to be running. To download the lab files and verify the status of the Docker service and the OpenShift cluster, run the following command in a new terminal window.

```
[student@workstation ~]$ lab openshift-template setup
```

- 1. Build the database container image and publish it to the private registry.

- 1.1. Build the MySQL database image.

Examine the `/home/student/D0180/labs/openshift-template/images/mysql/build.sh` script to see how the image is built. To build the base image, run the `build.sh` script.

```
[student@workstation ~]$ cd ~/D0180/labs/openshift-template/images/mysql
[student@workstation mysql]$ ./build.sh
```

- 1.2. Push the image to the private registry running in the **services** VM.

To make the image available as a template 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]$ docker tag \
do180/mysql-57-rhel7 \
registry.lab.example.com/do180/mysql-57-rhel7
[student@workstation mysql]$ docker push \
registry.lab.example.com/do180/mysql-57-rhel7
The push refers to a repository [registry.lab.example.com/do180/mysql-57-rhel7]
b3838c109ba6: Pushed
```

```
a72cf1d1d969d: Mounted from rhscl/mysql-57-rhel7
9ca8c628d8e7: Mounted from rhscl/mysql-57-rhel7
827264d42df6: Mounted from rhscl/mysql-57-rhel7
latest: digest:
sha256:170e2546270690fded13f3ced0d575a90cef58028abcef8d37bd62a166ba436b size:
1156
```

**NOTE**

The output for the layers might differ.

- 2. Build the parent image for the To Do List application using the Node.js Dockerfile.

To build the base image, run the **build.sh** script located at **~/D0180/labs/openshift-template/images/nodejs**.

```
[student@workstation mysql]$ cd ~/D0180/labs/openshift-template/images/nodejs
[student@workstation nodejs]$ ./build.sh
```

- 3. Build the To Do List application child image using the Node.js Dockerfile.

3.1. Go to the **~/D0180/labs/openshift-template/deploy/nodejs** directory and run the **build.sh** command to build the child image.

```
[student@workstation nodejs]$ cd ~/D0180/labs/openshift-template/deploy/nodejs
[student@workstation nodejs]$ ./build.sh
```

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]$ docker tag do180/todonodejs \
registry.lab.example.com/do180/todonodejs
[student@workstation nodejs]$ docker push \
registry.lab.example.com/do180/todonodejs
The push refers to a repository [registry.lab.example.com/do180/todonodejs]
c024668a8c8e: Pushed
b26e8fcb95bc: Pushed
dbc12c3540e: Pushed
140887c9341b: Pushed
3935c38159a7: Pushed
44b08d0727ff: Pushed
86888f0aea6d: Layer already exists
dda6e8dfdcf7: Layer already exists
latest: digest:
sha256:c2c0639d09b9e12d3236da9905bb57978f7548b7d1cc60025c76bb8b82fddd47 size:
1993
```

► 4. Create the persistent volume.

- 4.1. Log in to OpenShift Container Platform as administrator to create a persistent volume.

Similar to the volumes used with plain Docker containers, OpenShift uses the concept of persistent volumes to provide persistent storage for pods.

```
[student@workstation nodejs]$ oc login -u admin -p redhat \
https://master.lab.example.com
```

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/openshift-template
[student@workstation openshift-template]$ ./create-pv.sh
```



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 openshift-template]$ oc new-project template
Now using project "template" on server "https://master.lab.example.com:443".
```

You can add applications to this project with the 'new-app' command. For example, try:

```
oc new-app centos/ruby-22-centos7~https://github.com/openshift/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 openshift-template]$ ./setpolicy.sh
```

5.3. Review the template.

Using your preferred editor, open and examine the template located at **/home/student/D0180/labs/openshift-template/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 openshift-template]$ 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 openshift-template]$ oc get pods -w
NAME      READY     STATUS            RESTARTS   AGE
mysql    0/1      ContainerCreating  0          9s
todoapi  1/1      Running           0          9s
NAME      READY     STATUS            RESTARTS   AGE
mysql    1/1      Running           0          2m
^C
```

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 openshift-template]$ oc expose service todoapi
route "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 openshift-template]$ oc status
In project template on server https://master.lab.example.com:443

svc/mysql - 172.30.105.104:3306
pod/mysql runs registry.lab.example.com/do180/mysql-57-rhel7

http://todoapi-template.apps.lab.example.com to pod port 30080 (svc/todoapi)
pod/todoapi runs registry.lab.example.com/do180/todonodejs

2 infos identified, use 'oc status -v' to see details.
```

- 7.2. Use **curl** to test the REST API for the To Do List application.

```
[student@workstation openshift-template]$ curl -w "\n" \
http://todoapi-template.apps.lab.example.com/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.

- 7.3. Open Firefox on workstation and point your browser to <http://todoapi-template.apps.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 miss to include that in the URL, you may encounter issues with the application.

- 7.4. Verify that the correct images were built, and that the application is running correctly:

```
[student@workstation openshift-template]$ lab openshift-template grade
```

► **8.** Clean up.

- 8.1. Delete the project used by this exercise by running the following commands in the terminal window.

```
[student@workstation openshift-template]$ oc delete project template
```

- 8.2. Delete the persistent volumes using the provided shell script by running the following command in your terminal window.

```
[student@workstation openshift-template]$ ./delete-pv.sh
persistentvolume "pv0001" deleted
persistentvolume "pv0002" deleted
```

- 8.3. Delete the container images generated during the Dockerfile builds:

```
[student@workstation openshift-template]$ docker rmi -f $(docker images -q)
```

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 for the application.

RESOURCES	
Files:	/home/student/D0180/labs/deploy-multicontainer
Application URL:	http://quote-php-deploy.apps.lab.example.com/

OUTCOMES

You should be able to create a OpenShift application comprised of multiple containers and access it through a web browser.

The workstation requires the PHP application source code and lab files. Both Docker and the OpenShift cluster need to be running. To download the lab files and verify the status of the Docker service and the OpenShift cluster, run the following command in a new terminal window.

```
[student@workstation ~]$ lab deploy-multicontainer setup
[student@workstation ~]$ cd ~/D0180/labs/deploy-multicontainer
```

1. Log in to OpenShift cluster as the **admin** user and create a new project for this exercise.
 - 1.1. From **workstation** VM, log in as the **admin** user.
 - 1.2. Create a new project in OpenShift, named **deploy**, for this lab.
 - 1.3. Relax the default cluster security policy.
To allow the container, run with the appropriate privileges, set the security policy using the provided **setpolicy.sh** shell script.
2. Build the Database container image and publish it to the private registry.
 - 2.1. Build the MySQL Database image using the provided Dockerfile and build script in the **images/mysql** directory.
 - 2.2. Push the MySQL Image to the Private Registry
In order to make the image available for OpenShift to use in the template, give it a tag of **registry.lab.example.com/do180/mysql-57-rhel7** and push it to the private registry.
3. Build the PHP container image and publish it to the private registry.
 - 3.1. Build the PHP image using the provided Dockerfile and build script in the **images/quote-php** directory.
 - 3.2. 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]$ docker tag do180/quote-php \
registry.lab.example.com/do180/quote-php
[student@workstation quote-php]$ docker push \
registry.lab.example.com/do180/quote-php
The push refers to a repository [registry.lab.example.com/do180/quote-php]
4a01dd82afa9: Pushed
40e133ec4a04: Pushed
b32755a27787: Pushed
4cd1411a1153: Pushed
... output omitted ...
latest: digest:
sha256:a60a9c7a9523b4e6674f2cdbfb437107782084828e4a18dc4dfa62ad4939fd6a size:
2194
```

4. Review the provided template file **/home/student/D0180/labs/deploy-multicontainer/quote-php-template.json**.
Note the definitions and configuration of the pods, services, and persistent volume claims defined in the template.
5. 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 **template** project using the **oc delete project** command to remove any existing resources and restart the exercise from this step.

6. Upload the PHP application template so that it can be used by any developer with access to your project.
7. Process the uploaded template and create the application resources.
 - 7.1. Use the **oc process** command to process the template file and use the **pipe** command to send the result to the **oc create** command to create an application from the template.
 - 7.2. Verify the deployment.
Verify the status of the deployment using the **oc get pods** command with the **-w** option to monitor the deployment status. Wait until both the pods are running. It may take some time for both pods to start up.
8. 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.

9. Test the application.
 - 9.1. Find the FQDN where the application is available using the **oc get route** command and note the FQDN for the app.
 - 9.2. Use **curl** command to test the REST API for the PHP Quote application.

**NOTE**

The text displayed in the output above may differ. But the **curl** command should run successfully.

- 9.3. Verify that the correct images were built and that the application is running without errors.

```
[student@workstation deploy-multicontainer]$ lab deploy-multicontainer grade
```

10. Clean up.

- 10.1. Delete the project used in this exercise.
- 10.2. Delete the persistent volumes using the provided shell script, available at **/home/student/D0180/labs/deploy-multicontainer/delete-pv.sh**.
- 10.3. Delete the container images generated during the Dockerfile builds.

**WARNING**

You may see an error deleting one of the images if there are multiple tags for a single image. This can be safely ignored.

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 for the application.

RESOURCES	
Files:	/home/student/D0180/labs/deploy-multicontainer
Application URL:	http://quote-php-deploy.apps.lab.example.com/

OUTCOMES

You should be able to create a OpenShift application comprised of multiple containers and access it through a web browser.

The workstation requires the PHP application source code and lab files. Both Docker and the OpenShift cluster need to be running. To download the lab files and verify the status of the Docker service and the OpenShift cluster, run the following command in a new terminal window.

```
[student@workstation ~]$ lab deploy-multicontainer setup
[student@workstation ~]$ cd ~/D0180/labs/deploy-multicontainer
```

1. Log in to OpenShift cluster as the **admin** user and create a new project for this exercise.
 - 1.1. From **workstation** VM, log in as the **admin** user.
Run the following command in your terminal window.

```
[student@workstation deploy-multicontainer]$ oc login -u admin \
-p redhat https://master.lab.example.com
Login successful.
...output omitted...
```

- If the **oc login** command prompts about using insecure connections, answer **y** (yes).
- 1.2. Create a new project in OpenShift, named **deploy**, for this lab.
Run the following command to create the project.

```
[student@workstation deploy-multicontainer]$ oc new-project deploy
Now using project "deploy" on server "https://master.lab.example.com:443".
```

You can add applications to this project with the 'new-app' command. For example, try:

```
oc new-app centos/ruby-22-centos7~https://github.com/openshift/ruby-ex.git
to build a new example application in Ruby.
```

1.3. Relax the default cluster security policy.

To allow the container, run with the appropriate privileges, set the security policy using the provided **setpolicy.sh** shell script.

Run the following command.

```
[student@workstation deploy-multicontainer]$ ./setpolicy.sh
```

2. Build the Database container image and publish it to the private registry.

2.1. Build the MySQL Database image using the provided Dockerfile and build script in the **images/mysql** directory.

To build the base MySQL image, run the **build.sh** script.

```
[student@workstation ~]$ cd ~/DO180/labs/deploy-multicontainer/images/mysql
[student@workstation mysql]$ ./build.sh
```

2.2. Push the MySQL Image to the Private Registry

In order to make the image available for OpenShift to use in the template, give it a tag of **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]$ docker tag do180/mysql-57-rhel7 \
registry.lab.example.com/do180/mysql-57-rhel7
[student@workstation mysql]$ docker push \
registry.lab.example.com/do180/mysql-57-rhel7
The push refers to a repository [registry.lab.example.com/do180/mysql-57-rhel7]
...output omitted...
latest: digest:
sha256:170e2546270690fded13f3ced0d575a90cef58028abcef8d37bd62a166ba436b size:
1156
```

3. Build the PHP container image and publish it to the private registry.

3.1. Build the PHP image using the provided Dockerfile and build script in the **images/quote-php** directory.

To build the base PHP image, run the **build.sh** script.

```
[student@workstation ~]$ cd ~/DO180/labs/deploy-multicontainer/images/quote-php
[student@workstation quote-php]$ ./build.sh
```

3.2. 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.

To tag and push the image run the following commands in the terminal window.

```
[student@workstation quote-php]$ docker tag do180/quote-php \
registry.lab.example.com/do180/quote-php
```

```
[student@workstation quote-php]$ docker push \
registry.lab.example.com/do180/quote-php
The push refers to a repository [registry.lab.example.com/do180/quote-php]
4a01dd82afa9: Pushed
40e133ec4a04: Pushed
b32755a27787: Pushed
4cd1411a1153: Pushed
... output omitted ...
latest: digest:
sha256:a60a9c7a9523b4e6674f2cdbfb437107782084828e4a18dc4dfa62ad4939fd6a size:
2194
```

4. Review the provided template file **/home/student/DO180/labs/deploy-multicontainer/quote-php-template.json**.

Note the definitions and configuration of the pods, services, and persistent volume claims defined in the template.

5. 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 quote-php]$ cd ~/DO180/labs/deploy-multicontainer
[student@workstation deploy-multicontainer]$ ./create-pv.sh
```



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** command to remove any existing resources and restart the exercise from this step.

6. Upload the PHP application template so that it can be used by any developer with access to your project.

Use the **oc create -f** command to upload the template file to the project.

```
[student@workstation deploy-multicontainer]$ oc create -f quote-php-template.json
template "quote-php-persistent" created
```

7. Process the uploaded template and create the application resources.

- 7.1. Use the **oc process** command to process the template file and use the **pipe** command to send the result to the **oc create** command to create an application from the template.

Run the following command in the terminal window.

```
[student@workstation deploy-multicontainer]$ 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
```

7.2. Verify the deployment.

Verify the status of the deployment using the **oc get pods** command with the **-w** option to monitor the deployment status. Wait until both the pods are running. It may take some time for both pods to start up.

Run the following command in the terminal window.

```
[student@workstation deploy-multicontainer]$ oc get pods -w
NAME      READY     STATUS            RESTARTS   AGE
mysql     0/1      ContainerCreating   0          8s
quote-php 1/1      Running           0          8s
NAME      READY     STATUS            RESTARTS   AGE
mysql     1/1      Running           0          2m
^C
```

Press **Ctrl+C** to exit the command.

8. 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 deploy-multicontainer]$ oc expose svc quote-php
route "quote-php" exposed
```

9. Test the application.

9.1. Find the FQDN where the application is available using the **oc get route** command and note the FQDN for the app.

Run the following command in the terminal window.

```
[student@workstation deploy-multicontainer]$ oc get route
NAME      HOST/PORT                               PATH      SERVICES      PORT
TERMINATION  WILDCARD
quote-php  quote-php-deploy.apps.lab.example.com    quote-php  8080
None
```

9.2. Use **curl** command to test the REST API for the PHP Quote application.

```
[student@workstation ~]$ curl http://quote-php-deploy.apps.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.

9.3. Verify that the correct images were built and that the application is running without errors.

```
[student@workstation deploy-multicontainer]$ lab deploy-multicontainer grade
```

10. Clean up.

10.1. Delete the project used in this exercise.

Run the following command in your terminal window.

```
[student@workstation deploy-multicontainer]$ oc delete project deploy
```

10.2. Delete the persistent volumes using the provided shell script, available at **/home/student/D0180/labs/deploy-multicontainer/delete-pv.sh**.

Run the following commands in your terminal window.

```
[student@workstation deploy-multicontainer]$ ./delete-pv.sh
persistentvolume "pv0001" deleted
persistentvolume "pv0002" deleted
```

10.3. Delete the container images generated during the Dockerfile builds.

```
[student@workstation deploy-multicontainer]$ docker rmi -f $(docker images -q)
```

**WARNING**

You may see an error deleting one of the images if there are multiple tags for a single image. This can be safely ignored.

This concludes the lab.

SUMMARY

In this chapter, you learned:

- Containerized applications cannot rely on fixed IP addresses or host names to find services. Docker and Kubernetes provide mechanisms that define environment variables with network connection parameters.
- The user-defined docker networks allow containers to contact other containers by names. To be able to communicate with each other, containers must be attached to the same user-defined docker network, using **--network** option with the **docker run** command.
- Kubernetes services define environment variables injected into all pods from the same project.
- Kubernetes templates automate creating applications consisting of multiple pods interconnected by services.
- Template parameters define environment variables for multiple pods.

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, students 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* process (S2I) is a simple way to create images based on programming languages in OpenShift. 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.

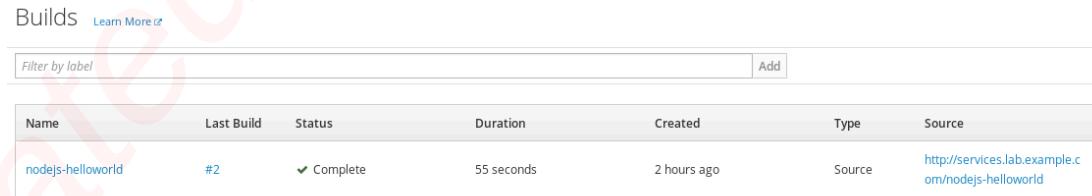
It is important to understand the basic workflow for most of the programming languages supported by OpenShift. The S2I image creation process is composed of two major steps:

- *Build step*: Responsible for compiling source code, downloading library dependencies, and packaging the application as a Docker image. Furthermore, the build step pushes the image to the OpenShift registry for the deployment step. The build step is managed by the **BuildConfig (BC)** object from OpenShift.
- *Deployment step*: Responsible for starting a pod and making the application available for OpenShift. This step is executed after the build step, but only if the build step is executed without problems. The deployment step is managed by the **DeploymentConfig (DC)** object from OpenShift.

For the S2I process, each application uses its own **BuildConfig** and **DeploymentConfig** objects, the name of which matches the application name. The deployment process is aborted 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 is executed and a log is saved. Upon a successful execution, the application is started 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 of the project page, depicted as follows.



Name	Last Build	Status	Duration	Created	Type	Source
nodejs-helloworld	#2	✓ Complete	55 seconds	2 hours ago	Source	http://services.lab.example.com/nodejs-helloworld

Figure 8.1: Detailed vision of a project

For each build attempt, a history of the build, tagged with a number, is provided for evaluation. Clicking on the number leads to the details page of the build.

php-hello-1 created 33 minutes ago

Rebuild Actions ▾

app php-hello buildconfig php-hello openshift.io/build-config.name php-hello More labels...

Details Environment Logs Events

Status

Status: ✓ Complete
Started: 33 minutes ago – Jul 5, 2018 8:19:08 AM
Duration: 41 seconds
Triggered By: Build configuration change

Figure 8.2: Detailed vision of a build configuration

Use the Applications link from the left panel to identify issues during the deployment step.

On each pod deployed, the logs can be obtained by accessing the Applications → Deployments from the left panel.

Deployments » php-hello » #1

php-hello-1 created 33 minutes ago

Actions ▾

app php-hello openshift.io/deployment-config.name php-hello

Details Environment Logs Events

Status: Active
Deployment Config: php-hello
Status Reason: config change
Selectors: deployment=php-hello-1
Replicas: 1 current / 1 desired

Figure 8.3: Detailed vision of a deployment configuration

The **oc** command-line interface has several verbs 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, the build configuration must be restarted. Run the following command to request a new build. By issuing the command, a new pod with the build process is automatically spawned.

```
$ oc start-build <application-name>
```

After a successful build, a pod is started, in which the application and the deployment process is executed.

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, which are usually provided as 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 process 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 labeled 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, developer run into permission issues, such as an access being 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. Moreover, 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. 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.

Troubleshooting Invalid Parameters

Multi-container applications may share parameters, such as login credentials. Ensure that the same values for parameters are passed to 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.

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 the persistent volume, in the mentioned order. Then recreate the persistent volume.

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 **docker 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 *Guidelines* section of the OpenShift Container Platform documentation accessible at:

Creating Images

https://docs.openshift.com/container-platform/3.9/creating_images/

► GUIDED EXERCISE

TROUBLESHOOTING AN OPENSHIFT BUILD

In this exercise, you will troubleshoot an OpenShift build and deployment process.

RESOURCES

Files:	NA
Application URL:	http://nodejs-helloworld-nodejs.apps.lab.example.com

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://master.lab.example.com`.

Retrieve the lab files and verify that Docker and the OpenShift cluster are running by running the following command.

```
[student@workstation ~]$ lab bc-and-dc setup
```

- 1. Log in to OpenShift with the **developer** user with a password of **redhat**. Create a new project named **nodejs**.

If the **oc login** command prompts about using insecure connections, answer **y** (yes).

```
[student@workstation ~]$ oc login -u developer https://master.lab.example.com
Authentication required for https://master.lab.example.com:443 (openshift)
Username: developer
Password: redhat
Login successful.
```

You don't have any projects. You can try to create a new project, by running

```
oc new-project <projectname>
```

```
[student@workstation ~]$ oc new-project nodejs
Now using project "nodejs" on server "https://master.lab.example.com".
... 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 **~/DO180/labs/bc-and-dc/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
```

The **--build-env** option defines an environment variable to the builder pod. The **npm** command, which is run as part of the build, fetches NPM modules from the Nexus server available on the **services** VM.



IMPORTANT

In the previous command, there should be no spaces between **registry=** and the URL of the Nexus server. There should also 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 will transition from a status of **running** to **Error**.

```
[student@workstation ~]$ oc get pods -w
NAME           READY   STATUS    RESTARTS   AGE
nodejs-helloworld-1-build   0/1     Error      0          35s
^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** verb in a terminal window to review the output of the build process.

```
[student@workstation ~]$ oc logs -f bc/nodejs-helloworld
```

The following error is raised as part of the build log.

```
[student@workstation nodejs-helloworld]$ oc logs -f bc/nodejs-helloworld
Cloning "http://services.lab.example.com/nodejs-helloworld" ...
Commit: 03956fd606dc034a85462fc3d390582737e1dcc47 (Establish remote repository)
Author: root <root@services.lab.example.com>
Date: Wed Jun 13 00:38:02 2018 +0000
--> 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
error: build error: non-zero (13) exit code
  from registry.lab.example.com/rhscl/nodejs-4-
rhe17@sha256:6bdbb943fd3401752cc11b07e574c418558205ffa9105435238f74dba1c1f328

```

The output indicates 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/bc-and-dc** directory.

```
[student@workstation ~]$ cd D0180/labs/bc-and-dc
[student@workstation bc-and-dc]$ git clone \
http://services.lab.example.com/nodejs-helloworld
```

The source code from the application is cloned locally in the **~/D0180/labs/bc-and-dc/nodejs-helloworld** directory.

3.2. Evaluate the **package.json** file.

Use your preferred editor to open the **~/D0180/labs/bc-and-dc/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. Notice the **x** in lower case.

```
"express": "4.14.x"
```

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 bc-and-dc]$ cd nodejs-helloworld
[student@workstation nodejs-helloworld]$ git commit -am "Fixed Express release"
[master 53d87dc] Updated Changes
 1 file changed, 1 insertion(+), 1 deletion(-)
[student@workstation nodejs-helloworld]$ git push
...
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 318 bytes | 0 bytes/s, done.
```

```
Total 3 (delta 1), reused 0 (delta 0)
To https://registry.lab.example.com/nodejs-helloworld
  87edb8c..53d87dc 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 ...
Pushing image docker-registry.default.svc:5000/nodejs/nodejs-helloworld:latest ...
Pushed 0/6 layers, 5% complete
Pushed 1/6 layers, 31% complete
Pushed 2/6 layers, 44% complete
Pushed 3/6 layers, 62% complete
Pushed 4/6 layers, 77% complete
Pushed 5/6 layers, 98% complete
Pushed 6/6 layers, 100% complete
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
nodejs-helloworld-1-build	0/1	Error	0	29m
nodejs-helloworld-1-rpx1d	0/1	CrashLoopBackOff	6	6m
nodejs-helloworld-2-build	0/1	Completed	0	7m

- 4.3. Review the logs generated by the **nodejs-helloworld-1-rpx1d** pod.

```
[student@workstation nodejs-helloworld]$ oc logs -f nodejs-helloworld-1-rpx1d
```

Use the same value from the output of the previous step. The output should read.

```
Environment:
...
npm info using node@v4.6.2
npm ERR! Linux 3.10.0-514.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
...
```

The application fails to start up because a script is missing.

► 5. Fix the problem by updating the application pod.

5.1. Update the **package.json** file to define a start up command.

The previous output indicates that the **~/D0180/labs/bc-and-dc/nodejs-helloworld/package.json** file is missing the **start** attribute in the **scripts** field. The **start** attribute defines a command to run when the container starts. It invokes the **node** binary which will run the **app.js** application.

```
node app.js
```

To fix the problem, add to the **package.json** file the following attribute. Do not forget the comma after the bracket.

```
...
  "main": "app.js",
  "scripts": {
    "start": "node app.js"
  },
...
```

5.2. Commit and push the changes made to the project.

```
[student@workstation nodejs-helloworld]$ git commit -am "Added start up script"
[master 20b2fe8] Updated packages
 1 file changed, 3 insertions(+)
[student@workstation nodejs-helloworld]$ git push
warning: push.default is unset; its implicit value is changing in
Git 2.0 from 'matching' to 'simple'. To squelch this message
...
To https://registry.lab.example.com/nodejs-helloworld
 cbeb4f1..20b2fe8 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
```

According to the following output, the third build is successful, and the application is able to start with no errors.

NAME	READY	STATUS	RESTARTS	AGE
DO180-OCP-3.9-en-1-20180926				

```

nodejs-helloworld-1-build  0/1      Error      0      20m
nodejs-helloworld-2-build  0/1      Completed   0      18m
nodejs-helloworld-2-g78z8  1/1      Running    0      20s
nodejs-helloworld-3-build  0/1      Completed   0      33s

```

5.5. Review the logs generated by the **nodejs-helloworld-2-g78z8** pod.

```
[student@workstation nodejs-helloworld]$ oc logs -f nodejs-helloworld-2-g78z8
```

Use the same value from the output of the previous step.

```

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** verb to expose the application.

```
[student@workstation nodejs-helloworld]$ oc expose svc/nodejs-helloworld
route "nodejs-helloworld" exposed
```

6.2. Retrieve the address associated with the application.

```

[student@workstation nodejs-helloworld]$ oc get route
NAME          HOST/PORT           PATH
SERVICES
nodejs-helloworld  nodejs-helloworld-nodejs.apps.lab.example.com
nodejs-helloworld

```

6.3. Access the application from the **workstation** VM by using the **curl** command.

```
[student@workstation nodejs-helloworld]$ curl \
http://nodejs-helloworld-nodejs.apps.lab.example.com
```

The output should read as follows.

```
Hello world!
```

Evaluation

Verify that the application is correctly set up by running the following command from a terminal window.

```
[student@workstation ~]$ lab bc-and-dc grade
```

Cleanup

Delete the project, which also deletes all the resources in the project.

```
[student@workstation ~]$ oc delete project nodejs
```

This concludes the guided exercise.

TROUBLESHOOTING CONTAINERIZED APPLICATIONS

OBJECTIVES

After completing this section, students 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 Docker and OpenShift cluster events.

FORWARDING PORTS FOR TROUBLESHOOTING

Sometimes developers and system administrators need network access to a container that would not be needed by application users. For example, they may need to use the administration console for a database or messaging service.

Docker provides port forwarding features by using the **-p** option along the **run** verb. In this case, there is no distinction between network access for regular application access and for troubleshooting. As a refresher, here is an example of configuring port forwarding by mapping the port from the host to a database server running inside a container:

```
$ docker 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 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, make sure to leave the terminal window running. Closing the window or canceling the process stops the port mapping.

While the **docker run -p** port-forwarding mapping can only be configured when the container is started, the mapping via the **oc port-forward** command can be created and destroyed at any time after a pod was created.



NOTE

Creating a **NodePort** service type for a database pod would be similar to running **docker run -p**. However, Red Hat discourages the usage of the **NodePort** approach to avoid exposing the service to direct connections. Mapping via port-forwarding in OpenShift is considered a more secure alternative.

USING oc port-forward FOR DEBUGGING APPLICATIONS

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 need to 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

Docker and OpenShift provide the ability to view logs in running containers and pods to facilitate the troubleshooting process. 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 Docker starts a container either directly or on the OCP 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 **docker** and **oc** commands, even after the container was stopped, but not removed.

To retrieve the output of a running container, use the following **docker** command.

```
$ docker 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** will just default to the only running container and return the output.

DOCKER AND OPENSHIFT EVENTS

Some developers consider Docker and OpenShift logs to be too low-level, making the troubleshooting difficult. Fortunately, both of these two technologies provide a high-level logging and auditing facility called **events**.

Docker and OpenShift events signal significant actions like starting a container or destroying a pod.

Docker events

To show Docker events, use the **events** verb, as follows.

```
$ docker events --since=10m
```

The **--since** command option allows specifying a time stamp as an absolute string or as a time interval. The previous example shows events generated during the last 10 minutes.

OpenShift events

To read OpenShift events, use the **get** verb with the **ev** resource type for the **oc** command, as follows.

```
$ oc get ev
```

Events listed by the **oc** command this way are not filtered and span the whole OCP cluster. Using a pipe to standard UNIX filters such as **grep** can help, but OpenShift and Docker offer an alternative in order to consult cluster events. The approach is provided by the **describe** verb.

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:           ...output omitted...
```

ACCESSING RUNNING CONTAINERS

The **docker logs** and **oc logs** commands can be useful for viewing the 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, Docker and OpenShift provide the **exec** verb, which allows the creation of new processes inside a running container, and have the standard output and input of these processes redirected to the user terminal. The following screen display the usage of the **docker exec** command.

```
$ docker 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 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.

```
$ docker 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 **docker 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.

```
$ docker 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 **rhel7.5** image used throughout this course. The **RUN** instruction installs the tools that are commonly used for network troubleshooting.

```
FROM rhel7.5

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, minus 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.

docker cp

The **cp** verb 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.

```
$ docker 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.

```
$ docker cp todoapi:/opt/jboss/standalone/conf/standalone.conf .
```

The **docker 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.

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.

```
$ docker run -v /conf:/etc/httpd/conf -d do180/apache
```

Piping docker exec

For containers that are already running, the **docker 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.

```
$ docker exec -i <containerName> mysql -uroot -proot < /path/on/host/to/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.

```
$ docker 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, 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 **docker cp** for containers running under OpenShift pods.

DEMONSTRATION: FORWARDING PORTS

- From **workstation** system, log in to the OpenShift cluster as **developer** user with the password **redhat**. Create the **port-forward** project.

```
[student@workstation ~]$ oc login -u developer -p redhat \
https://master.lab.example.com
Login successful.
...output omitted...
[student@workstation ~]$ oc new-project port-forward
```

If the **oc login** command prompts about using insecure connections, answer **y** (yes).

- Create a new application from the **mysql:5.6** container image using the **oc new-app** command.

This image requires several environment variables:

- **MYSQL_USER**
- **MYSQL_PASSWORD**
- **MYSQL_DATABASE**
- **MYSQL_ROOT_PASSWORD**

Use the **-e** option to pass these values as environment variables.

```
[student@workstation ~]$ oc new-app \
mysql:5.6 \
--name=port-forwarding \
-e MYSQL_USER=user1 -e MYSQL_PASSWORD=mypa55 \
```

```
-e MYSQL_ROOT_PASSWORD=r00tpa55 -e MYSQL_DATABASE=testdb
```

- Run the **oc status** command to view the status of the new application, and to ensure that the deployment of the MySQL image is successful.

```
[student@workstation ~]$ oc status
In project port-forward on server https://master.lab.example.com:443

svc/port-forwarding - 172.30.90.144:3306
dc/port-forwarding deploys istag/port-forwarding:latest
  deployment #1 deployed 23 seconds ago - 1 pod
...

```

Wait for the MySQL server pod to be ready and running.

```
[student@workstation ~]$ oc get pods
NAME           READY   STATUS    RESTARTS   AGE
port-forwarding-1-t3qfb   1/1     Running   0          14s
```

- To allow clients from the developer's machine (in this environment, **workstation**) to access the database server from outside the OpenShift cluster, forward a local port to the pod by using the **oc port-forward** command.

```
[student@workstation ~]$ oc port-forward \
port-forwarding-1-t3qfb 13306:3306
Forwarding from 127.0.0.1:13306 -> 3306
```



IMPORTANT

The **oc port-forward** command does not return the shell prompt. Leave this terminal open, and then open another terminal for the next step.

- From the second terminal of **workstation** machine, run the following command to connect to the database using the local port:

```
[student@workstation ~]$ mysql -h127.0.0.1 -P13306 -uuser1 -pmypa55
Welcome to the MariaDB monitor. Commands end with ; or \g.
Your MySQL connection id is 1
Server version: 5.6.39 MySQL Community Server (GPL)

Copyright (c) 2000, 2017, Oracle, MariaDB Corporation Ab and others.

... output omitted ...

MySQL [(none)]>
```

- Verify that the **testdb** database is present.

```
MySQL [(none)]> show databases;
+-----+
| Database      |
+-----+
| information_schema |
```

```
| testdb           |
+-----+
2 rows in set (0.00 sec)
```

7. Exit from the MySQL prompt:

```
MySQL [(none)]> exit
Bye
```

8. Press **Ctrl+C** from the terminal window that runs the **oc port-forward** command. This stops the port-forwarding and prevents further access to the MySQL database from the developer's machine.
9. Delete the **port-forward** project.

```
[student@workstation ~]$ oc delete project port-forward
```

This concludes the demonstration.



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/3.9/html/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/3.9/html/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 check **docker logs** and events.

RESOURCES	
Files:	/home/student/D0180/labs/debug- httpd
Resources	HTTPD 2.4 image (httpd)

OUTCOMES

You should be able to configure an Apache HTTPD container to send debug logs to **stdout** and view them using the **docker 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 debug-httpd setup
```

- 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 the standard output, **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/debug-**httpd**/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/debug-httdp
[student@workstation D0180/labs/debug-httdp]$ ./build.sh
```

- 2.2. Verify that the image is created.

From the terminal window, run the following command.

```
[student@workstation D0180/labs/debug-httdp]$ docker images
```

The new image must be available in the local docker cache.

REPOSITORY	TAG	IMAGE ID	...
debug-httdp	latest	c86936c8e791	...

- ▶ 3. Create a new HTTPd container from the custom image.

```
[student@workstation ~]$ docker run \
--name debug-httdp -d \
-p 10080:80 debug-httdp
```

- ▶ 4. Review the container's log messages and events.

- 4.1. View the debug log messages from the container using the **docker logs** command.

```
[student@workstation ~]$ docker logs -f debug-httdp
[Wed Apr 12 07:30:18.542794 2017] [mpm_event:notice] [pid 1:tid 140153712912256]
AH00489: Apache/2.4.25 (Unix) configured -- resuming normal operations
[Wed Apr 12 07:30:18.543388 2017] [mpm_event:info] [pid 1:tid 140153712912256]
AH00490: Server built: Mar 21 2017 20:50:17
[Wed Apr 12 07:30:18.543396 2017] [core:notice] [pid 1:tid 140153712912256]
AH00094: Command line: 'httpd -D FOREGROUND'
[Wed Apr 12 07:30:18.543398 2017] [core:debug] [pid 1:tid 140153712912256]
log.c(1546): AH02639: Using SO_REUSEPORT: yes (1)
[Wed Apr 12 07:30:18.543253 2017] [mpm_event:debug] [pid 6:tid 140153633576704]
event.c(2132): AH02471: start_threads: Using epoll
[Wed Apr 12 07:30:18.544103 2017] [mpm_event:debug] [pid 7:tid 140153633576704]
event.c(2132): AH02471: start_threads: Using epoll
```

```
[Wed Apr 12 07:30:18.545100 2017] [mpm_event:debug] [pid 8:tid 140153633576704]
event.c(2132): 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. Notice the new entries from the terminal that runs the **docker logs** command, as new logs are created.

```
[student@workstation ~]$ docker logs debug-httdp
[Wed Apr 12 07:03:43.435321 2017] [authz_core:debug] [pid 8:tid 140644341249792]
mod_authz_core.c(809): [client 172.17.0.1:43808] AH01626: authorization result of
Require all granted: granted
[Wed Apr 12 07:03:43.435337 2017] [authz_core:debug] [pid 8:tid 140644341249792]
mod_authz_core.c(809): [client 172.17.0.1:43808] AH01626: authorization result of
<RequireAny>: granted
172.17.0.1 - - [12/Apr/2017:07:03:43 +0000] "GET / HTTP/1.1" 200 45
```

- 4.4. Stop the **docker** command with **Ctrl+C**.

Verify the latest events from the docker daemon, related to previous steps using the **docker events** command.

Use the **--since** to adjust the time interval. Give it a value of **10m**, which approximates the elapsed time since you started this exercise.

```
[student@workstation ~]$ docker events --since=10m
2017-04-28T07:40:52.057034869-04:00 image pull
infrastructure.lab.example.com:5000/httpd:2.4
(name=infrastructure.lab.example.com:5000/httpd)
2017-04-28T07:40:52.355087400-04:00 container create ... name=serene_gates)
2017-04-28T07:40:53.166857912-04:00 container commit ... name=serene_gates)
2017-04-28T07:40:53.475573799-04:00 container destroy ... name=serene_gates)
2017-04-28T07:40:53.480235940-04:00 image tag ... (name=debug-httdp:latest)
2017-04-28T07:41:51.835352650-04:00 container create ... image=debug-httdp,
name=debug-httdp)
2017-04-28T07:41:51.962549582-04:00 network connect ... name=bridge, type=bridge)
2017-04-28T07:41:52.221989909-04:00 container start ... image=debug-httdp,
name=debug-httdp)
```



NOTE

Your output may be different compared to the one above. The **docker events** command will not return to the prompt unless you kill the command with **Ctrl+C**.

Evaluation

Verify that the debug configuration for the container is properly set up. Run the following from a terminal window.

```
[student@workstation ~]$ lab debug-httdp grade
```

Cleanup

Delete the container and image used in this exercise.

```
[student@workstation ~]$ docker stop debug-httdp  
[student@workstation ~]$ docker rm debug-httdp  
[student@workstation ~]$ docker rmi -f httpd:2.4 debug-httdp
```

This concludes the guided exercise.

▶ LAB

TROUBLESHOOTING CONTAINERIZED APPLICATIONS

PERFORMANCE CHECKLIST

In this lab, you will configure the **broken-`httpd`** container to send the logs to the **`stdout`**. The container runs an application that has a broken link, which originally downloads a file.

RESOURCES	
Files	<code>/home/student/D0180/labs/troubleshooting-lab</code>
Application URL	<code>http://localhost:3000</code>
Resources	<ul style="list-style-type: none"> Broken HTTP container image (do180/broken-<code>httpd</code>) Updated HTTP container image (do180/updated-<code>httpd</code>)

OUTCOMES

You should be able to send the Apache server logs to the standard output and fix a containerized application from the container that is not working as planned.

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 troubleshooting-lab setup
```

The previous command downloads the following files:

- **Dockerfile**: file that is used for building the container image.
 - **training.repo**: file that defines the Yum repositories of the classroom environment.
 - **httpd.conf**: file that configures the Apache HTTP server to send the logs to the standard output, **`stdout`**.
 - **src/index.html**: application file.
1. A custom **httpd** configuration file is provided to send the logs to **`stdout`**. Update the **Dockerfile** file by replacing the default configuration file with the custom HTTPD configuration file. Add the directive below the **EXPOSE** entry. The files are available in the `/home/student/D0180/labs/troubleshooting-lab` directory.
Build the new container image, and give it a tag of **do180/broken-`httpd`**.
 2. Create a container from the **do180/broken-`httpd`** image, which has a web page that contains a link. The link should download a file from `http://materials.example.com/labs/archive.tgz`.

Use the following specifications for the container.

- Name: **broken-`httpd`**
- Run as daemon: yes
- Volume: from **/usr/bin** host folder to **/usr/bin** container folder.
- Container image: **do180/broken-`httpd`**
- Port forward: from 3000 host port to 80 container port.



NOTE

To assist in the troubleshooting, the volume mount is responsible for sharing commands, such as **ping** from the **/usr/bin** host folder.

3. Open a web browser and download the file to see that the previous URL is correct. Troubleshoot the application and rebuild the image with the right settings.
The source code for the web page opened is available in the **/var/www/html** folder.
4. While troubleshooting, make the necessary changes to the build files and rebuild the image. Give it a tag of **do180/updated-`httpd`**
5. Terminate the running container and delete it. Create a new container by using the updated image and retry to use Download the file link from the web browser.

Use the following specifications for the new container.

- Name: **updated-`httpd`**
- Run as daemon: yes
- Volume: from **/usr/bin** host folder to **/usr/bin** container folder.
- Container image: **do180/updated-`httpd`**
- Port forward: from 3000 host port to 80 container port.

Evaluation

Grade your work by running the following command from a terminal window.

```
[student@workstation troubleshooting-lab]$ lab troubleshooting-lab grade
```

Cleanup

Delete all containers and images created by this lab and remove the two HTTP images.

```
[student@workstation troubleshooting-lab]$ cd ~
[student@workstation ~]$ docker stop updated-httpd
[student@workstation ~]$ docker rm updated-httpd
[student@workstation ~]$ docker rmi do180/broken-httpd do180/updated-httpd
```

This concludes the lab.

► SOLUTION

TROUBLESHOOTING CONTAINERIZED APPLICATIONS

PERFORMANCE CHECKLIST

In this lab, you will configure the **broken-`httpd`** container to send the logs to the **`stdout`**. The container runs an application that has a broken link, which originally downloads a file.

RESOURCES	
Files	<code>/home/student/D0180/labs/troubleshooting-lab</code>
Application URL	<code>http://localhost:3000</code>
Resources	<ul style="list-style-type: none"> Broken HTTP container image (do180/broken-<code>httpd</code>) Updated HTTP container image (do180/updated-<code>httpd</code>)

OUTCOMES

You should be able to send the Apache server logs to the standard output and fix a containerized application from the container that is not working as planned.

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 troubleshooting-lab setup
```

The previous command downloads the following files:

- **Dockerfile**: file that is used for building the container image.
 - **training.repo**: file that defines the Yum repositories of the classroom environment.
 - **httpd.conf**: file that configures the Apache HTTP server to send the logs to the standard output, **`stdout`**.
 - **src/index.html**: application file.
1. A custom **httpd** configuration file is provided to send the logs to **`stdout`**. Update the **Dockerfile** file by replacing the default configuration file with the custom HTTPD configuration file. Add the directive below the **EXPOSE** entry. The files are available in the `/home/student/D0180/labs/troubleshooting-lab` directory.
Build the new container image, and give it a tag of **do180/broken-`httpd`**.
 - 1.1. From the **workstation** machine, as the **student** user, go to the working directory located at `~student/D0180/labs/troubleshooting-lab`. Update the **Dockerfile** by adding a directive for installing the custom Apache configuration file.

```
[student@workstation ~]$ cd ~student/D0180/labs/troubleshooting-lab
```

Update the **Dockerfile**, which should contain an extra **COPY** directive as follows.

```
FROM rhel7:7.5
... output omitted ...

EXPOSE 80

COPY httpd.conf /etc/httpd/conf/
COPY src/index.html /var/www/html/
CMD ["httpd", "-D", "FOREGROUND"]
```

- 1.2. Save and exit your editor.
- 1.3. Run the **docker build** command to build the image

```
[student@workstation troubleshooting-lab]$ docker build -t do180/broken-httd .
```

2. Create a container from the **do180/broken-httd** image, which has a web page that contains a link. The link should download a file from <http://materials.example.com/labs/archive.tgz>.

Use the following specifications for the container.

- Name: **broken-httd**
- Run as daemon: yes
- Volume: from **/usr/bin** host folder to **/usr/bin** container folder.
- Container image: **do180/broken-httd**
- Port forward: from 3000 host port to 80 container port.



NOTE

To assist in the troubleshooting, the volume mount is responsible for sharing commands, such as **ping** from the **/usr/bin** host folder.

- 2.1. Use the **docker run** command to create a new container according to the specifications.

```
[student@workstation troubleshooting-lab]$ docker run \
--name broken-httd -d \
-p 3000:80 -v /usr/bin:/usr/bin do180/broken-httd
```

- 2.2. Ensure that the logs of the **httpd** service running inside the container are forwarded to the standard output, **stdout**.

```
[student@workstation troubleshooting]$ docker logs broken-httd
... output omitted ...
[Wed Feb 10 11:59:25.648268 2016] [auth_digest:notice] [pid 1] AH01757: generating
secret for digest authentication ...
[Wed Feb 10 11:59:25.649942 2016] [lmbmethod_heartbeat:notice] [pid 1] AH02282: No
slotmem from mod_heartmonitor
[Wed Feb 10 11:59:25.652586 2016] [mpm_prefork:notice] [pid 1] AH00163:
Apache/2.4.6 (Red Hat Enterprise Linux) configured -- resuming normal operations
```

```
... output omitted ...
```

- 2.3. Open a web browser and navigate to the URL `http://localhost:3000.`

**IMPORTANT**

The final period is not part of the URL.

Click the Download the file link. You should see a **server not found** message.

3. Open a web browser and download the file to see that the previous URL is correct.
Troubleshoot the application and rebuild the image with the right settings.

The source code for the web page opened is available in the `/var/www/html` folder.

- 3.1. Use the `docker exec` command to access the container shell for troubleshooting.

```
[student@workstation troubleshooting-lab]$ docker exec \
-it broken-httd \
/bin/bash
```

- 3.2. Use the `ping` command to ensure that the host name from the HTML link is accessible.

Display the content of the `index.html` file.

```
[root@9ef58f71371c /]# cat /var/www/html/index.html
<html>
<body>
<h1>Download application</h1>
<br/>
<a href="http://materials.example.net/troubleshooting/archive.tgz">Download the
file</a>
</body>
</html>
```

Copy the host part of the URL and use the `ping` command reach it.

```
[root@9ef58f71371c /]# ping materials.example.net
ping: materials.example.net: Name or service not known
```

Notice that the hostname `materials.example.net` does not resolve to a reachable host.

- 3.3. Exit out of the container.

```
[root@9ef58f71371c /]# exit
```

4. While troubleshooting, make the necessary changes to the build files and rebuild the image. Give it a tag of **do180/updated-httdp**
- 4.1. According to the previous steps, there is a misspelling in the domain name, example.net instead of example.com. From the **workstation** machine, locate and inspect the directive in the **Dockerfile** that installs the **index.html** file.
- Fix the URL in the **index.html** file located at **~/D0180/labs/troubleshooting-lab/src/index.html** by replacing the domain name.

The final contents of **index.html** file should read as follows:

```
<html>
<body>
<h1>Download application</h1>
<br/>
<a href="http://materials.example.com/labs/archive.tgz">Download the file</a>
</body>
</html>
```

- 4.2. Run the **docker build** command to build the image

```
[student@workstation troubleshooting-lab]$ docker build -t do180/updated-httdp .
... output omitted ...
--> Running in 76e5aa715524
--> 2b7b19a223e6
Removing intermediate container 76e5aa715524
Successfully built 2b7b19a223e6
```

5. Terminate the running container and delete it. Create a new container by using the updated image and retry to use Download the file link from the web browser.

Use the following specifications for the new container.

- Name: **updated-httdp**
- Run as daemon: yes
- Volume: from **/usr/bin** host folder to **/usr/bin** container folder.
- Container image: **do180/updated-httdp**
- Port forward: from 3000 host port to 80 container port.

- 5.1. Terminate and delete the **broken-httdp** container.

```
[student@workstation troubleshooting-lab]$ docker stop broken-httdp
[student@workstation troubleshooting-lab]$ docker rm broken-httdp
```

- 5.2. Use the **docker run** command to create the new container.

```
[student@workstation troubleshooting-lab]$ docker run \
--name updated-httdp -d \
-p 3000:80 -v /usr/bin:/usr/bin do180/updated-httdp
```

- 5.3. Switch back to the web browser and reattempt to access **http://localhost:3000** URL.

Click the Download the file link. The download should start.

Evaluation

Grade your work by running the following command from a terminal window.

```
[student@workstation troubleshooting-lab]$ lab troubleshooting-lab grade
```

Cleanup

Delete all containers and images created by this lab and remove the two HTTP images.

```
[student@workstation troubleshooting-lab]$ cd ~  
[student@workstation ~]$ docker stop updated-httdp  
[student@workstation ~]$ docker rm updated-httdp  
[student@workstation ~]$ docker rmi do180/broken-httdp do180/updated-httdp
```

This concludes the lab.

SUMMARY

In this chapter, you learned:

- S2I images require that security concerns must be addressed to minimize build and deployment issues.
- Development and sysadmin teams must work together to identify and mitigate problems with respect to S2I image creation process.
- Troubleshoot containers by using the **oc port-forward** command to debug applications as a last resource.
- OpenShift events provide low-level information about a container and its interactions. They can be used as a last resource to identify communication problems.

CHAPTER 9

COMPREHENSIVE REVIEW OF INTRODUCTION TO CONTAINERS, KUBERNETES, AND RED HAT OPENSHIFT

GOAL

Demonstrate how to containerize a software application, test it with Docker, and deploy it on an OpenShift cluster.

OBJECTIVE

Review concepts in the course to assist in completing the comprehensive review lab.

SECTIONS

- Comprehensive Review

LAB

- Containerizing and Deploying a Software Application

COMPREHENSIVE REVIEW

OBJECTIVES

After completing this section, students 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, Getting Started with Container Technology

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

- Describe the architecture of Linux containers.
- Describe how containers are implemented using Docker.
- Describe the architecture of a Kubernetes cluster running on the Red Hat OpenShift Container Platform.

Chapter 2, Creating Containerized Services

Provision a server using container technology.

- Describe three container development environment scenarios and build one using OpenShift.
- Create a database server from a container image stored on Docker Hub.

Chapter 3, Managing Containers

Manipulate pre-built container images to create and manage containerized services.

- Manage the life cycle of a container from creation to deletion.
- Save application data across container restarts through the use of persistent storage.
- Describe how Docker provides network access to containers, and access a container through port forwarding.

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.

- Install the OpenShift CLI tool and execute basic commands.
- Create standard Kubernetes resources.
- Build an application using the Source-to-Image facility of OpenShift.
- Create a route to a service.
- Create an application using the OpenShift web console.

Chapter 7, Deploying Multi-Container Applications

Deploy applications that are containerized using multiple container images.

- Describe the considerations for containerizing applications with multiple container images.
- Deploy a multi-container application with user-defined Docker network.
- 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 **docker** command allows you to build, run, and manage container images. Docker command documentation can be found by issuing the command **man docker**.
- The **oc** command allows you to create and manage OpenShift resources. OpenShift command-line documentation can be found by issuing either of the commands **man oc** or **oc help**. OpenShift commands that are particularly useful include:

oc login -u

Log in to OpenShift as the specified user. In this classroom, there are two user accounts defined: **admin** and **developer**.

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 -f

Convert a template into OpenShift resources that can be created 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. The label option, **-l label-value** is helpful with this command to delete multiple resources simultaneously.

- Before mounting any volumes on the Docker and OpenShift host, make sure to apply the correct SELinux context to the directory. The correct context is **svirt_sandbox_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 UFS 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 Docker, and deploy it on 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 Docker.
- Deploy the Nexus server container image to an OpenShift cluster.

Run the setup script for this comprehensive review.

```
[student@workstation ~]$ lab review setup
```

The lab files are located in the **/home/student/D0180/labs/review** directory. The solution files are located in the **/home/student/D0180/solutions/review** directory.

Instructions

Create a Docker container image that starts an instance of a Nexus server:

- The server should run as the **nexus** user and group. They have a UID and GID of **1001**, respectively.
- The server requires that the *java-1.8.0-openjdk-devel* package be installed. 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.
- The server is provided as a compressed tar file: **nexus-2.14.3-02-bundle.tar.gz**. The file can be retrieved from the server at http://content.example.com/ocp3.9/x86_64/installers/{tarball_name}. A script is provided to retrieve the tar bundle before you build the image: **get-nexus-bundle.sh**.
- Run the following script to start the nexus server: **nexus-start.sh**.
- The working directory and home for the nexus installation should be **/opt/nexus**. The version-specific nexus directory should be linked to a directory named **nexus2**.
- Nexus produces persistent data at **/opt/nexus/sonatype-work**. Make sure this can be mounted as a volume. You may want to initially build and test your container image without a persistent volume and add this in a second pass.
- There are two snippet files in the **image** lab directory that provide the commands needed to create the nexus account and install Java. Use these snippets to assist you in writing the **Dockerfile**.

Build and test the container image using Docker with and without a volume mount. In the directory, **/home/student/D0180/labs/deploy/docker**, there is a shell script to assist you in running the container with a volume mount. Remember to inspect the running container to determine its IP address. Use **curl** as well as the container logs to determine if the Nexus server is running properly.

```
[student@workstation review]$ docker logs -f {container-id}
[student@workstation review]$ curl http://{ipaddress}:8081/nexus/
```

Deploy the Nexus server container image into the OpenShift cluster:

- Publish the container image to the classroom private registry at **registry.lab.example.com**.
- The **/home/student/D0180/labs/review/deploy/openshift** directory contains several shell scripts, Kubernetes resource definitions, and an OpenShift template to help complete the lab.

- **create-pv.sh**

This script creates the Kubernetes persistent volume that stores the Nexus server persistent data.

- **delete-pv.sh**

This script deletes the Kubernetes persistent volume.

- **resources/pv.yaml**

This is a Kubernetes resource file that defines the persistent volume. This file is used by the **create-pv.sh** script.

- **resources/nexus-template.json**

This is an OpenShift template to deploy the Nexus server container image.

- Several helpful scripts are located in the **/home/student/D0180/solutions/review** directory for this lab that can help you deploy and undeploy the application if you are unsure how to proceed.
- Remember to push the container image to the classroom private registry at **registry.lab.example.com**. The container must be named **nexus** and have a tag **latest**. The OpenShift template expects this name.
- Use **review** for the OpenShift project name. Execute the **setpolicy.sh** shell script, located at **/home/student/D0180/labs/review/deploy/openshift**, after creating the project. The script sets the right privileges for the container.
- Make sure to create the persistent volume with the provided shell script before deploying the container with the template.
- Expose the Nexus server service as a route using the default route name. Test the server using a browser.
- Run the grading script to evaluate your work:

```
[student@workstation review]$ lab review grade
```

- Clean up your project by deleting the OpenShift project created in this lab and removing the local copy of the Docker container image as well.

Evaluation

After deploying the Nexus server container image into the OpenShift cluster, verify your work by running the lab grading script:

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

Cleanup

Delete the **review** project. Also delete the persistent volume using **delete-pv.sh** script.

```
[student@workstation review]$ oc delete project review  
[student@workstation review]$ ./delete-pv.sh
```

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 Docker, and deploy it on 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 Docker.
- Deploy the Nexus server container image to an OpenShift cluster.

Run the setup script for this comprehensive review.

```
[student@workstation ~]$ lab review setup
```

The lab files are located in the **/home/student/D0180/labs/review** directory. The solution files are located in the **/home/student/D0180/solutions/review** directory.

Instructions

Create a Docker container image that starts an instance of a Nexus server:

- The server should run as the **nexus** user and group. They have a UID and GID of **1001**, respectively.
- The server requires that the *java-1.8.0-openjdk-devel* package be installed. 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.
- The server is provided as a compressed tar file: **nexus-2.14.3-02-bundle.tar.gz**. The file can be retrieved from the server at http://content.example.com/ocp3.9/x86_64/installers/{tarball_name}. A script is provided to retrieve the tar bundle before you build the image: **get-nexus-bundle.sh**.
- Run the following script to start the nexus server: **nexus-start.sh**.
- The working directory and home for the nexus installation should be **/opt/nexus**. The version-specific nexus directory should be linked to a directory named **nexus2**.
- Nexus produces persistent data at **/opt/nexus/sonatype-work**. Make sure this can be mounted as a volume. You may want to initially build and test your container image without a persistent volume and add this in a second pass.
- There are two snippet files in the **image** lab directory that provide the commands needed to create the nexus account and install Java. Use these snippets to assist you in writing the **Dockerfile**.

Build and test the container image using Docker with and without a volume mount. In the directory, **/home/student/D0180/labs/deploy/docker**, there is a shell script to assist you in running the container with a volume mount. Remember to inspect the running container to determine its IP address. Use **curl** as well as the container logs to determine if the Nexus server is running properly.

```
[student@workstation review]$ docker logs -f {container-id}
[student@workstation review]$ curl http://{ipaddress}:8081/nexus/
```

Deploy the Nexus server container image into the OpenShift cluster:

- Publish the container image to the classroom private registry at **registry.lab.example.com**.
- The **/home/student/D0180/labs/review/deploy/openshift** directory contains several shell scripts, Kubernetes resource definitions, and an OpenShift template to help complete the lab.

- **create-pv.sh**

This script creates the Kubernetes persistent volume that stores the Nexus server persistent data.

- **delete-pv.sh**

This script deletes the Kubernetes persistent volume.

- **resources/pv.yaml**

This is a Kubernetes resource file that defines the persistent volume. This file is used by the **create-pv.sh** script.

- **resources/nexus-template.json**

This is an OpenShift template to deploy the Nexus server container image.

- Several helpful scripts are located in the **/home/student/D0180/solutions/review** directory for this lab that can help you deploy and undeploy the application if you are unsure how to proceed.
- Remember to push the container image to the classroom private registry at **registry.lab.example.com**. The container must be named **nexus** and have a tag **latest**. The OpenShift template expects this name.
- Use **review** for the OpenShift project name. Execute the **setpolicy.sh** shell script, located at **/home/student/D0180/labs/review/deploy/openshift**, after creating the project. The script sets the right privileges for the container.
- Make sure to create the persistent volume with the provided shell script before deploying the container with the template.
- Expose the Nexus server service as a route using the default route name. Test the server using a browser.
- Run the grading script to evaluate your work:

```
[student@workstation review]$ lab review grade
```

- Clean up your project by deleting the OpenShift project created in this lab and removing the local copy of the Docker container image as well.

1. Write a **Dockerfile** that containerizes the Nexus server. Go to **/home/student/DO180/labs/review/image** directory and create the **Dockerfile**.

1.1. Specify the base image to use:

```
FROM rhel7:7.5
```

1.2. Enter arbitrary name and email as the maintainer:

```
FROM rhel7:7.5
MAINTAINER username <username@example.com>
```

1.3. Set the environment variables for **NEXUS_VERSION** and **NEXUS_HOME**:

```
FROM rhel7:7.3
MAINTAINER username <username@example.com>

ENV NEXUS_VERSION=2.14.3-02 \
    NEXUS_HOME=/opt/nexus
```

1.4. Add the **training.repo** 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
```

1.5. Create the server home directory and service account/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 && \
    chown -R nexus:nexus ${NEXUS_HOME} && \
    chmod -R 755 ${NEXUS_HOME}
```

1.6. Install the Nexus server software at **NEXUS_HOME** and add the start-up script. Note that the **ADD** directive will extract the Nexus files. Create the **nexus2** symbolic link pointing to the Nexus server directory.

```
...
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}
```

- 1.7. Make the container run as the *nexus* user and make the working directory **/opt/nexus**:

```
...
USER nexus
WORKDIR ${NEXUS_HOME}
```

- 1.8. Define a volume mount point to store the Nexus server persistent data:

```
...
VOLUME ["/opt/nexus/sonatype-work"]
```

- 1.9. Execute the Nexus server shell script. The completed **Dockerfile** should read as follows:

```
FROM rhel7:7.3
MAINTAINER New User <user1@myorg.com>

ENV NEXUS_VERSION=2.14.3-02 \
    NEXUS_HOME=/opt/nexus

ADD training.repo /etc/yum.repos.d/training.repo
RUN yum --noplugins update -y && \
    yum --noplugins install -y --setopt=tsflags=nodocs \
        java-1.8.0-openjdk-devel && \
    yum --noplugins 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 && \
    chown -R nexus:nexus ${NEXUS_HOME} && \
    chmod -R 755 ${NEXUS_HOME}

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"]
```

2. Retrieve the Nexus server files to the **image** directory by using the script **get-nexus-bundle.sh**. Upon retrieval, build the container image with tag **nexus**:

```
[student@workstation ~]$ cd /home/student/D0180/labs/review/image
[student@workstation image]$ ./get-nexus-bundle.sh
```

```
[student@workstation image]$ docker build -t nexus .
```

3. Test the image with Docker using the provided **run-persistent.sh** shell script located under **/home/student/D0180/labs/review/deploy/docker** directory. Replace the container name appropriately as shown in the output of **docker ps** command.

```
[student@workstation images]$ cd /home/student/D0180/labs/review/deploy/docker
[student@workstation docker]$ ./run-persistent.sh
80970007036bbb313d8eeb7621fada0ed3f0b4115529dc50da4dccef0da34533
[student@workstation docker]$ docker ps
CONTAINER ID        IMAGE       COMMAND      CREATED          STATUS          NAMES
80970007036b        nexus      "sh nexus-start.sh"   5 seconds ago   Up 4 seconds   quizzical_fermat
[student@workstation docker]$ docker logs -f quizzical_fermat
...output omitted...
2018-07-05 12:09:31,851+0000 INFO  [jetty-main-1] *SYSTEM
org.sonatype.nexus.webresources.internal.WebResourceServlet - Max-age: 30 days
(2592000 seconds)
2018-07-05 12:09:31,873+0000 INFO  [jetty-main-1] *SYSTEM
org.sonatype.nexus.bootstrap.jetty.InstrumentedSelectChannelConnector - Metrics
enabled
2018-07-05 12:09:31,878+0000 INFO  [pxpool-1-thread-1] *SYSTEM
org.sonatype.nexus.configuration.application.DefaultNexusConfiguration - Applying
Nexus Configuration due to changes in [Scheduled Tasks] made by *TASK...
2018-07-05 12:09:31,887+0000 INFO  [jetty-main-1] *SYSTEM
org.eclipse.jetty.server.AbstractConnector - Started
InstrumentedSelectChannelConnector@0.0.0.0:8081
2018-07-05 12:09:31,887+0000 INFO  [jetty-main-1] *SYSTEM
org.sonatype.nexus.bootstrap.jetty.JettyServer - Running
2018-07-05 12:09:31,887+0000 INFO  [main] *SYSTEM
org.sonatype.nexus.bootstrap.jetty.JettyServer - Started
Ctrl+C
[student@workstation docker]$ docker inspect \
-f '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' \
quizzical_fermat
172.17.0.2
[student@workstation docker]$ curl -v 172.17.0.2:8081/nexus/
* About to connect() to 172.17.0.2 port 8081 (#0)
*   Trying 172.17.0.2...
* Connected to 172.17.0.2 (172.17.0.2) port 8081 (#0)
> GET /nexus HTTP/1.1
> User-Agent: curl/7.29.0
> Host: 172.17.0.2:8081
> Accept: */*
>
< HTTP/1.1 302 Found
< Date: Fri, 15 Jun 2018 22:59:27 GMT
< Location: http://172.17.0.2:8081/nexus/
< Content-Length: 0
< Server: Jetty(8.1.16.v20140903)
<
* Connection #0 to host 172.17.0.2 left intact
[student@workstation docker]$ docker kill quizzical_fermat
quizzical_fermat
```

4. Publish the Nexus server container image to the classroom private registry:

```
[student@workstation docker]$ docker tag nexus:latest \
registry.lab.example.com/nexus:latest
[student@workstation docker]$ docker push \
registry.lab.example.com/nexus:latest
The push refers to a repository [registry.lab.example.com/nexus]
92ee09aa3f02: Pushed
8759f7e05003: Pushed
fa7c74031ba6: Pushed
91ddbb7de6ee: Pushed
a8011fed33d7: Pushed
86d3cd59a99c: Pushed
d6a4dd6ace1f: Mounted from rhel7
f4fa6c253d2f: Mounted from rhel7
latest: digest:
sha256:b71aff6921e524fa6395b2843b2fc1a96ba6a455d1c3f59a783b51a358efff size:
1995
```

5. Deploy the Nexus server container image into the OpenShift cluster using the resources located beneath **/home/student/D0180/labs/review/deploy/openshift** directory.
- 5.1. Create the OpenShift project and set the security context policy by running the **setpolicy.sh** script located beneath **/home/student/D0180/labs/review/deploy/openshift** directory.

```
[student@workstation docker]$ cd ~/student/D0180/labs/review/deploy/openshift
```

```
[student@workstation openshift]$ oc login -u developer -p redhat \
https://master.lab.example.com
Login successful.
...output omitted...
[student@workstation openshift]$ oc new-project review
Now using project "review" on server "https://master.lab.example.com:443".
```

You can add applications to this project with the 'new-app' command. For example, try:

```
oc new-app centos/ruby-22-centos7-https://github.com/openshift/ruby-ex.git
```

to build a new example application in Ruby.

```
[student@workstation openshift]$ ./setpolicy.sh
scc "anyuid" added to: ["system:serviceaccount:review=default"]
```

- 5.2. Create the persistent volume that will hold the Nexus server's persistent data:

```
[student@workstation openshift]$ ./create-pv.sh
Warning: Permanently added 'node1,172.25.250.11' (ECDSA) to the list of known
hosts.
Warning: Permanently added 'node2,172.25.250.12' (ECDSA) to the list of known
hosts.
```

```
persistentvolume "pv-nexus" created
```

5.3. 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 "nexus" created
[student@workstation openshift]$ oc get pods
NAME           READY   STATUS    RESTARTS   AGE
nexus-1-wk8rv  1/1     Running   1          1m
```

5.4. Expose the service by creating a route:

```
[student@workstation openshift]$ oc expose svc/nexus
route "nexus" exposed
[student@workstation openshift]$ oc get route
NAME      HOST/PORT          PATH      SERVICES      PORT
TERMINATION  WILDCARD
nexus      nexus-review.apps.lab.example.com      nexus      nexus
None
```

5.5. Use a browser to connect to the Nexus server web application using `http://nexus-review.apps.lab.example.com/nexus/` URL.

Evaluation

After deploying the Nexus server container image into the OpenShift cluster, verify your work by running the lab grading script:

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

Cleanup

Delete the `review` project. Also delete the persistent volume using `delete-pv.sh` script.

```
[student@workstation review]$ oc delete project review
[student@workstation review]$ ./delete-pv.sh
```

This concludes the lab.

APPENDIX A

IMPLEMENTING MICROSERVICES ARCHITECTURE

GOAL

Design and build a custom container image for the deployment of an application containing multiple containers.

OBJECTIVES

- Divide an application across multiple containers to separate distinct layers and services.
- Implementing Microservices Architectures (with Guided Exercise)

SECTIONS

IMPLEMENTING MICROSERVICES ARCHITECTURES

OBJECTIVES

After completing this section, students should be able to:

- Divide an application across multiple containers to separate distinct layers and services.
- Describe typical approaches to decompose 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 DECOMPOSING AN APPLICATION TO DEPLOY INTO CONTAINERS

Among recommended practices for decomposing applications in microservices is running a minimum function set on each container. This is the opposite of traditional development where many distinct functions are packaged as a single deployment unit, or a *monolithic* application. In addition, traditional development may deploy supporting services, such as databases and other middleware services, on the same server as the application.

Having smaller containers and decomposing 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 parts of the application can be updated without affecting other parts of the same application.

Two popular ways of breaking an application are as follows:

- Tiers: based on architectural layers.
- Services: based on application functionality.

DIVIDING BASED ON LAYERS (TIERS)

Many applications are organized into tiers, based on how close the functions are to end users and how far from data stores. The traditional three-tier architecture: presentation, business logic, and persistence is a good example.

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.

Decomposing an application based on tiers allows developers to specialize in particular tier technologies. For example, there are web developers and database developers. 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. More demanding applications are often built and deployed in this monolithic fashion.

When a monolithic application is broken into tiers, it usually has to go through several changes:

- Connection parameters to database and other middleware services, such as messaging, were hard coded to fixed IPs 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 the page was downloaded from, as a security measure. The application needs to have provisions 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, shipping, and so on.

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, be updated, and be 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 a single monolithic unit using a service approach.

Containers enable architectures based on services to be realized during deployment. That is the reason microservices are so frequently talked about alongside containers. But containers alone are not enough; they need to be complemented by orchestration tools to manage dependencies among services.

Microservices can be viewed as taking 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 it cannot be truly broken into services. But refactoring it into presentation and business tiers, that is, into a front end and a back end to be deployed into distinct containers, illustrates the same kind of changes a typical application would need to be broken into services.

The following figure shows how the To Do List application would be deployed to three containers, one for each tier:

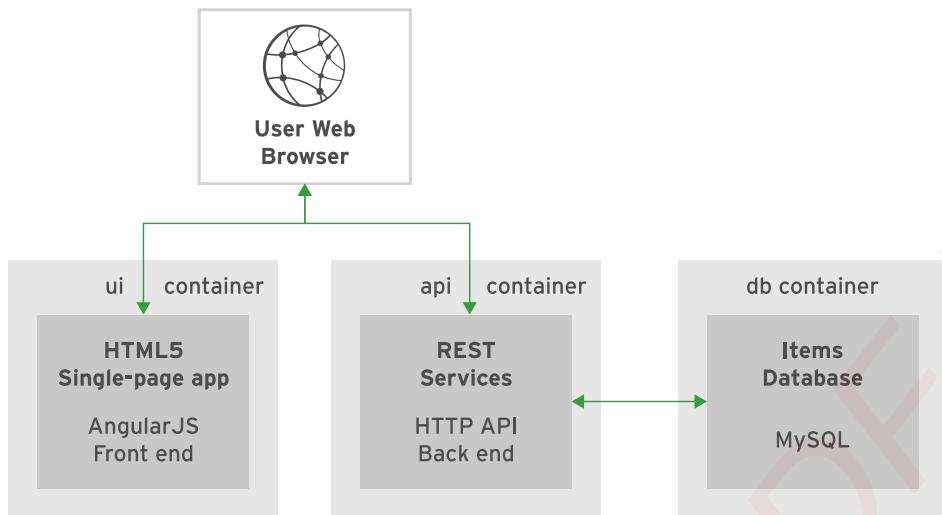


Figure A.1: To Do List application broken into tiers and each deployed as containers

Comparing the source code of the original monolithic application with the new one re-factored into services, following are the high-level 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 has their own syntax and features.



REFERENCES

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.

RESOURCES	
Files	/home/student/D0180/labs/todoapp /home/student/D0180/labs/appendix-microservices
Application URL	http://127.0.0.1:30080
Resources	MySQL 5.7 image (rhscl/mysql-57-rhel7), RHEL 7.5 Image (rhel7.5), To Do API image (d0180/todoapi_nodejs), To Do front-end image (d0180/todo_frontend)

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 setup
```

Create a new directory to host the new front-end application.

```
[student@workstation ~]$ mkdir -p ~/D0180/labs/appendix-microservices/apps/html5/src
```

► 1. Move 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 its own directory, so that it can be deployed 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/
```

12. 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 ~]$ cd /home/student/D0180/labs/appendix-microservices/
[student@workstation appendix-microservices]$ cd images/apache
[student@workstation apache]$ ./build.sh
```

- 2.2. Verify that the image is built correctly:

```
[student@workstation apache]$ docker images
REPOSITORY      TAG      IMAGE ID      CREATED      VIRTUAL SIZE
do180/httpd    latest   34376f2a318f  2 minutes ago  282.6 MB
...
```

- 2.3. Build the child Apache image:

```
[student@workstation ~]$ cd /home/student/D0180/labs/appendix-microservices/
[student@workstation appendix-microservices]$ cd deploy/html5
[student@workstation html5]$ ./build.sh
```

- 2.4. Verify that the image is built correctly:

```
[student@workstation html5]$ docker images
REPOSITORY          TAG      IMAGE ID      CREATED      VIRTUAL SIZE
do180/todo_frontend      latest   30b3fc531bc6  2
minutes ago          286.9 MB
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. Update these values to utilize environment variables instead. Edit the **/home/student/D0180/labs/appendix-microservices/apps/nodejs/models/db.js** file which holds the database configuration. Replace the contents with the following:

```
module.exports.params = {
  dbname: process.env.MYSQL_DATABASE,
  username: process.env.MYSQL_USER,
  password: process.env.MYSQL_PASSWORD,
  params: {
    host: "mysql",
    port: "3306",
    dialect: 'mysql'
  }
};
```



NOTE

This file can be copied and pasted from **/home/student/D0180/solutions/appendix-microservices/apps/nodejs_api/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 following line 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 semi-colon 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 ~]$ cd /home/student/D0180/labs/appendix-microservices/
[student@workstation appendix-microservices]$ cd deploy/nodejs
```

```
[student@workstation nodejs]$ ./build.sh
```

- 4.2. Run the **docker images** command to verify that all of the required images are built successfully:

```
[student@workstation nodejs]$ docker images
REPOSITORY      TAG      IMAGE ID      CREATED      SIZE
do180/httpd    latest   2963ca81ac51  5 seconds ago  249 MB
do180/todonodejs  latest   7b64ef105c50  7 minutes ago  533 MB
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
```

- 5.2. Run the **docker ps** command to confirm that all three containers are running:

```
[student@workstation linked]$ docker ps
CONTAINER ID  IMAGE          ...  PORTS
NAMES
11d0b1c095af  do180/todo_frontend  ...  0.0.0.0:30000->80/tcp
todo_frontend
40435ef68533  do180/todonodejs     ...  8080/tcp, 0.0.0.0:30080->30080/tcp
todoapi
9cde95ff22ca  do180/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 127.0.0.1:30080/todo/api/items/1
{"description": "Pick up newspaper", "done": false, "id":1}
```

- 6.2. Open Firefox on **workstation** machine and navigate to <http://127.0.0.1:30000>, where you should see the **To Do List** application.

- 6.3. Verify that the correct images were built and that the application is running correctly:

```
[student@workstation linked]$ lab appendix-microservices grade
```

Clean Up

Stop the running containers:

```
[student@workstation linked]$ docker stop todoapi todo_frontend mysql
```

Remove the containers:

```
[student@workstation linked]$ docker rm todoapi todo_frontend mysql
```

Remove the images created during the exercise:

```
[student@workstation linked]$ docker rmi -f \
do180/todonodejs do180/todo_frontend \
do180/httpd
```

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 from 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.
- Passing environment variables into a container is not an ideal solution for starting an application composed of multiple containers, because it is prone to typing mistakes, or the connection information can be dynamic. Docker's user-defined networks resolve this issue.