Comprehensive, hands-on training that solves real world problems

Introduction to Containers, Kubernetes, and Red Hat OpenShift

Student Workbook (ROLE)

DO180-OCP3.5-en-1-20170524

INTRODUCTION TO CONTAINERS, KUBERNETES, AND RED HAT OPENSHIFT

OCP 3.5 DO180
Introduction to Containers, Kubernetes, and Red Hat OpenShift
Edition 1 20170524 20170524

Authors: Ravishankar Srinivasan, Fernando Lozano, Richard Allred, Ricardo Taniguchi, Jim Rigsbee
Editor: David O'Brien, Dave Sacco

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.

Contributors: Jim Rigsbee, George Hacker, Philip Sweany, Rob Locke

Document Conventions

Notes and Warnings

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.

References

"References" describe where to find external documentation relevant to a subject.

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.

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, **infrastructure**, with the host name **infrastructure.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, **ocp**, with the host name **ocp.lab.example.com**, hosts the OpenShift Container Platform (OCP) 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.

Students do not need access to the **infrastructure** VM to perform tasks for this course, but it needs to be up and running. Students will need to access the **ocp** VM to start the OpenShift cluster and configure persistent storage for OpenShift pods. Most other actions related to OpenShift will be performed from the **workstation** VM using the OpenShift client.

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

Classroom Machines

Machine name	IP addresses	Role
content.example.com, **materials.example.com**, **classroom.example.com**	172.25.254.254, 172.25.252.254	Classroom utility server
workstation.lab.example.com, **workstation*X*.example.com**	172.25.250.254, 172.25.252.*X*	Student graphical workstation
infrastructure.lab.example.com	172.25.250.10	Student development infrastructure server
ocp.lab.example.com	172.25.250.11	OpenShift Container Platform cluster server

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 **workstation*X*.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 turn back the clock and start the activity over again, usually followed by **`setup`**.

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

Button or Action	Description
	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
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

Overview	
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.
Sections	• Container Architecture (and Quiz) • Docker Architecture (and Quiz) • Container Orchestration with Kubernetes and OpenShift (and Quiz)

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 next diagram, there might be a risk of breaking it if an update that fixes the first application libraries affects the second application.

Thus, 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.

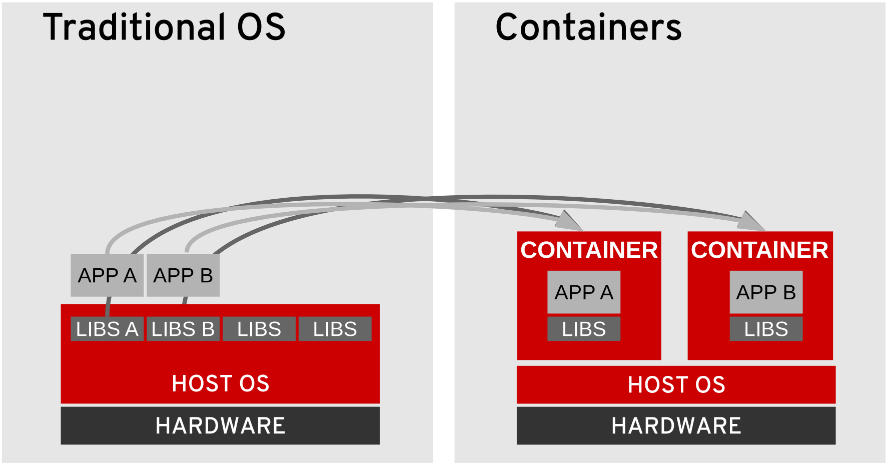


Figure 1.1: Container versus operating system differences

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 the previous diagram.

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 isolation 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 for a full OS install or restart. 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, filesystems) 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. Blog software (such as WordPress)
   c. Database
   d. A local file system recovery tool

e. A web server

Solution

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. **Blog software (such as WordPress)**
   c. **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).

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, 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 pre-existing 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 a nutshell, to create a new image, there are two approaches:

- *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

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

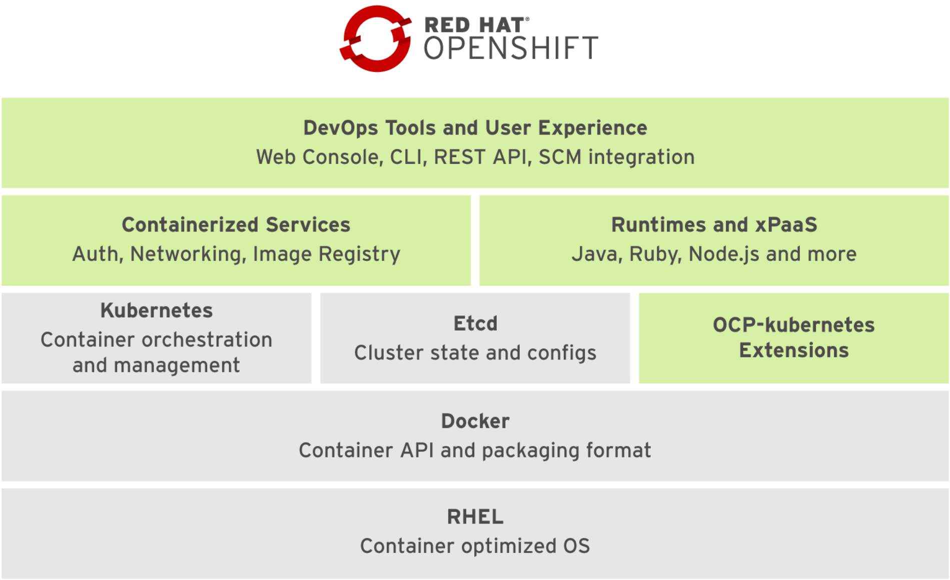
After completing this section, students should be able to:

- 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.

OpenShift 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, 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 following figure illustrates the OpenShift software stack.



RED HAT OPENSHIFT

DevOps Tools and User Experience
Web Console, CLI, REST API, SCM integration

Containerized Services
Auth, Networking, Image Registry

Runtimes and xPaaS
Java, Ruby, Node.js and more

Kubernetes
Container orchestration and management

Etcd
Cluster state and configs

OCP-kubernetes Extensions

Docker
Container API and packaging format

RHEL
Container optimized OS

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 works with *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 keeps it moving.

- *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 + 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 leveraged 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 Keywords

Term	Definition
Master	A server that manages the workload and communications in a Kubernetes cluster.
Node	A server that performs work 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.

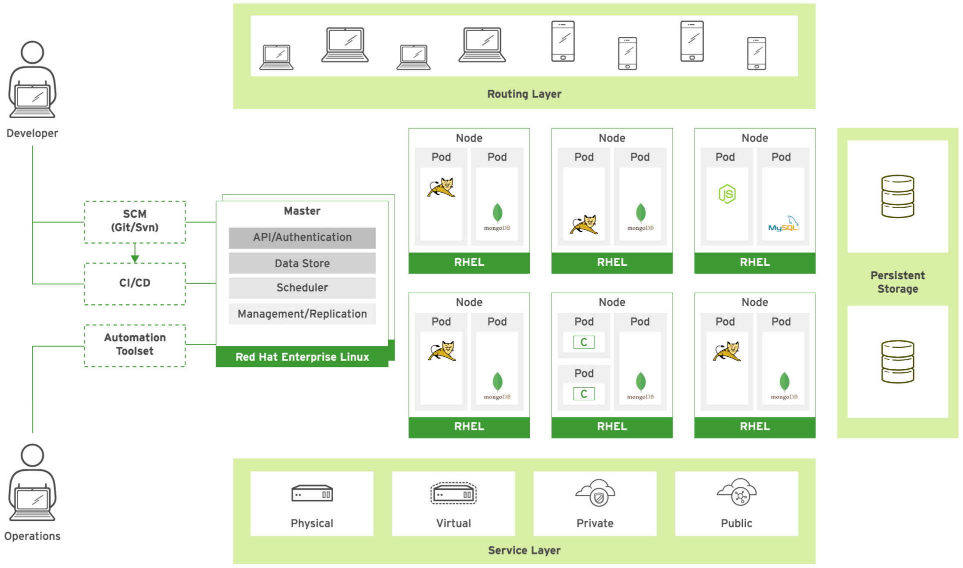


Figure 1.3: OpenShift and Kubernetes architecture

An OpenShift cluster is a Kubernetes cluster, and can be managed the same way, but using the management tools provided OpenShift (CLI/Web Console) 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 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 higher-level resources such as replication controllers.

OpenShift Resource Types

The main resource types added by OCP 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/Continuous Delivery workflow.

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 on from host 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 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 increasing/decreasing the number of pods from 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? (Select one.)

   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? (Select one.)

   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

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 increasing/decreasing the number of pods from 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? (Select one.)

    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? (Select one.)

    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/ Continuous Development features.
- OpenShift routes are key to exposing containerized applications to external users in a manageable way.

CHAPTER 2

CREATING CONTAINERIZED SERVICES

Overview	
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	• Building a Development Environment (and Guided Exercise) • 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:

- Identify three ways to build an environment where you can build, test, and deploy containers.
- Describe the process for deploying containers with an installed OpenShift cluster.
- Describe the process for developing containerized applications locally using Red Hat Container Development Kit 3 (Minishift).
- Run an OpenShift cluster locally with the OpenShift command-line interface and Docker.

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 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 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 which provides 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 runs 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	Hypervisor
MacOS	xhyve (default)
	VirtualBox
	VMware Fusion (supported in upstream Minishift only)
GNU/Linux	KVM (default)
	VirtualBox
Windows	Hyper-V (default)
	VirtualBox

The following procedures assume that the CDK is running on MacOS or GNU/Linux. Prepare the Minishift binary by renaming it to **minishift** and setting the execute permission using the **chmod** command. Run the following command to allow Minishift to do its required setup:

```
$ ./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/v1.5.0** is in the environment **PATH** so that the OpenShift command-line interface can be executed. The version number in the path may vary with the specific version of the CDK.

The **./minishift config view** command shows all the properties that are set for the creation of the virtual machine and OpenShift cluster. To update values, use the **./minishift config set {property} {value}** command. The most common properties include:

- **vm-driver**

  Specify which virtual machine driver Minishift will use 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 will allocate.

- **memory**

  The amount of memory, in MB, that the virtual machine will allocate.

- **openshift-version**

  The specific version of OpenShift Origin to run. It is a string starting with the letter "v", for example, **v1.5.0**.

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

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

```
$ ./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. 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. When the developer is finished with the cluster, the command **./minishift stop** can be issued. To destroy the virtual machine and lose all work, run the **./minishift delete** command. If the developer needs access to the virtual machine directly, run the **./minishift ssh** command. To determine the status of the virtual machine, run the **./minishift status** command.

To launch the web console in the default browser, run the **./minishift console** command.

Developing with a Local OpenShift 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.

The Docker daemon default configuration considers the local OpenShift cluster registry an insecure one because it uses a self-signed SSL certificate, and refuses to pull images from it.

To allow creating containers from images created using OpenShift Source-to-Image feature, the Docker configuration has to changed to allow using the OpenShift insecure registry. Edit the **`/etc/sysconfig/docker`** file and modify the **`INSECURE_REGISTRY`** parameter as follows:

```
INSECURE_REGISTRY='--insecure-registry 172.30.0.0/16'
```

It is acceptable to have more than one insecure registry defined. List each insecure registry on the same line. On Docker for Mac, this setting is made through the Docker for Mac preferences on the **Daemon** tab. After making this configuration change, restart the Docker daemon.

The cluster is started on the local machine using the **`oc cluster up`** command. For information on this command, 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 will be written. If the **`--host-data-dir`** option is not specified, no data will be saved and the cluster state will not be preserved over a shutdown.

`--host-volumes-dir`

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

The version of OpenShift can be controlled by the **`--version`** option. An example version is **`v3.5.5.5`**. If the developer needs 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='myreg:5000/openshift3/ose'
```

The complete suite of OpenShift containers will be 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

1. From the **workstation** machine, **ssh** into **ocp.lab.example.com** as the user **student**.

2. Establish that **docker** has been installed, enabled, and is running.

```
[student@ocp ~]$ sudo systemctl status docker
● docker.service - Docker Application Container Engine
   Loaded: loaded (/usr/lib/systemd/system/docker.service; enabled; vendor preset:
 disabled)
   Active: active (running) since Wed 2017-04-12 15:45:41 UTC; 1h 4min ago
...
[student@ocp ~]$ docker version
Client:
 Version:         1.12.6
 API version:     1.24
 Package version: docker-common-1.12.6-16.el7.x86_64
 Go version:      go1.7.4
 Git commit:      3a094bd/1.12.6
 Built:           Tue Mar 21 13:30:59 2017
 OS/Arch:         linux/amd64

Server:
 Version:         1.12.6
 API version:     1.24
 Package version: docker-common-1.12.6-16.el7.x86_64
 Go version:      go1.7.4
 Git commit:      3a094bd/1.12.6
 Built:           Tue Mar 21 13:30:59 2017
 OS/Arch:         linux/amd64
```

   The current Docker version may differ from the output given above.

3. Establish that the OpenShift command line interface has been installed.

```
[student@ocp ~]$ oc version
oc v3.5.5.5
kubernetes v1.5.2+43a9be4
features: Basic-Auth GSSAPI Kerberos SPNEGO
```

   The current OpenShift version may differ from the output given above.

4. Verify that **172.30.0.0/16** has been set up as an insecure registry for Docker.

```
[student@ocp ~]$ grep -i insecure /etc/sysconfig/docker
# adding the registry to the INSECURE_REGISTRY line and uncommenting it.
INSECURE_REGISTRY='--insecure-registry infrastructure.lab.example.com:5000
--insecure-registry 172.30.0.0/16'
```

5. Review the output of the **oc cluster up -h** command. Notice the following items:

- The configuration and cluster data is, by default, ephemeral. The **--host-config-dir**, **--host-data-dir**, and **--use-existing-config** command options preserve the cluster state across cluster shutdown.
- The specific version of OpenShift to run can be controlled by the **--version** command option.
- The host name for the web console can be specified by the **--public-hostname** command option.

6. Examine the **oc cluster up** command in the **/home/student/ocp-up.sh** shell script. Discuss the purpose of the values for each command-line option in the script. This classroom is set up to run offline, and so the OpenShift container images have to be loaded from a private registry running at **infrastructure.lab.example.com:5000**.

7. Bring the cluster up by running the **./ocp-up.sh** command. Wait for the cluster to complete its initialization. A successful start produces output similar to the following:

```
-- Checking OpenShift client ... OK
-- Checking Docker client ... OK
-- Checking Docker version ... OK
-- Checking for existing OpenShift container ... OK
-- Checking for infrastructure.lab.example.com:5000/openshift3/ose:v3.5.5.5
 image ... OK
...
   OpenShift server started.
   The server is accessible via web console at:
       https://172.25.250.11:8443

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

8. Log in as the **system:admin** user and list the pods running in the **default** namespace.

```
[student@ocp ~]$ oc login -u system:admin
[student@ocp ~]$ oc get pods -n default
NAME                         READY     STATUS    RESTARTS   AGE
docker-registry-1-2z9v5      1/1       Running   2          3h
persistent-volume-setup-q87j0 1/1       Running   2          3h
router-1-zlzp0               1/1       Running   2          3h
```

9. Display the containers that are running the OpenShift Cluster:

```
[student@ocp ~]$ docker ps | grep ose
```

With this method of running an OpenShift cluster, the only thing installed on the operating system is the OpenShift command-line interface. The OpenShift cluster runs entirely in containers.

10. Exit from the **ssh** shell on the **ocp** server. You should now be logged in as **student** on the **workstation** server.

11. Access the OpenShift cluster remotely using the **oc login** command and display the same pods again.

```
[student@workstation ~]$ oc login -u developer -p developer \
https://ocp.lab.example.com:8443
[student@workstation ~]$ oc get pods -n default
```

12. From the **workstation** machine, open an SSH session into **ocp.lab.example.com** as the user **student**.

13. Bring the cluster down.

```
[student@ocp ~]$ oc cluster down
```

14. Show where the configuration and cluster data is being stored at **/var/lib/origin**. The student must bring the cluster down and remove this folder recursively as **root** to start up a fresh cluster again if they use the persistent options we have used in this demonstration.

    This concludes this demonstration.

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/

Guided Exercise: Starting an OpenShift Cluster

In this exercise, you will start an all-in-one OpenShift cluster for a developer user.

Resources	
Files:	N/A
Application URL:	**`https://ocp.lab.example.com:8443`**

Outcomes

You should be able to start and access an all-in-one OCP instance for a developer.

Before you begin

There are no prerequisites for this exercise.

Steps

1. Check if Docker is installed and running.

   1.1. Open a terminal on **`workstation`** and start an SSH session to the **`ocp`** host:

   ```
   [student@workstation ~]$ ssh ocp
   ```

   The **`ocp`** VM is already configured to accept SSH connections from **`workstation`** without requiring a password.

   1.2. Check that the *docker* RPM package is installed:

   ```
   [student@ocp ~]$ rpm -q docker
   docker-1.12.6-11.el7.x86_64
   ```

   1.3. Check that the **`docker`** service is loaded and active:

   ```
   [student@ocp ~]$ systemctl status docker
   ● docker.service - Docker Application Container Engine
      Loaded: loaded (/usr/lib/systemd/system/docker.service; enabled; vendor
    preset: disabled)
      Active: active (running) since Mon 2017-04-03 14:58:42 UTC; 7h ago
        Docs: http://docs.docker.com
    Main PID: 1225 (dockerd-current)
   ...
   ```

   1.4. Check that the OpenShift client RPM (*atomic-openshift-clients*) package is installed:

   ```
   [student@ocp ~]$ rpm -q atomic-openshift-clients
   atomic-openshift-clients-3.5.5.5-1.git.0.f2e87ab.el7.x86_64
   ```

2. Start a persistent all-in-one OCP cluster

   2.1. Inspect the **`ocp-up.sh`** script and review the **`oc cluster up`** command-line options required by the classroom environment.

```
[student@ocp ~]$ cat ~/ocp-up.sh
#!/bin/bash
up=$(oc cluster status | grep started | wc -l)
if [ $up -eq 0 ]; then
  oc cluster up --public-hostname='ocp.lab.example.com'
--host-data-dir='/var/lib/origin/etcd' --use-existing-config
--image='infrastructure.lab.example.com:5000/openshift3/ose'
--version='v3.5.5.5' --routing-suffix='cloudapps.lab.example.com'
...
```

These options ensure that the cluster works without an active internet connection, using the container images from the private registry in the **infrastructure** VM, and configures a default DNS suffix for OpenShift routes that follows the classroom network settings.

A few other tasks are performed by the **ocp-up.sh** script for your convenience:

- Add OCP cluster administrator privileges to the **developer** user.
- Patch predefined image streams to pull s2i builder images from the classroom private registry.

2.2. Invoke the script to start the OCP cluster.

```
[student@ocp ~]$ ~/ocp-up.sh
-- Checking OpenShift client ... OK
-- Checking Docker client ... OK
-- Checking Docker version ... OK
-- Checking for existing OpenShift container ... OK
-- Checking for infrastructure.lab.example.com:5000/openshift3/ose:v3.5.5.5
 image ... OK
...
-- Starting OpenShift container ...
...
-- Installing registry ... OK
-- Installing router ... OK
...
-- Creating initial project "myproject" ... OK
...
   OpenShift server started.
...
```

2.3. Check that the **origin** container is running:

```
[student@ocp ~]$ docker ps
CONTAINER ID  IMAGE
  COMMAND                  CREATED            STATUS         PORTS     NAMES
...
c4fe03a8d7f0  infrastructure.lab.example.com:5000/openshift3/ose:v3.5.5.5
  "/usr/bin/openshift s"   51 seconds ago     Up 49 seconds            origin
```

The OCP cluster starts several other containers apart from the **origin** container, but this is the container that runs OpenShift master and node services. The **openshift3/ose** image is the all-in-one OCP container image.

3. Check that the registry and router pods are ready and running.

These pods provide essential OCP features. If they are not working you may still be able to create simple pods but will not be able to use other OpenShift features such as Source-to-Image (S2I).

```
[student@ocp ~]$ oc get pod -n default
NAME                           READY     STATUS    RESTARTS   AGE
docker-registry-1-10e3r        1/1       Running   0          10m
persistent-volume-setup-q87j0 1/1       Running   0          11m
router-2-2v5ko                 1/1       Running   0          9m
```

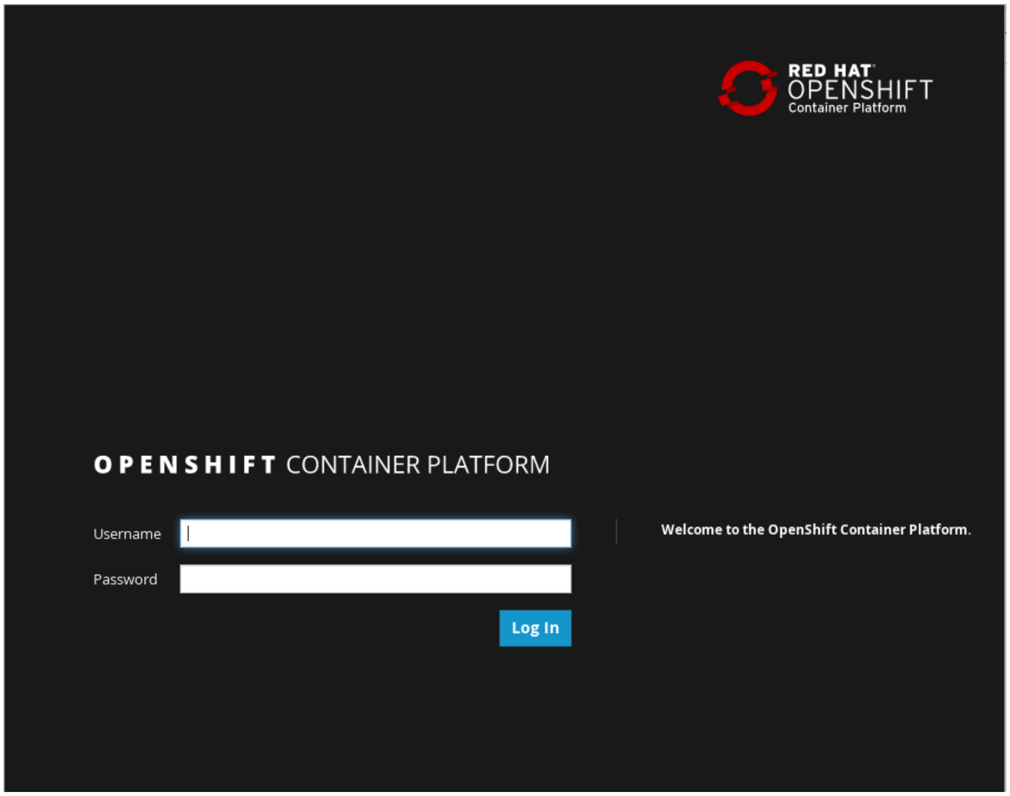
The pod names will have different suffixes each time the cluster is started.

4. Access the web console as a developer user.

   4.1. Open the OCP web console.

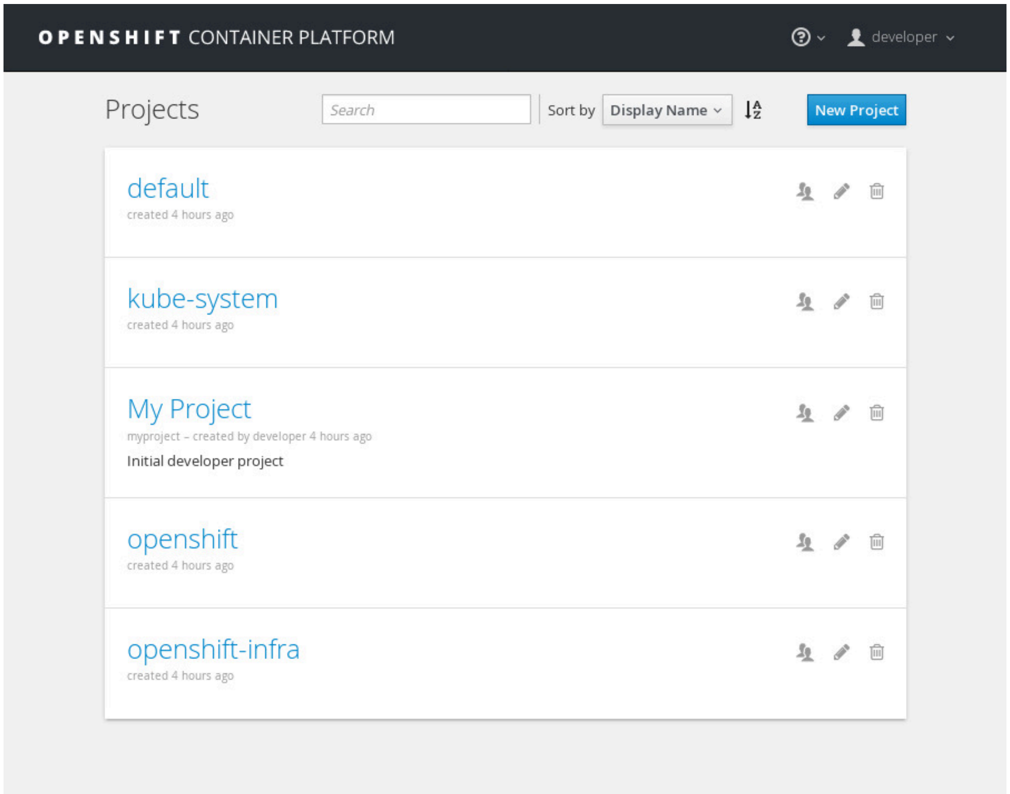
   Open a web browser on **workstation** and navigate to **https://ocp.lab.example.com:8443**. The web browser will indicate that the connection is insecure, because it uses a self-signed SSL certificate. Accept this certificate and proceed with the connection.

   You will see the web console login page.


RED HAT
OPENSHIFT
Container Platform
OPENSHIFT CONTAINER PLATFORM
Username
Password
Log In
Welcome to the OpenShift Container Platform.

   4.2. Log in as the **developer** user. The password is **developer**.

   You will see the web console projects page. The **My Project** project is created the first time you start the all-in-one OCP cluster.



default
created 4 hours ago

kube-system
created 4 hours ago

My Project
myproject – created by developer 4 hours ago
Initial developer project

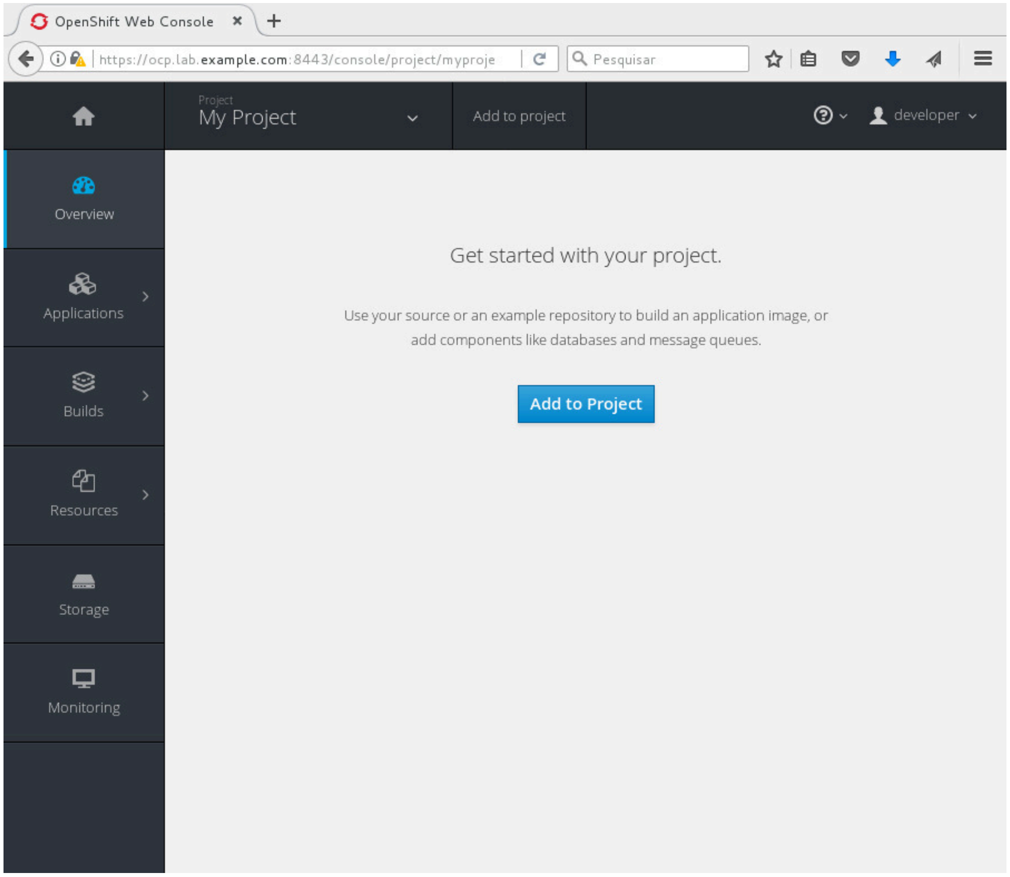
openshift
created 4 hours ago

openshift-infra
created 4 hours ago

All the projects in the list, except for **`My Project`**, are internal OCP projects used by the OpenShift infrastructure. Normal users should not make changes to OpenShift internal project, but because the **`developer`** user was given the cluster administration role, he has permission to change these projects.

4.3. Enter the My Project page.

Click **My Project** to enter the project page. Notice that the initial project is empty.


OpenShift Web Console
https://ocp.lab.example.com:8443/console/project/myproje
Pesquisar
Project
My Project
Add to project
developer
Overview
Applications
Builds
Resources
Storage
Monitoring
Get started with your project.
Use your source or an example repository to build an application image, or add components like databases and message queues.
Add to Project

5. Clean up.

5.1. Stop the all-in-one OCP cluster.

Close the web browser and go back to the terminal connected to the **ocp** VM. Run the following command:

```
[student@ocp ~]$ oc cluster down
```

5.2. Check that no containers are left, either running or stopped.

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

5.3. Check that the OCP container images are still in the local Docker cache.

```
[student@ocp ~]$ docker images
REPOSITORY                                                     TAG
         IMAGE ID            CREATED             SIZE
infrastructure.lab.example.com:5000/openshift3/ose-haproxy-router   v3.5.5.5
        7a430959c00c        4 weeks ago         745.2 MB
infrastructure.lab.example.com:5000/openshift3/ose-deployer         v3.5.5.5
        fa7264fa4dea        4 weeks ago         726.6 MB
infrastructure.lab.example.com:5000/openshift3/ose-docker-registry  v3.5.5.5
        3809e30272a6        4 weeks ago         806.5 MB
```

```
infrastructure.lab.example.com:5000/openshift3/ose                   v3.5.5.5
        e0f949132baa        4 weeks ago         726.6 MB
infrastructure.lab.example.com:5000/openshift3/ose-pod               v3.5.5.5
        9c11230663aa        4 weeks ago         205 MB
```

The **`oc cluster down`** command leaves the container images in the local Docker cache to speed up the next **`oc cluster up`** command. If the user removes them, they will be pulled again.

5.4. Close the SSH session and go back to the workstation VM prompt.

```
[student@ocp ~]$ logout
Connection to ocp closed.
[student@workstation ~]$
```

This concludes the guided exercise.

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.

Finding an Image on Docker Hub

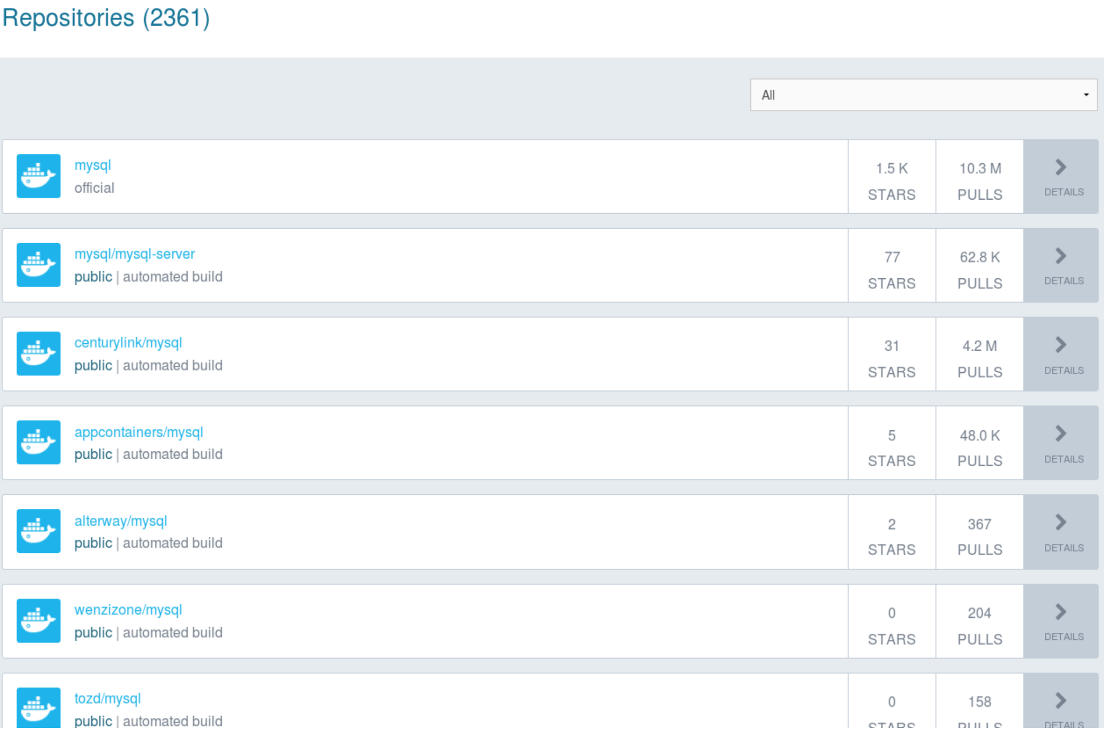
Many container images are available for download from the Docker community website at **`https://docker.io`**. It is a large repository where developers and administrators can get a number of container images developed by the community and some companies.

Anyone can publish images to Docker Hub after they register. For larger projects (with multiple images), a paid subscription is needed.

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.



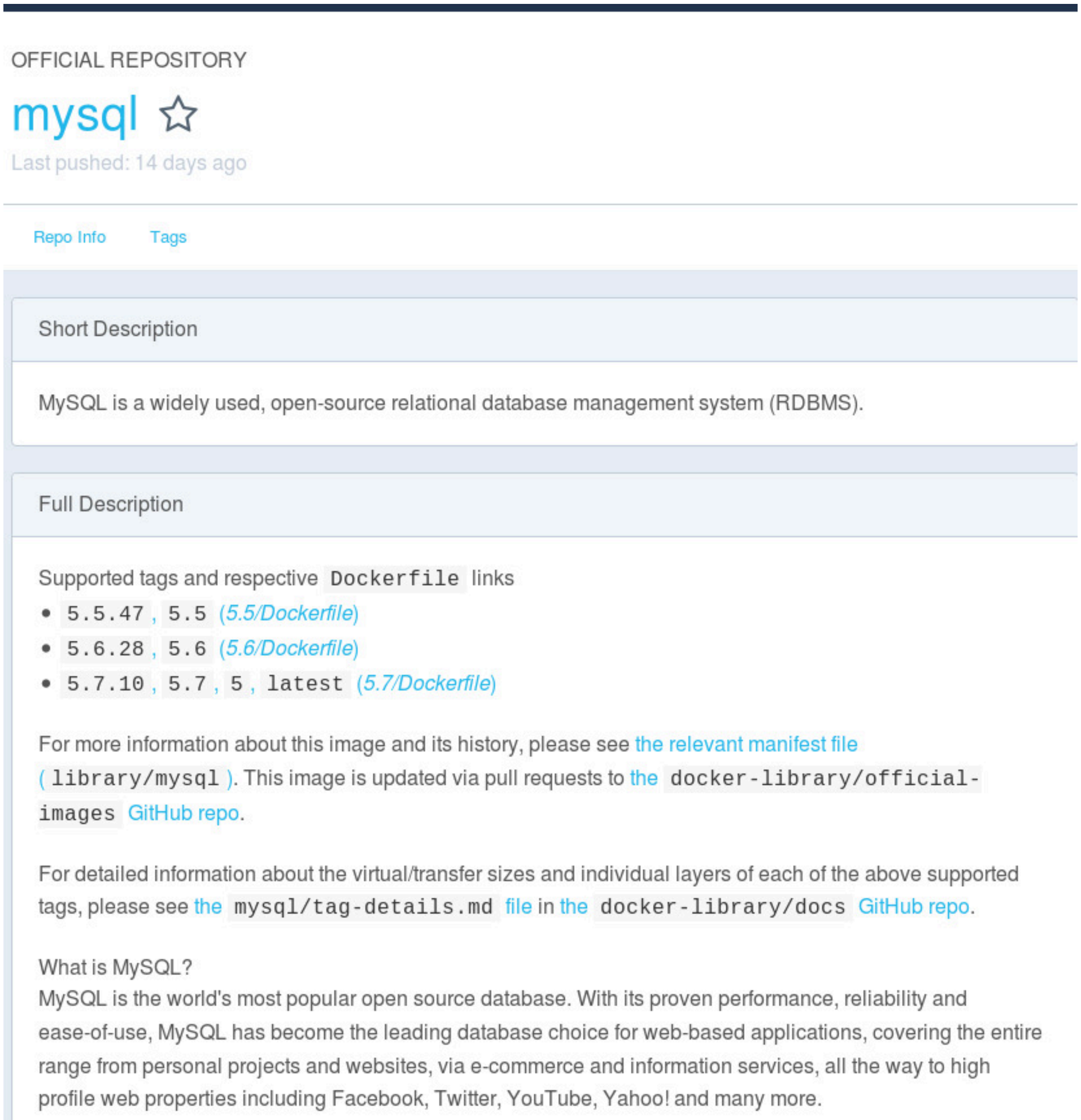
Repositories (2361)

All

mysql	official	1.5 K STARS	10.3 M PULLS	DETAILS
mysql/mysql-server	public | automated build	77 STARS	62.8 K PULLS	DETAILS
centurylink/mysql	public | automated build	31 STARS	4.2 M PULLS	DETAILS
appcontainers/mysql	public | automated build	5 STARS	48.0 K PULLS	DETAILS
alterway/mysql	public | automated build	2 STARS	367 PULLS	DETAILS
wenzizone/mysql	public | automated build	0 STARS	204 PULLS	DETAILS
tozd/mysql	public | automated build	0	158	

Figure 2.4: Search results page from Docker Hub

Clicking an image name to display the container image details page. The details page is not guaranteed to display any particular information. Different authors provide different levels of information about the images they create.



OFFICIAL REPOSITORY

mysql ☆

Last pushed: 14 days ago

Short Description

MySQL is a widely used, open-source relational database management system (RDBMS).

Full Description

Supported tags and respective `Dockerfile` links

- `5.5.47`, `5.5` (5.5/Dockerfile)
- `5.6.28`, `5.6` (5.6/Dockerfile)
- `5.7.10`, `5.7`, `5`, `latest` (5.7/Dockerfile)

For more information about this image and its history, please see the relevant manifest file (`library/mysql`). This image is updated via pull requests to the `docker-library/official-images` GitHub repo.

For detailed information about the virtual/transfer sizes and individual layers of each of the above supported tags, please see the `mysql/tag-details.md` file in the `docker-library/docs` GitHub repo.

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 2.5: Detailed information about the image

Searching From the Docker Client

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

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

The search uses the Docker Hub registry and also any other version 1-compatible registries configured in the local Docker daemon.

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:

```
Get http:///var/run/docker.sock/v1.20/version: dial unix /var/run/docker.sock:
 permission denied.
```

```
* Are you trying to connect to a TLS-enabled daemon without TLS?
* Is your docker daemon up and running?
```

> **Note**
>
> For a production environment, the **docker** command access should be given via 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 **docker search** command and use **docker pull** command:

```
[student@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** assumes 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:

```
[student@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:

```
# docker images
REPOSITORY                    TAG                 IMAGE ID            CREATED
       VIRTUAL SIZE
docker.io/mysql               5.5                 5bf0aea4b3e1        3 weeks ago
       256.4 MB
docker.io/mysql               latest              a5ad9eb2ff48        4 weeks ago
       359.8 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 will be 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**.

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

The **docker images** command shows that each image has a unique ID. This allows checking 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.

```
# 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 example output, the container was started with a noninteractive process, and 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:

```
# docker run -d mysql:5.5
```

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

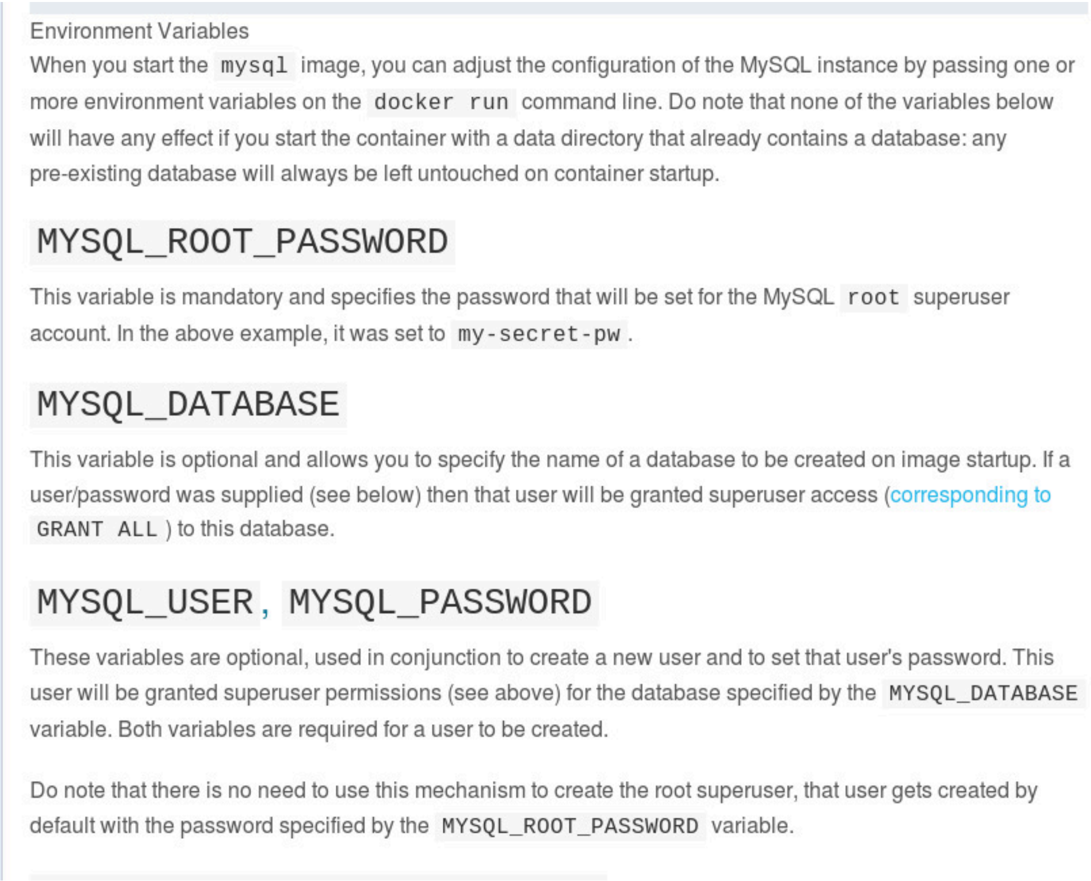
```
# docker run --name mysql-container mysql:5.5
```

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:

```
# docker run --name mysql-container -it mysql:5.5 /bin/bash
[root@8682f6516d6f /]#
```

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 next image is a snapshot from the MySQL official image documentation listing all the 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.

`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.

Figure 2.6: Environment variables supported by the MySQL official Docker Hub image

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

```
# docker run --name mysql-custom \
-e MYSQL_USER=redhat -e MYSQL_PASSWORD=r3dh4t \
-d mysql:5.5
```

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:	Docker Hub Official MySQL 5.6 image (**`mysql:5.6`**)

Outcomes

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

Before you begin

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

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

Steps

1. Create a MySQL container instance.

   1.1. Start a container from the Docker Hub MySQL image.

   Open a terminal on **`workstation`** (**Applications** > **Utilities** > **Terminal**) and run the following command:

   ```
   [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 mysql:5.6
   ```

   This command downloads (pulls) the **`mysql`** container image with the **`5.6`** tag, and starts a container based on it. It creates a database named **`items`**, owned by a user named **`user1`**, with **`mypa55`** as the password. The database administrator user password is set to **`r00tpa55`**. The container runs in the background.

   1.2. Check that the container started without errors.

   Open a terminal on **`workstation`** and check if the container was started correctly. Run the following command:

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

   An output similar to the following should appear:

   ```
   CONTAINER ID    IMAGE       COMMAND                     CREATED         STATUS
       PORTS      NAMES
   ```

```
13568029202d    mysql:5.6   "docker-entrypoint.sh"   6 seconds ago   Up 4
 seconds   3306/tcp   mysql-basic
```

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

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

The command starts a Bash shell, running as root inside the MySQL container:

```
root@13568029202d:/#
```

Note

The prompt displays the container ID. This is a characteristic of the official MySQL image from Docker Hub, and other images may display a different Bash prompt.

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:

```
root@13568029202d:/# mysql -pr00tpa55
```

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 |
+--------------------+
4 rows in set (0.00 sec)
```

3.2. Create a new table in the **items** database. From the MySQL prompt, run the following command to access the database:

```
mysql> use items;
```

3.3. From the MySQL prompt, 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));
```

3.4. Run the following command to verify that the table was created:

```
mysql> show tables;
+---------------------+
| Tables_in_items     |
+---------------------+
| Projects            |
+---------------------+
```

3.5. Insert a row in the table by running the following command:

```
mysql> insert into Projects (id, name, code) values (1,'DevOps','DO180');
```

3.6. Run the following command to verify that the project information was added to the table:

```
mysql> select * from Projects;
+----+-----------+-------+
| id | name      | code  |
+----------------+-------+
|  1 | DevOps    | DO180 |
+----------------+-------+
```

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

```
mysql> exit
Bye
root@13568029202d:/# exit
exit
```

4. Verify that the database was correctly set up. Run the following command:

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

5. Undo the changes made by the lab when you are finished:

5.1. Stop the running container by running the following command:

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

5.2. Remove the data from the stopped container by running the following command:

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

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

```
[student@workstation ~]$ docker rmi mysql:5.6
```

This concludes the guided exercise.

Lab: Creating Containerized Services

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

Resources	
Files:	N/A
Application URL:	**`http://localhost:8081`**
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.

Before you begin

Open a terminal and run the following command to check that Docker is running:

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

Steps

1. Find the documentation for the Apache HTTP Server container image in Docker Hub.

   Open a web browser (**Applications** > **Internet** > **Firefox**) and navigate to **`https://hub.docker.com/_/httpd/`**. This will open the Docker Hub documentation for the official Apache HTTP Server container image.

   If your classroom has no access to the internet, the Docker Hub Apache Http Server container documentation is available from **`~/DO180/labs/httpd-basic/httpd-container-image.pdf`**.

   1.1. Check how to start a container from the Apache HTTP Server container image using the **`docker`** command.

   ## Insight

   The command provided by the Docker Hub documentation is complex and it publishes web pages from a volume. A simpler approach is used in this lab.

   The documentation provides the following command:

   ```
   # Do not run!
   docker run -it --rm --name my-apache-app \
   -v "$PWD":/usr/local/apache2/htdocs/httpd:2.4
   ```

   1.2. Start a container named **`httpd-basic`** and forward port 8081 from **`workstation`** to port 80 in the container. Use the **`httpd`** container and the **`2.4`** tag.

> ### Note
>
> Use the **docker run -p 8081:80** command to forward the port.

This command starts the Apache HTTP Server in the foreground and does not return to the Bash prompt. Leave it running.

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

   Open a new web browser on **workstation** and navigate to **http://localhost:8081**.

   An "It works" message is displayed, which is the default welcome page from the Apache HTTP Server community container.

3. Customize the **httpd-basic** container.

   3.1. Start a Bash session inside the container to create a web page.

   3.2. From the Bash session, check the directory structure using the **ls -la** command.

   3.3. Search in the Docker Hub Apache HTTP Server container image documentation for the folder where web pages are stored.

   The pages are stored in the **/usr/local/apache2/htdocs** folder.

   3.4. Change the **index.html** page to contain the text **Hello World**, without HTML tags.

   3.5. Check if the updated web page contents can be accessed by refreshing the open web browser.

4. Grade your work.

   Open a terminal on **workstation** and 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

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

Resources	
Files:	N/A
Application URL:	**`http://localhost:8081`**
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.

Before you begin

Open a terminal and run the following command to check that Docker is running:

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

Steps

1. Find the documentation for the Apache HTTP Server container image in Docker Hub.

   Open a web browser (**Applications** > **Internet** > **Firefox**) and navigate to **`https://hub.docker.com/_/httpd/`**. This will open the Docker Hub documentation for the official Apache HTTP Server container image.

   If your classroom has no access to the internet, the Docker Hub Apache Http Server container documentation is available from **`~/DO180/labs/httpd-basic/httpd-container-image.pdf`**.

   1.1. Check how to start a container from the Apache HTTP Server container image using the **`docker`** command.

Insight

The command provided by the Docker Hub documentation is complex and it publishes web pages from a volume. A simpler approach is used in this lab.

The documentation provides the following command:

```
# Do not run!
docker run -it --rm --name my-apache-app \
-v "$PWD":/usr/local/apache2/htdocs/httpd:2.4
```

   1.2. Start a container named **`httpd-basic`** and forward port 8081 from **`workstation`** to port 80 in the container. Use the **`httpd`** container and the **`2.4`** tag.

Note

Use the **`docker run -p 8081:80`** command to forward the port.

Run the following command from **`workstation`**:

```
[student@workstation ~]$ docker run -p 8081:80 \
--name httpd-basic httpd:2.4
```

This command starts the Apache HTTP Server in the foreground and does not return to the Bash prompt. Leave it running.

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

   Open a new web browser on **workstation** and navigate to **http://localhost:8081**.

   An "It works" message is displayed, which is the default welcome page from the Apache HTTP Server community container.

3. Customize the **httpd-basic** container.

   3.1. Start a Bash session inside the container to create a web page.

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

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

   3.2. From the Bash session, check the directory structure using the **ls -la** command.

```
root@e91768d643dd:/usr/local/apache2# ls -la
```

   The expected output is similar to the following:

```
total 28
drwxr-sr-x. 12 www-data www-data  139 Mar 21 20:50 .
drwxrwsr-x. 11 root     staff     129 Mar 21 20:50 ..
drwxr-sr-x.  2 root     www-data  276 Mar 21 20:50 bin
drwxr-sr-x.  2 root     www-data  167 Mar 21 20:50 build
drwxr-sr-x.  2 root     www-data   78 Mar 21 20:50 cgi-bin
drwxr-sr-x.  4 root     www-data   84 Mar 21 20:50 conf
drwxr-sr-x.  3 root     www-data 4096 Mar 21 20:50 error
drwxr-sr-x.  2 root     www-data   24 Mar 21 20:50 htdocs
drwxr-sr-x.  3 root     www-data 8192 Mar 21 20:50 icons
drwxr-sr-x.  2 root     www-data 4096 Mar 21 20:50 include
drwxr-sr-x.  2 root     www-data   23 Apr 19 00:58 logs
drwxr-sr-x.  2 root     www-data 4096 Mar 21 20:50 modules
```

   3.3. Search in the Docker Hub Apache HTTP Server container image documentation for the folder where web pages are stored.

   The pages are stored in the **/usr/local/apache2/htdocs** folder.

   Check the **index.html** page contents.

```
root@e91768d643dd:/usr/local/apache2# cat /usr/local/apache2/htdocs/index.html
```

   The expected output is shown below:

```
<html><body><h1>It works!</h1></body></html>
```

3.4. Change the **index.html** page to contain the text **Hello World**, without HTML tags.

From the Bash session, run the following command:

```
root@e91768d643dd:/usr/local/apache2# echo "Hello World" > \
 /usr/local/apache2/htdocs/index.html
```

3.5. Check if the updated web page contents can be accessed by refreshing the open web browser.

4. Grade your work.

Open a terminal on **workstation** and 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
[student@workstation ~]$ docker rm httpd-basic
```

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

```
[student@workstation ~]$ docker rmi httpd:2.4
```

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, supporting 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 needs to 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.

CHAPTER 3

MANAGING CONTAINERS

Overview	
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.

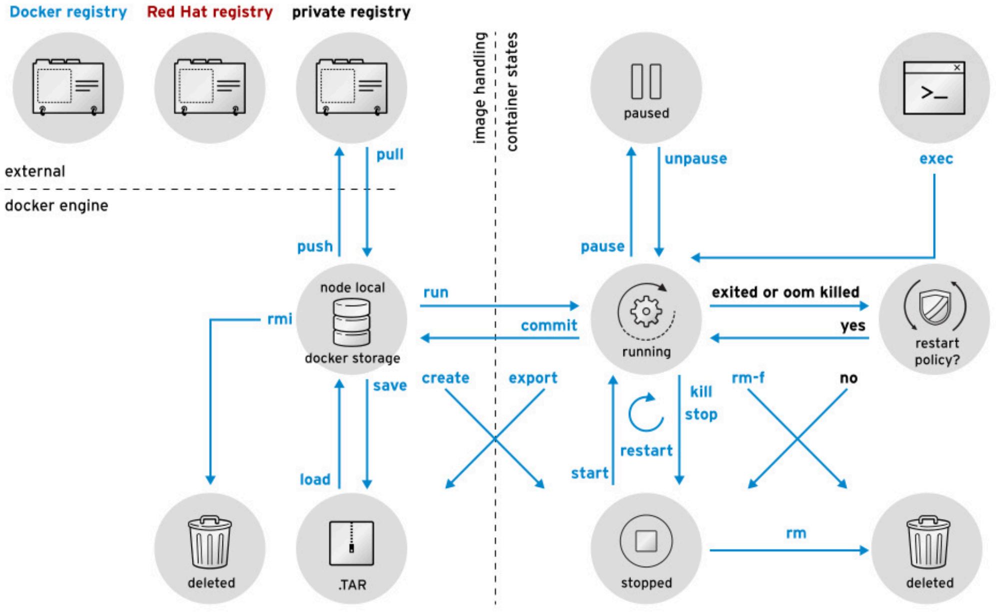


Figure 3.1: Docker client action verbs

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.

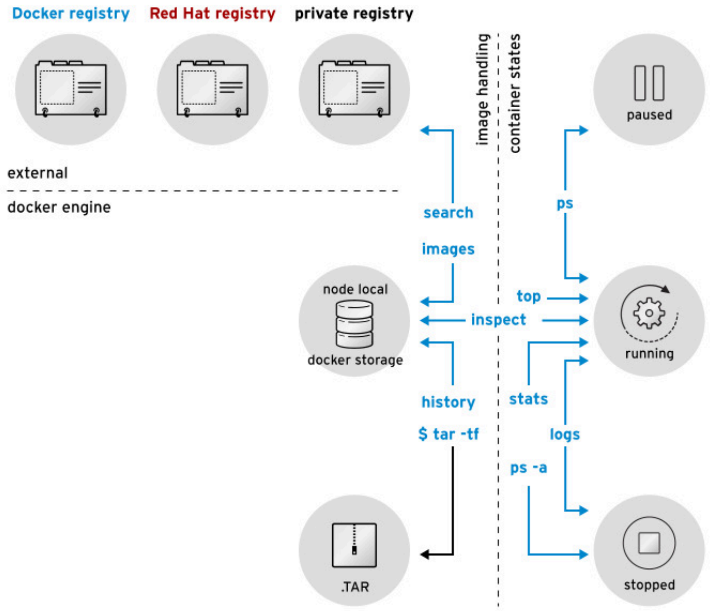


Figure 3.2: Docker client query verbs

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 infrastructure.lab.example.com:5000/rhscl/httpd-24-rhel7 ...
latest: Pulling from infrastructure.lab.example.com:5000/rhscl/httpd-24-rhel7
3436c67883ad: Pull complete
c85416a3d375: Pull complete
aa724488ad02: Pull complete
b70312066a34: Pull complete
Digest: sha256:6c92ae9f2cd45923a6e8013282028263529a7d29cd1a34774abf3ee5614fec5e
...
[Thu Mar 30 13:58:16.375095 2017] [core:notice] [pid 1] AH00094: Command line: 'httpd -D
 FOREGROUND'
$ ^C
```

Whatever output **docker run** shows is generated by the process inside the container, which is just a regular process from the host OS perspective. Killing that process stops the container. 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

 Up 30 seconds           80/tcp               focused_fermat❷
```

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-httpd-container do180/httpd
```

Important

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-httpd-container -d do180/httpd
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/httpd 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-httpd-container -it do180/httpd /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-httpd-container cat /etc/hostname
7ed6e671a600
```

Demonstration: Creating Containers

1. Open a terminal window from the workstation VM (**Applications** > **Favorites** > **Terminal**) and run the following command to prepare files used by this demonstration.

   ```
   [student@workstation ~]$ demo create-containers setup
   ```

2. Run the following command:

   ```
   [student@workstation ~]$ docker run --name demo-container rhel7.3 \
   dd if=/dev/zero of=/dev/null
   ```

   This command downloads the official Red Hat Enterprise Linux 7.3 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.

3. 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.

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

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

   This is the best practice for stopping containers.

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

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

6. Start a new container without specifying a name:

```
[student@workstation ~]$ docker run rhel7.3 dd if=/dev/zero of=/dev/null
```

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

7. 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        rhel7.3             "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.

8. Stop the container with the generated name:

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

9. 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 rhel7.3 \
dd if=/dev/zero of=/dev/null
```

10. Stop the container:

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

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

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

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

12. Verify that the container is not running:

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

13. 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 rhel7.3 \
/bin/bash
```

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

14. Exit the Bash shell from the container:

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

15. 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
```

16. 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>
```

17. Remove the rhel7.3 container image:

```
[student@workstation ~]$ docker rmi rhel7.3
```

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
77d4b7b8ed1f❶       do180/httpd❷        "httpd -D FOREGROUND"❸     15 hours ago ❹
 Up 15 hours ❺       80/tcp❻             my-httpd-container ❼
```

❶ Each container, when created, gets a **container ID**, which is a hexadecimal number and looks like an image ID, but is actually unrelated.
❷ Container image that was used to start the container.
❸ Command that was executed when the container started.
❹ Date and time the container was started.
❺ Total container uptime, if still running, or time since terminated.
❻ Ports that were exposed by the container or the port forwards, if configured.
❼ 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-httpd-container
```

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

```
$ docker stop my-httpd-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-httpd-container
```

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

```
$ docker kill -s SIGKILL my-httpd-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-httpd-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-httpd-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. Open a terminal window from the workstation VM (**Applications** > **Favorites** > **Terminal**) and run the following command to prepare files used by this demonstration.

   ```
   [student@workstation ~]$ demo manage-containers setup
   ```

2. Run the following command:

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

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

3. List all running containers:

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

4. Stop the container with the following command:

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

5. Verify that the container is not running:

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

6. Run a new container with the same name:

   ```
   [student@workstation ~]$ docker run --name demo-container -d rhscl/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.

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

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

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

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

9. 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-httpd
   ```

10. 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-httpd
    ```

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

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

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

12. 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
```

13. 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
```

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

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

15. 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-httpd | grep IPAddress
[student@workstation ~]$ curl IP:8080/do180.html
```

16. Stop the **HTTP** container:

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

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

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

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

```
[student@workstation ~]$ docker inspect demo-2-httpd | 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.

19. 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-httpd
```

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

20. 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-httpd** container:

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

21. 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)
    ```

22. Verify that all containers were removed:

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

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

    ```
    [student@workstation ~]$ docker rmi rhscl/httpd-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 5.6 container image (**rhscl/mysql-56-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-56-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. Specify each variable using the **-e** parameter.

   ```
   [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-56-rhel7
   ```

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

```
[student@workstation ~]$ docker ps
CONTAINER ID        IMAGE                   COMMAND                  CREATED
    STATUS              PORTS                NAMES
5cd89eca81dd        rhscl/mysql-56-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));
```

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

```
MySQL [items]> insert into Projects (id, name, code) values (1,'DevOps','DO180');
```

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 rhscl/mysql-56-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              PORTS               NAMES
8b2c0ee86419        rhscl/mysql-56-rhel7   "container-entrypoint"   4 minutes ago
    Up 4 minutes        3306/tcp            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 rhscl/mysql-56-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.

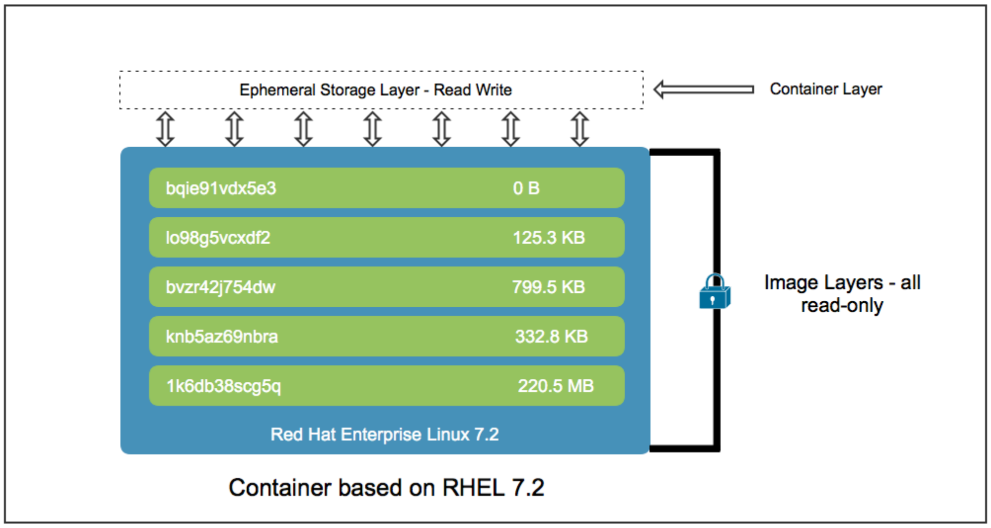

Ephemeral Storage Layer - Read Write
Container Layer
bqie91vdx5e3 0 B
lo98g5vcxdf2 125.3 KB
bvzr42j754dw 799.5 KB
knb5az69nbra 332.8 KB
1k6db38scg5q 220.5 MB
Red Hat Enterprise Linux 7.2
Image Layers - all read-only
Container based on RHEL 7.2

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 will not be 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:

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

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

- 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
  ```

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

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

Of course, 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 5.6 image (**`rhscl/mysql-56-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** > **Utilities** > **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 rhscl/mysql-56-rhel7
   ```

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

```
[student@workstation ~]$ docker run --name persist-mysqlddb \
-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-56-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
8d6acfaa55a5   rhscl/mysql-56-rhel7   "container-entrypoint"   11 seconds ago
 9 seconds ago    3306/tcp            persist-mysqlddb
```

3. Verify that the **`/var/local/mysql`** directory contains an **`items`** directory:

```
[student@workstation ~]$ ls -l /var/local/mysql
total 28688
-rw-rw----. 1 27 27        2 Apr 19 16:52 10ddcb9edec5.pid
-rw-rw----. 1 27 27       56 Apr 19 16:52 auto.cnf
-rw-rw----. 1 27 27 12582912 Apr 19 16:52 ibdata1
-rw-rw----. 1 27 27  8388608 Apr 19 16:52 ib_logfile0
-rw-rw----. 1 27 27  8388608 Apr 19 16:52 ib_logfile1
drwx------. 2 27 27       20 Apr 19 16:52 items
drwx------. 2 27 27     4096 Apr 19 16:52 mysql
drwx------. 2 27 27     4096 Apr 19 16:52 performance_schema
```

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-mysqlddb
```

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

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

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

```
[student@workstation ~]$ docker rmi rhscl/mysql-56-rhel7
```

5.4. Delete the volume directory created earlier in the exercise b y 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.

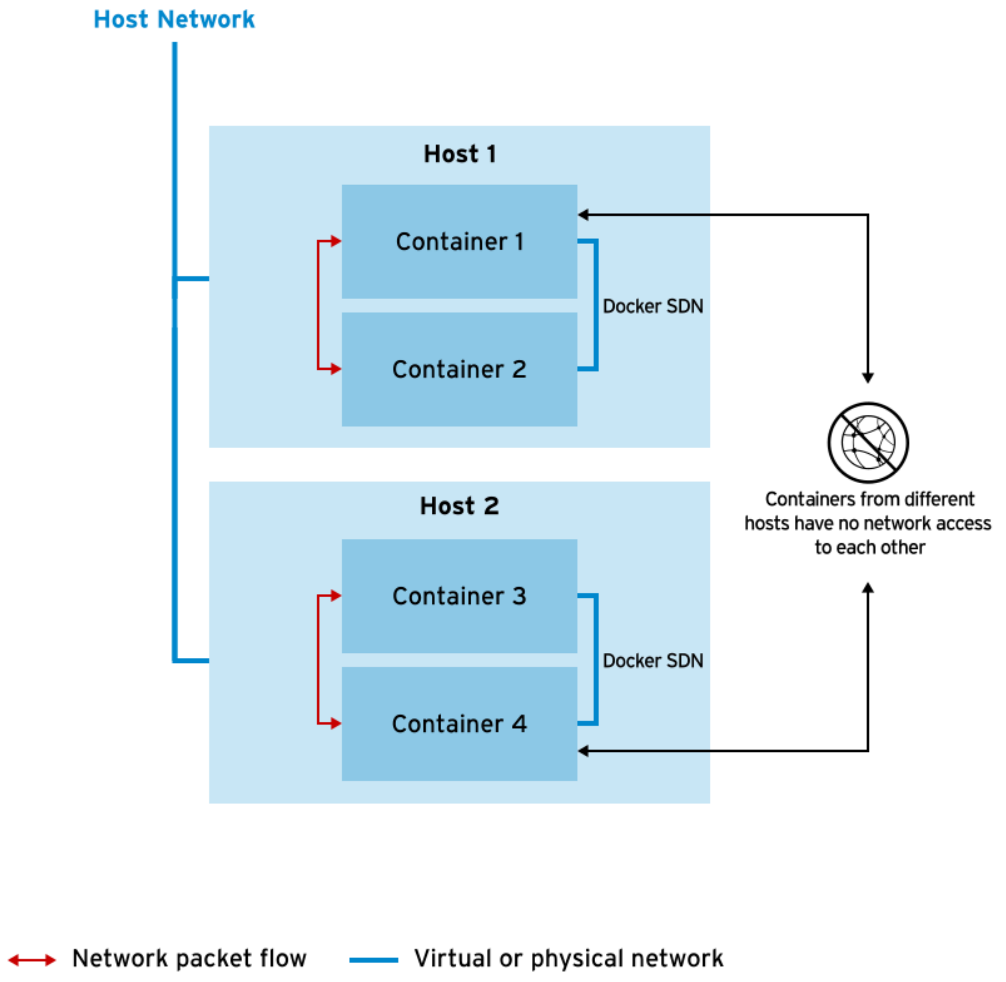


Figure 3.4: Basic Linux containers networking

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. Also, all containers that connect to the same bridge on the same host can talk to 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, 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.

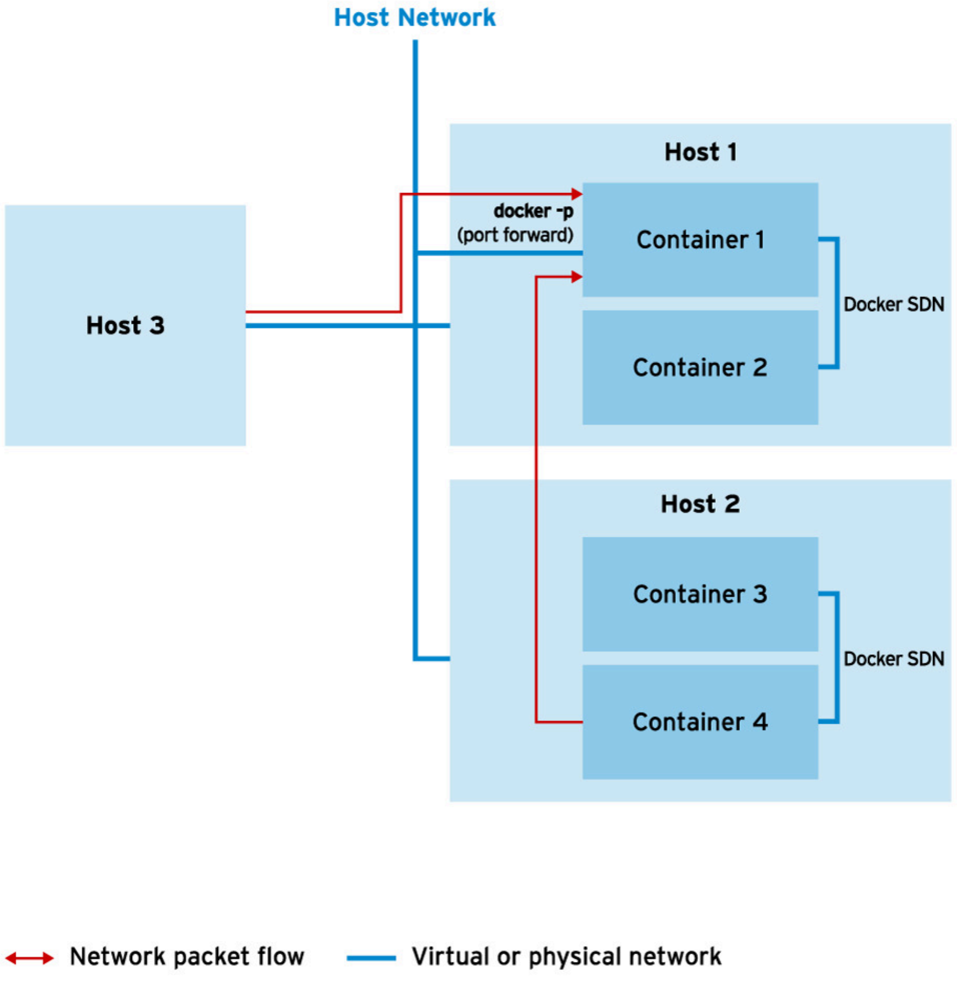


Figure 3.5: Allowing external accesses to Linux containers

It is also possible to determine a specific 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 5.6 image (**`rhscl/mysql-56-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 \
rhscl/mysql-56-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-56-rhel7   "container-entrypoint"   4 seconds ago    Up
 2 seconds     0.0.0.0:13306->3306/tcp   mysqlddb-port
```

3. Load the database:

```
[student@workstation ~]$ mysql -uuser1 -h 127.0.0.1 -pmypa55 \
-P13306 items < /home/student/DO180/labs/load-mysqldb/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 |      |
|  2 | Buy groceries     |   X  |
+----+-------------------+------+
```

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 mysqldb-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 mysqldb-port \
/opt/rh/rh-mysql56/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 rhscl/mysql-56-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

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/DO180/labs/work-containers**
Application URL:	NA
Resources:	RHSCL MySQL 5.6 image (**rhscl/mysql-56-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.

Before you begin

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-56-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/DO180/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.

> ## Warning
>
> 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-56-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 row in the **`Item`** table.

8.1. Connect to the MySQL database.

8.2. Insert the new 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.

This concludes the lab.

Solution

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/DO180/labs/work-containers`**
Application URL:	NA
Resources:	RHSCL MySQL 5.6 image (**`rhscl/mysql-56-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.

Before you begin

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-56-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-56-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
6l6azfaa55x8     rhscl/mysql-56-rhel7   "container-entrypoint"   11 seconds ago
  9 seconds ago    3306/tcp            mysql-1
```

3. Load the **`items`** database using the **`/home/student/DO180/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/DO180/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"
```

4. Stop the container gracefully.

Warning

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
```

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-56-rhel7`**;
   - **Port forward**: yes, from host port 13306 to container port 3306;
   - **Environment variables**:
     - **`MYSQL_USER`**: **`user1`**;
     - **`MYSQL_PASSWORD`**: **`mypa55`**;
     - **`MYSQL_DATABASE`**: **`items`**;
     - **`MYSQL_ROOT_PASSWORD`**: **`r00tpa55`**

5.1. Create and start the container.

```
[student@workstation ~]$ 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-56-rhel7
```

5.2. 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
281c0e2790e5     rhscl/mysql-56-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;
   ```

   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 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 row.

   ```
   MySQL[items]> insert into Item (description, done) values ('Walk the dogs',
    true);
   ```

   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 rhscl/mysql-56-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
    ```

    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

Overview	
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.

- Work with 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.

Unfortunately, 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 to a host, and all **docker** daemons from a development team should add a new registry to make it searchable.

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-registry❶
 infrastructure.lab.example.com:5000❷'
```

❶ The **--add-registry** parameter requires the registry FQDN host and port.

❷ 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:

```
INSECURE_REGISTRY='--insecure-registry❶ infrastructure.lab.example.com:5000❷'
```

❶ The **--insecure-registry** parameter requires the registry FQDN host and port.

❷ The FQDN host and port number where the **docker-registry** service is running.

Accessing Registries

A container image registry is accessed via the **docker** daemon service from a **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 Public 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.

For example, to run a **search** command using the RESTful API, the following HTTP request can be sent to the registry:

```
GET /v1/search?q=<term>
```

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 /v1/search?q=<term>&n=<number>
```

For example, to get the list of images from Docker Hub called **mysql**, the following **curl** command can be used:

```
# curl https://registry.hub.docker.com/v1/search?q=mysql&n=30
```

```
{
    "num_pages": 101,
    "num_results": 2511,
    "results":
        [
            {"is_automated": false,
             "name": "mysql",
             "is_trusted": false,
             "is_official": true,
             "star_count": 1604,
             "description": "MySQL is a widely used, open-source relational database
 management system (RDBMS)."
            },
 ... Output omitted
}
```

Searching for Image Tags in Public Registries

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

```
GET /v1/repositories/<imageName>/tags
```

To list the tags from the official MySQL image container registry, use the following **curl** command:

```
# curl https://registry.hub.docker.com/v1/repositories/mysql/tags
```

Searching for Images in Private Registries

Public registries, such as **registry.access.redhat.com**, support the **search** verb from docker. However, for private registries, such as the one provided by this classroom, do not provide the search facility compatible with docker, because it is running the version 2 release of the docker registry, whereas the public registries are using the version 1 registry.

To search in private registries, using the latest API version, 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**. To get all the images available at a private registry, use the following syntax:

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

For example, to get the list of all images available at **infrastructure.lab.example.com:5000**:

```
# docker-registry-cli infrastructure.lab.example.com:5000 list all
```

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

```
# docker-registry-cli infrastructure.lab.example.com:5000 search mysql
```

Pulling Images

To pull container images from a registry, the **docker** command line supports the **pull** verb. The verb syntax is:

```
# 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. **docker pull** 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. Also, 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     VIRTUAL 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. On the other hand, 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. For example, to pull an image with the tag **5.5** from **mysql**, use the following command:

```
# docker pull mysql:5.5
```

To start a new container based on the **mysql:5.5** image, use the following command:

```
# docker run mysql:5.5
```

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 certified container images website
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. What is the command 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 will be the output from 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

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. What is the command 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 will be the output from 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 **`commit`**.
- Identify the changed artifacts in a container.
- Manage image tags for distribution purposes.

Introduction

Suppose a developer finished testing a custom container in his machine and needs to transfer this container image to another host, for another developer to use it, or to a production server. There are two ways this could be accomplished:

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 will be shown later in this chapter (**`docker commit`**), but the preferred way to do so (**`Dockerfiles`**) will be the subject of the following chapters.

Saving and Loading Images

An existing image from the Docker cache can be saved to a **`tar`** file using the **`docker save`** command. The generated file is not just a regular **`tar`** file; it contains image metadata and preserves original image layers, so the original image can be later re-created exactly as it was.

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

```
# 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 Red Hat Software Collections is saved to the file **`mysql.tar`**:

```
# docker save -o mysql.tar registry.access.redhat.com/rhscl/mysql-56-rhel7
```

A tar file 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 will fail.

Following the previous **docker save** example, the image may be restored to the Docker cache using the following command:

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

> Note
>
> To save disk space, the file generated by the **save** verb can be gzipped. The **load** verb will automatically **gunzip** the file before importing it to the daemon's cache directory.

Publishing an Image to a Registry

To push an image to the registry, it must be stored in the **docker**'s cache, and it should be tagged for identification purposes. To tag an image, use the **tag** subcommand:

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

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

```
# docker tag nginx nginx
```

To push the image to the registry, run the following command:

```
# docker push nginx
```

Deleting an Image

Any image downloaded to the Docker cache is kept there even if no containers are using it. However, images can become outdated, and should be subsequently replaced.

> Note
>
> Any updates to images in a registry are *not* automatically updated in the daemon's cache. The image must be removed and then pulled again to guarantee that the cache has all updates made to the registry.

To delete an image from the cache, use 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** to force the removal of an image. An image can be referenced using its name or its ID for removal purposes.

The same image can be shared among multiple tags, and to use the **rmi** verb with the image ID will fail. To avoid a tag-by-tag removal for an image, the simplest approach would be using 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)
```

This 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 will not be deleted, but this does *not* prevent any unused images from being removed.

Modifying Images

Ideally, all container images should be built using a **Dockerfile** to create a clean and slim set of image layers, without log files, temporary files, or other artifacts created by the container customization. Despite these recommendations, some container images may be provided as they are, without any **Dockerfile** available. 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. **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 a running container for OS management purposes.

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 the author responsible for the container image creation.
--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, **`docker`** client has a verb called **`diff`** to identify the changes made to a container. The **`diff`** 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 marked with an **`A`**, and any changed file is marked 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 extra work to manage and deploy the images to the correct locations.

Container image registries support the tag concept so that you can 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 will be used by a container image.

Usually, the tags are used by container developers to distinguish between multiple versions of the same software, such as the one observed for MySQL container image documentation.

Note

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 (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 that was locally stored to the docker daemon. The following argument refers to alternative names for the image that are stored locally. If no tag is provided, the **`latest`** tag will be considered. For example, to tag an image, the following command may be used:

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

The **`mysql-custom`** option is the image name that is stored in the **`docker`** daemon's cache.

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

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

Removing Tags from the Image

To associate multiple tags with a single image, use the **`docker tag`** command. Tags can be removed using the **`docker rmi`** command mentioned previously. Therefore, to delete a specific image tag from the daemon, run the following command:

```
# 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.

Tagging Practices

Normally, the **`latest`** tag is automatically added by **`docker`** if nothing is provided, because it is considered to be the image's latest build. However, this may not be true depending on how the tags are used. For example, most open source projects consider "latest" as the most recent release, not the latest build.

Moreover, multiple tags are provided to minimize the need to recall 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 to simplify 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 lab, you will create a custom Apache container image using the **docker commit** command.

Resources	
Files:	N/A
Application URL:	**http://127.0.0.1:8180/do180.html**, **http://127.0.0.1:8280/do180.html**
Resources:	CentOS httpd image (**centos/httpd**)

Outcomes

You should be able to create a custom container image.

Before you begin

To verify that the **workstation** has the docker daemon running, open a terminal and run the following command:

```
[student@workstation ~]$ lab container-images-manipulating setup
```

Steps

1. Open a terminal on **workstation** (**Applications** > **Utilities** > **Terminal**) and start a container from the **centos/httpd** image with the following command:

```
[student@workstation ~]$ docker run -d --name official-httpd -p 8180:80 \
centos/httpd
```

2. Create a new HTML page in the **official-httpd** container.

   2.1. Access the container **bash** shell:

```
[student@workstation ~]$ docker exec -it official-httpd /bin/bash
```

   2.2. Create the HTML page:

```
[root@00fd8d4d1846 /]# echo "DO180 Page" > /var/www/html/do180.html
```

   2.3. Exit the **bash** shell:

```
[root@00fd8d4d1846 /]# exit
```

   2.4. Test if the page is reachable:

```
[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:

```
[student@workstation ~]$ docker diff official-httpd
```

The expected output is similar to:

```
...
C /run
C /run/httpd
...
A /var/www/html/do180.html
...
```

The previous output lists the directories and files that were changed or added to the **official-httpd** container. Remember that these changes are only for this container.

4. It is possible to create a new image with the changes created by the previous container. One way is by saving the container to a tar file.

4.1. Stop the **official-httpd** container:

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

4.2. Commit the changes to a new container image:

```
[student@workstation ~]$ docker commit -a 'Your Name' \
-m 'Added do180.html page' official-httpd
sha256:46f0349b897f89843c2c15b429c2eab29358887421d97f5e35a6b7fa35576888
```

4.3. List the available container images:

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

The expected output is similar to the following:

```
REPOSITORY                  TAG                 IMAGE ID
<none>                      <none>              46f0349b897f
infrastructure... /httpd    latest              77946d4aa0c2
... output omitted...
```

Compare the output to that from the previous step to see which image was created by **docker commit**. It is the one created more recently and will be the first one listed.

4.4. The new container image has neither a name (REPOSITORY column) nor a tag. Use the following command to add this information:

```
[student@workstation ~]$ docker tag 46f0349b897f do180/custom-httpd
```

Note

The *46f0349b897f* container image ID is the truncated version of the ID returned by the previous step.

4.5. List the available container images again to confirm 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               IMAGE ID
do180/custom-httpd          latest            34145b0ca3e4
infrastructure... /httpd    latest            77946d4aa0c2
... output omitted...
```

4.6.

Note

Additional images may be listed, but they are not relevant to this exercise.

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

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

```
[student@workstation ~]$ docker tag do180/custom-httpd \
infrastructure.lab.example.com:5000/do180/custom-httpd:v1.0
```

5.2. Ensure that the new name was added to the cache:

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

The expected output is similar to the following:

```
REPOSITORY                                             TAG    ...
do180/custom-httpd                                     latest ...
infrastructure.lab.example.com:5000/do180/custom-httpd v1.0   ...
infrastructure.lab.example.com:5000/centos/HTTP        latest ...
```

5.3. Push the image to the private registry on the infrastructure VM:

```
[student@workstation ~]$ docker push \
infrastructure.lab.example.com:5000/do180/custom-httpd:v1.0
```

Note

Each student gets their own private registries on the infrastructure VM so there is no chance students will interfere with each other's work. These private registries require no authentication, but most public registries require that you log in before the push operation.

5.4. Verify that the image can be found by a search in the classroom registry:

```
[student@workstation ~]$ docker-registry-cli \
infrastructure.lab.example.com:5000 search custom-httpd
```

The expected output is as follows:

```
available options:-

-----------
1) Name: do180/custom-httpd
Tags: v1.0

1 images found !
```

6. Test the image that got pushed to the registry.

6.1. Start a new container using the recently uploaded image to the docker registry.

From the terminal, run the following command:

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

6.2. Check that the do180 page is accessible.

From the terminal, run the following command:

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

The DO180 page created in the previous steps is displayed.

7. Grade your work. Run the following command from a terminal:

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

8. Delete the containers and images created by this lab:

8.1. Stop the containers that are running:

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

8.2. Remove the containers from the cache:

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

8.3. Remove the exported container image:

```
[student@workstation ~]$ docker rmi do180/custom-httpd
```

Remove the committed container image:

```
[student@workstation ~]$ docker rmi \
infrastructure.lab.example.com:5000/do180/custom-httpd:v1.0
```

8.4. Remove the `centos/httpd` container image:

```
[student@workstation ~]$ docker rmi centos/httpd
```

8.5. Check the reset was executed with success. Run the following from a terminal:

```
[student@workstation ~]$ lab container-images-manipulating gradeclean
```

This concludes the guided exercise.

Lab: Managing Images

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

Resources	
Files:	N/A
Application URL:	**http://127.0.0.1:8380**, **http://127.0.0.1:8280**
Resources	Official nginx image (**nginx**)

Outcomes

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

Before you begin

To verify the Docker daemon is running on workstation, run the following command:

```
[student@workstation ~]$ lab container-images-lab setup
```

Steps

1. Open a terminal on the workstation VM (**Applications** > **Utilities** > **Terminal**) and pull the official **nginx** container image.

   1.1. Search the **nginx** container image:

   1.2. Pull the **nginx** container image:

> ## Note
>
> The image is pulled from the classroom registry, but the container image is the same one that is in the **docker.io** registry. This is to save time in the download process.

   1.3. Check that the container image is available in the cache:

2. To start a new container, forward the server port 80 to port 8080 from the host using the **-p** option. You can use syntax similar to the following:

```
# DO NOT RUN THIS COMMAND
$ docker run --name some-nginx -d -p 8080:80 some-content-nginx
```

3. Create a new container with the following characteristics:

   - **Name**: official-nginx
   - **Run as daemon**: yes
   - **Container image**: **nginx**
   - **Port forward**: yes, from host port 8380 to container port 80

4. Replace the current **index.html** content with the following content: "DO180 Page". Use the container documentation to find the folder that contains the HTML pages.

   4.1. Access the container **bash** shell.

   4.2. Replace the **index.html** file with the "DO180 Page"' string.

   4.3. Exit the container.

   4.4. Test if the **index.html** content was updated.

5. Stop the **official-nginx** container and commit the changes to create a new container image. Set the name of this container image to **do180/mynginx** and tag it as **v1.0**.

   5.1. Stop the **official-nginx** container.

   5.2. Commit the changes to a new container image.

   5.3. List the available container images to get the ID from the new container.

   5.4. Set the name and tag the new container image.

6. Create a new container with the following characteristics:

   - **Name**: my-nginx
   - **Run as daemon**: yes
   - **Container image**: **do180/mynginx:v1.0**
   - **Port forward**: yes, from host port 8280 to container port 80

7. Test if the **index.html** page is available with the custom content.

8. Verify that the lab was correctly set up. Run the following command from a terminal:

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

9. Delete the containers and resources created by this lab:

   9.1. Stop the **my-nginx** container:

   9.2. Remove the containers from the cache:

   9.3. Remove the container images:

   9.4. Verify that the lab was correctly cleaned up. Run the following from a terminal:

```
[student@workstation ~]$ lab container-images-lab gradeclean
```

This concludes the laboratory.

Solution

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

Resources	
Files:	N/A
Application URL:	**`http://127.0.0.1:8380`**, **`http://127.0.0.1:8280`**
Resources	Official nginx image (**nginx**)

Outcomes

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

Before you begin

To verify the Docker daemon is running on workstation, run the following command:

```
[student@workstation ~]$ lab container-images-lab setup
```

Steps

1. Open a terminal on the workstation VM (**Applications** > **Utilities** > **Terminal**) and pull the official **nginx** container image.

    1.1. Search the **nginx** container image:

    ```
    [student@workstation ~]$ docker-registry-cli \
    infrastructure.lab.example.com:5000 search nginx
    ```

    The following output is displayed:

    ```
    available options:-

    -----------
    1) Name: nginx
    Tags: latest

    1 images found !
    ```

    1.2. Pull the **nginx** container image:

    ```
    [student@workstation ~]$ docker pull nginx
    ```

    ## Note

    The image is pulled from the classroom registry, but the container image is the same one that is in the **docker.io** registry. This is to save time in the download process.

    1.3. Check that the container image is available in the cache:

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

This command produces output similar to the following:

```
REPOSITORY                                  TAG...
...
infrastructure.lab.example.com:5000/nginx    latest...
...
```

2. To start a new container, forward the server port 80 to port 8080 from the host using the **-p** option. You can use syntax similar to the following:

```
# DO NOT RUN THIS COMMAND
$ docker run --name some-nginx -d -p 8080:80 some-content-nginx
```

3. Create a new container with the following characteristics:

   - **Name**: official-nginx
   - **Run as daemon**: yes
   - **Container image**: `nginx`
   - **Port forward**: yes, from host port 8380 to container port 80

```
[student@workstation ~]$ docker run --name official-nginx -d -p 8380:80 nginx
```

4. Replace the current **`index.html`** content with the following content: "DO180 Page". Use the container documentation to find the folder that contains the HTML pages.

   4.1. Access the container **bash** shell.

```
[student@workstation ~]$ docker exec -it official-nginx /bin/bash
```

   4.2. Replace the **`index.html`** file with the "DO180 Page"' string.

```
root@cf6ccf453cde:/# echo 'DO180 Page' > /usr/share/nginx/html/index.html
```

   4.3. Exit the container.

```
root@cf6ccf453cde:/# exit
```

   4.4. Test if the **`index.html`** content was updated.

```
[student@workstation ~]$ curl 127.0.0.1:8380
```

5. Stop the **`official-nginx`** container and commit the changes to create a new container image. Set the name of this container image to **`do180/mynginx`** and tag it as **v1.0**.

5.1. Stop the **`official-nginx`** container.

```
[student@workstation ~]$ docker stop official-nginx
```

5.2. Commit the changes to a new container image.

```
[student@workstation ~]$ docker commit -a 'Your Name' \
-m 'Changed index.html page' official-nginx
sha256:28bb1696c4cf61b5258e56a79ce71fc5a2ed055b4875f0848eb403e7c16a8058
```

Note the container ID that is returned. This ID will be necessary to the tag the container image.

5.3. List the available container images to get the ID from the new container.

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

You should see the following output:

```
REPOSITORY                                      TAG        IMAGE ID     ...
<none>                                          <none>     28bb1696c4cf ...
infrastructure.lab.example.com:5000/nginx       latest     5e69fe4b3c31 ...
```

5.4. Set the name and tag the new container image.

```
[student@workstation ~]$ docker tag 28bb1696c4cf do180/mynginx:v1.0
```

The *28bb1696c4cf* container image ID is the truncated version of the ID returned by *Step 5.2*.

6. Create a new container with the following characteristics:

- **Name**: my-nginx
- **Run as daemon**: yes
- **Container image**: **`do180/mynginx:v1.0`**
- **Port forward**: yes, from host port 8280 to container port 80

```
[student@workstation ~]$ docker run -d --name my-nginx -p 8280:80 \
 do180/mynginx:v1.0
```

7. Test if the **`index.html`** page is available with the custom content.

```
[student@workstation ~]$ curl 127.0.0.1:8280
```

8. Verify that the lab was correctly set up. Run the following command from a terminal:

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

9. Delete the containers and resources created by this lab:

   9.1. Stop the **my-nginx** container:

   ```
   [student@workstation ~]$ docker stop my-nginx
   ```

   9.2. Remove the containers from the cache:

   ```
   [student@workstation ~]$ docker rm my-nginx official-nginx
   ```

   9.3. Remove the container images:

   ```
   [student@workstation ~]$ docker rmi nginx do180/mynginx:v1.0
   ```

   9.4. Verify that the lab was correctly cleaned up. Run the following from a terminal:

   ```
   [student@workstation ~]$ lab container-images-lab gradeclean
   ```

This concludes the laboratory.

Summary

In this chapter, you learned:

- Registries must be used to pull and push container images for internal use (known as a private registry) or for outside consumption (known as a public registry).
    - The Red Hat registry provides tested and certified images at **registry.access.redhat.com**.
    - A **docker** daemon supports extra registries by editing the **/etc/sysconfig/docker** file; customize the **ADD_REGISTRY** line accordingly.
    - For registries without a certificate, **docker** will fail. In order to support such registries, the **INSECURE_REGISTRY** variable in the **/etc/sysconfig/docker** file must be customized.
    - Registries implement a RESTful API to pull, push, and manipulate contents and this API is used by the **docker** daemon or by a tool capable of generating an HTTP request (such as **curl** or a web browser).
    - 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** is the preferred approach.
- The **docker** daemon cache can be used as a staged area to customize and push images to a registry.
- Docker images in the local cache can be tagged using the **docker tag** command.
- 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

Overview	
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. Run a Dockerfile that uses 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 as a child of 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 5.6, which is named **`rh-mysql56`**), 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/rhscl-dockerfiles`**.

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.

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/

RHSCL Dockerfiles on GitHub
https://github.com/sclorg/rhscl-dockerfiles

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 three common valid security concerns a container image must address? (Choose three.)

   a. Whether the image is updated regularly for security issues.
   b. Whether the image requires elevated privileges to run, such as running as the root user.
   c. Whether the image is designed to be secure by default, for example without accepting default user credentials for remote access.
   d. Whether the image is an S2I-compatible builder image.
   e. Whether the Dockerfile for the image is available for customization.

Solution

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 three common valid security concerns a container image must address? (Choose three.)

    a. **Whether the image is updated regularly for security issues.**
    b. **Whether the image requires elevated privileges to run, such as running as the root user.**
    c. **Whether the image is designed to be secure by default, for example without accepting default user credentials for remote access.**
    d. Whether the image is an S2I-compatible builder image.
    e. Whether the Dockerfile for the image is available for customization.

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 ❶
FROM rhel7.3 ❷
LABEL description="This is a custom httpd container image" ❸
MAINTAINER John Doe <jdoe@xyz.com> ❹
RUN yum install -y httpd ❺
EXPOSE 80 ❻
ENV LogLevel "info" ❼
```

```
ADD http://someserver.com/filename.pdf /var/www/html ❽
COPY ./src/ /var/www/html/ ❾
USER apache ❿
ENTRYPOINT ["/usr/sbin/httpd"] ⓫
CMD ["-D", "FOREGROUND"] ⓬
```

❶ Lines that begin with a pound sign (#) are comments.

❷ The new container image will be constructed on the **rhel7.3** container 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.

❸ **LABEL** is responsible for adding generic metadata to an image. A **LABEL** is a simple key/value pair.

❹ **MAINTAINER** is responsible for setting the **Author** field of the generated container image. You can use the **docker inspect** command to view image metadata.

❺ **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**.

❻ **EXPOSE** indicates that the container listens on the specified network ports at runtime. The Docker containerized environment uses this information to interconnect containers using the *linked containers* feature. The **EXPOSE** instruction is only metadata; 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.

❼ **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.

❽ **ADD** copies new files, directories, or remote **URLs** and adds them to the container file system.

❾ **COPY** also copies new files and directories and adds them to the container file system. However, it is not possible to use a **URL**.

❿ **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.

⓫ **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.

⓬ **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 ping the local host:

```
ENTRYPOINT ["/bin/ping", "localhost"]
```

The following example provides the same functionality, with the added benefit of being overwritable when a container is started:

```
ENTRYPOINT ["/bin/ping"]
CMD ["localhost"]
```

When a container is started without providing a parameter, **localhost** is pinged:

```
[student@workstation ~]$ docker run -it do180/rhel
```

If a parameter is provided after the image name in the **docker run** command, however, it overwrites the **CMD** instruction. For example, the following command will ping **redhat.com** instead of **localhost**:

```
[student@workstation demo-basic]$ docker run -it do180/rhel redhat.com
```

As previously mentioned, because the default **ENTRYPOINT** is **/bin/sh -c**, the following instruction also pings **localhost**, without the added benefit of being able to override the parameter at run time.

```
CMD ["ping", "localhost"]
```

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** instruction in a **Dockerfile**:

```
RUN yum --disablerepo=* --enablerepo="rhel-7-server-rpms"
RUN yum update
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 && 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 && \
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" \
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:

```
-bash-4.2# 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. Open a terminal on the workstation VM (**Applications** > **Favorites** > **Terminal**) and run the following command to download the lab files:

   ```
   [student@workstation ~]$ demo simple-container setup
   ```

2. Briefly review the provided **Dockerfile** by opening it in a text editor:

   ```
   [student@workstation ~]$ vim /home/student/DO180/labs/simple-container/Dockerfile
   ```

3. Observe the **FROM** instruction on the first line of the **Dockerfile**:

   ```
   FROM rhel7.3
   ```

   rhel7.3 is the base image from which the container is built and where subsequent instructions in the **Dockerfile** are executed.

4. Observe the **MAINTAINER** instruction:

   ```
   MAINTAINER John Doe <jdoe@abc.com>
   ```

   The **MAINTAINER** instruction generally indicates the author of the **Dockerfile**. It sets the *Author* field of the generated images. You can run the **docker inspect** command on the generated image to view the *Author* field.

5. 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 2017"
```

You can run the **docker inspect** command to view the labels in the generated image.

6. 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"
```

7. In the **ADD** instruction, add the **training.repo** file, which points to the classroom yum repository:

```
ADD training.repo /etc/yum.repos.d/training.repo
```

> ### Note
>
> The **training.repo** file configures **yum** to use the local repository instead of attempting to use subscription manager.

8. Observe the RUN instruction where the **yum update** command is executed, and the *bind-utils* package is installed in the container image:

```
RUN yum update -y && \
yum install -y bind-utils && \
yum clean all
```

The **yum update** command updates the RHEL 7.3 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.

9. Save the **Dockerfile** and run the following commands in the terminal to begin building the new image:

```
[student@workstation ~]$ cd /home/student/DO180/labs/simple-container
[student@workstation simple-container]$ docker build -t do180/rhel .
Sending build context to Docker daemon 3.584 kB
Step 1 : FROM rhel7.3
 ---> 41a4953dbf95
Step 2 : MAINTAINER John Doe <jdoe@abc.com>
 ---> Running in dc51ab993f8d
 ---> 8323ca92801d
Removing intermediate container dc51ab993f8d
Step 3 : LABEL version "1.0" description "This is a simple container image"
 creationDate "31 March 2017"
 ---> Running in ca52442dd68e
 ---> 61a7cf7539b2
```

```
Removing intermediate container ca52442dd68e
Step 4 : ENV VAR1 "hello" VAR2 "world"
 ---> Running in 947711d586f5
 ---> 7608612e25ca
Removing intermediate container 947711d586f5
Step 5 : ADD training.repo /etc/yum.repos.d/training.repo
 ---> 07c4cc769601
Removing intermediate container ee7f9de68ca5
Step 5 : RUN yum update -y && yum install -y bind-utils && yum clean all
 ---> Running in 84d40f3c9a6c
...
Installed:
  bind-utils.x86_64 32:9.9.4-38.el7_3.2

Dependency Installed:
  GeoIP.x86_64 0:1.5.0-11.el7              bind-libs.x86_64 32:9.9.4-38.el7_3.2
  bind-license.noarch 32:9.9.4-38.el7_3.2

Complete!
Loaded plugins: ovl, product-id, search-disabled-repos, subscription-manager
This system is not registered to Red Hat Subscription Management. You can use
 subscription-manager to register.
Cleaning repos: rhel-7-server-extras-rpms rhel-7-server-optional-rpms
              : rhel-7-server-ose-3.4-rpms rhel-server-rhscl-7-rpms rhel_dvd
Cleaning up everything
 ---> f5d4f131191c
Removing intermediate container 84d40f3c9a6c
Successfully built f5d4f131191c
```

10. After the build completes, run the **`docker images`** command. It should produce output similar to the following:

```
[student@workstation simple-container]$ docker images
REPOSITORY      TAG      IMAGE ID       CREATED              SIZE
do180/rhel      latest   f5d4f131191c   About a minute ago   877.1 MB
...
```

11. Inspect the created image for the **`MAINTAINER`** and **`LABEL`** metadata:

```
[student@workstation simple-container]$ docker inspect do180/rhel | grep Author
        "Author": "John Doe \u003cjdoe@abc.com\u003e",
[student@workstation simple-container]$ docker inspect do180/rhel \
  | grep 'version\|description\|creationDate'
                "creationDate": "31 March 2017",
                "description": "This is a simple container image",
                "version": "1.0"
...
```

Note

The **`MAINTAINER`** instruction sets the *Author* field in the image metadata.

12. Execute the following command to run the new image and to open an interactive Bash terminal:

```
[student@workstation simple-container]$ docker run --name simple-container \
        -it do180/rhel /bin/bash
```

13. In the RHEL 7.3 container, verify that the environment variables set in the **`Dockerfile`** are injected into the container:

    ```
    [root@8b1580851134 /]# env | grep 'VAR1\|VAR2'
    VAR1=hello
    VAR2=world
    ```

14. Execute a **`dig`** command to verify that the *bind-utils* package is installed and working correctly:

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

15. Exit from the Bash shell in the container:

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

    Exiting Bash terminates the running container.

16. Remove the **`simple-container`** container:

    ```
    [student@workstation simple-container]$ docker rm simple-container
    ```

17. Remove the **`do180/rhel`** container image:

    ```
    [student@workstation simple-container]$ docker rmi do180/rhel
    ```

18. Remove the intermediate containers 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/3.5/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/DO180/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 7.3 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
```

Steps

1. Create the Apache Dockerfile

    1.1. Open a terminal on workstation (**Applications** > **Utilities** > **Terminal**) and run the following command to create a new Dockerfile:

    ```
    [student@workstation ~]$ vim /home/student/DO180/labs/basic-apache/Dockerfile
    ```

    1.2. Use RHEL 7.3 as a base image by adding the following **`FROM`** instruction at the top of the new Dockerfile:

    ```
    FROM rhel7.3
    ```

    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 rhel7.3

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 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 : FROM rhel7.3
 ---> 41a4953dbf95
Step 2 : MAINTAINER Your Name <youremail>
 ---> Running in c5ea56af133d
 ---> 3888ce170fbd
Removing intermediate container c5ea56af133d
Step 3 : LABEL description "A basic Apache container on RHEL 7"
 ---> Running in acbbc1a76bab
 ---> 5e3af99cb70e
Removing intermediate container acbbc1a76bab
Step 4 : ADD training.repo /etc/yum.repos.d/training.repo
 ---> 860244ece018
Removing intermediate container f13f16e0b5e5
Step 5 : RUN yum -y update && yum install -y httpd && yum clean all
 ---> Running in 9b374ef232f3
...
Installed:
  httpd.x86_64 0:2.4.6-45.el7
...
Complete!
...
Cleaning up everything
 ---> aeaa00f3e683
Removing intermediate container 9b374ef232f3
Step 6 : EXPOSE 80
 ---> Running in 08f4ade2c978
 ---> d52e0a0f8cfe
Removing intermediate container 08f4ade2c978
Step 7 : CMD httpd -D FOREGROUND
 ---> Running in d315804d9795
 ---> 6249f4144f67
Removing intermediate container d315804d9795
Successfully built 6249f4144f67
```

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          VIRTUAL SIZE
do180/apache   latest   6249f4144f67   4 minutes ago    885.8 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 a **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

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 7.3 image that serves a custom **index.html** page.

Resources	
Files:	**/home/student/DO180/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.

Before you begin

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
```

Steps

1. Review the provided Dockerfile stub in the **/home/student/DO180/labs/custom-images/** folder. Edit the **Dockerfile** and ensure that it meets the following specifications:

   - Base image: **rhel7.3**
   - 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** > **Utilities** > **Terminal**) and edit the Dockerfile located in the **/home/student/DO180/labs/custom-images/** folder.

1.2. Set the base image for the Dockerfile to **rhel7.3**.

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. 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 was built.

3. Create a new container 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 container images created in this lab.

8. Remove the intermediate containers images generated by the **docker build** command.

```
[student@workstation simple-container]$ docker rmi -f $(docker images -q)
```

This concludes the lab.

Solution

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 7.3 image that serves a custom **`index.html`** page.

Resources	
Files:	**`/home/student/DO180/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.

Before you begin

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
```

Steps

1. Review the provided Dockerfile stub in the **`/home/student/DO180/labs/custom-images/`** folder. Edit the **`Dockerfile`** and ensure that it meets the following specifications:

   - Base image: **`rhel7.3`**
   - 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** > **Utilities** > **Terminal**) and edit the Dockerfile located in the **`/home/student/DO180/labs/custom-images/`** folder.

   ```
   [student@workstation ~]$ cd /home/student/DO180/labs/custom-images/
   ```

```
[student@workstation custom-images]$ vim Dockerfile
```

1.2. Set the base image for the Dockerfile to **rhel7.3**.

```
FROM rhel7.3
```

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 update -y && \
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. 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 rhel7.3

MAINTAINER Your Name <youremail>

ENV PORT 8080

ADD training.repo /etc/yum.repos.d/training.repo
RUN yum update -y && \
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  5.12 kB
Step 1 : FROM rhel7.3
 ---> 41a4953dbf95
Step 2 : MAINTAINER Your Name <youremail>
 ---> Using cache
 ---> 5d82aa1c07fb
Step 3 : ENV PORT 8080
 ---> Using cache
 ---> 79f6695c630e
Step 4 : RUN yum update -y && yum install -y httpd yum && lean all
 ---> Running in 106fbaa38eac

 Installed:
  httpd.x86_64 0:2.4.6-45.el7
...
Complete!
 ---> c4477302415d
Removing intermediate container 106fbaa38eac
Step 5 : 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 180dd2a70e28
 ---> 6bbced8a9d23
Removing intermediate container 180dd2a70e28
Step 6 : USER apache
 ---> Running in ff13e13e0842
 ---> 8a790c853a32
Removing intermediate container ff13e13e0842
Step 7 : EXPOSE ${PORT}
 ---> Running in 10fcab2529f9
 ---> 6592c276138e
Removing intermediate container 10fcab2529f9
```

```
Step 8 : COPY ./src/ /var/www/html/
 ---> fa4824ed19cf
Removing intermediate container 92433738ae71
Step 9 : CMD httpd -D FOREGROUND
 ---> Running in 10ff53b33796
 ---> e772db759680
Removing intermediate container 10ff53b33796
Successfully built e772db759680
```

2.3. Run the **`docker images`** command to verify that the custom image was built.

```
[student@workstation custom-images]$ docker images
REPOSITORY                                    TAG      IMAGE ID      ...
do180/custom-apache                           latest   e772db759680  ...
infrastructure.lab.example.com:5000/rhel7.3   latest   41a4953dbf95  ...
```

3. Create a new container 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
CONTAINER ID    IMAGE                COMMAND               ...  PORTS
         NAMES
367823e35c4a    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 container images created in this lab.

```
[student@workstation custom-images]$ docker rmi -f do180/custom-apache
```

8. Remove the intermediate containers 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

Overview	
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 OCP. • 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 systems administrators and developers to work with an OCP 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 OCP 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 OCP 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 which platform is 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 OCP 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 OCP 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://ocp.lab.example.com:8443 \
  -u developer -p developer
```

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, OCP saves an authorization token in the user 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://ocp.lab.example.com:8443

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://ocp.lab.example.com:8443"
```

It is possible to log in as the OCP *cluster administrator* from any master node without a password by using the **system:admin** argument for the **-u** option. This gives you full privileges over all the operations and resources in the OCP instance and should be used with care.

```
[root@ocp ~]# oc login -u system:admin
```

If running the all-in-one OCP cluster, the user who started the cluster can also log in as the **system:admin** user without a password, because the **oc cluster up** command saves an cluster administrator authentication token in the user home folder.

References

Further information is available in the OpenShift Container Platform documentation:

> CLI Reference
> https://access.redhat.com/documentation/en-us/openshift_container_platform/3.5/html/cli_reference/

Quiz: OpenShift 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 OCP 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 OCP 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 OCP master is accessible to all users without any authentication or authorization. Any user can issue any command from the **oc** client.
   b. OCP requires the user to authenticate with a user name and password prior to issuing any authorized command.
   c. For security reasons, OCP requires you to authenticate with a user name and password every single time you issue a command.
   d. OCP generates an authentication token upon successful login, and subsequent requests need not provide credentials again.

3. As the root user, which command below allows you to log in as an OCP cluster administrator on any master node?

   a. **`oc login -u cluster:admin`**
   b. **`oc login -u system -p admin`**
   c. **`oc login -u system:admin`**
   d. **`oc login system:admin -p`**
   e. **`oc login -u root:admin -p`**

Solution

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 OCP 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 OCP 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 OCP master is accessible to all users without any authentication or authorization. Any user can issue any command from the **oc** client.
    b. OCP requires the user to authenticate with a user name and password prior to issuing any authorized command.
    c. For security reasons, OCP requires you to authenticate with a user name and password every single time you issue a command.
    d. OCP generates an authentication token upon successful login, and subsequent requests need not provide credentials again.

3. As the root user, which command below allows you to log in as an OCP cluster administrator on any master node?

    a. `oc login -u cluster:admin`
    b. `oc login -u system -p admin`
    c. `oc login -u system:admin`
    d. `oc login system:admin -p`
    e. `oc login -u root:admin -p`

Creating Kubernetes Resources

Objectives

After completing this section, students should be able to create standard Kubernetes resources.

OpenShift Container Platform (OCP) Resources

The OpenShift Platform organizes entities in the OCP 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 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 OCP 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 a size, and the OCP 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 OCP 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) running. These pods are created from a template, which is part of the replication controller definition. If pods 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 the Source-to-Image (S2I) process or be Dockerfile-based. 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 OCP 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 be created, but administrators control the resource quotas allocated to projects.

Regardless of the type of resource that the administrator is managing, the OCP 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 OCP.

Pod Resource Definition Syntax

OpenShift Container Platform runs containers inside Kubernetes pods, and to create a pod from a container image, OCP 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 OCP 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: Pod❶
metadata:
  name: wildfly❷
  labels:
    name: wildfly❸
spec:
  containers:
    - resources:
        limits :
          cpu: 0.5
      image: do276/todojee
      name: wildfly
      ports:
        - containerPort: 8080❹
          name: wildfly
      env:❺
        - name: MYSQL_ENV_MYSQL_DATABASE
          value: items
        - name: MYSQL_ENV_MYSQL_USER
          value: user1
        - name: MYSQL_ENV_MYSQL_PASSWORD
          value: mypa55
```

❶ Declares a pod Kubernetes resource type.

❷ A unique name for a pod in Kubernetes that allows administrators to run commands on it.

❸ Creates a label with a key called **name** that can be used to be found by other resources, usually a service, from Kubernetes.

❺ A container-dependent attribute identifying which port from the container is exposed.

❺ 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.

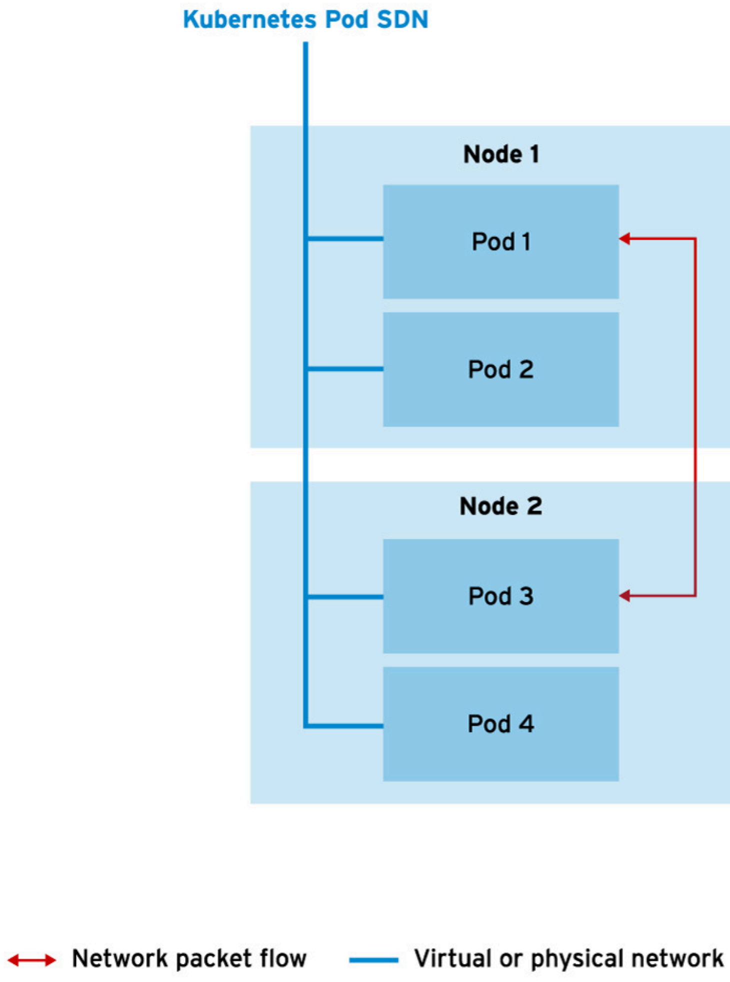


Figure 6.1: Kubernetes basic networking

Services are essential resources to any OCP 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 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.

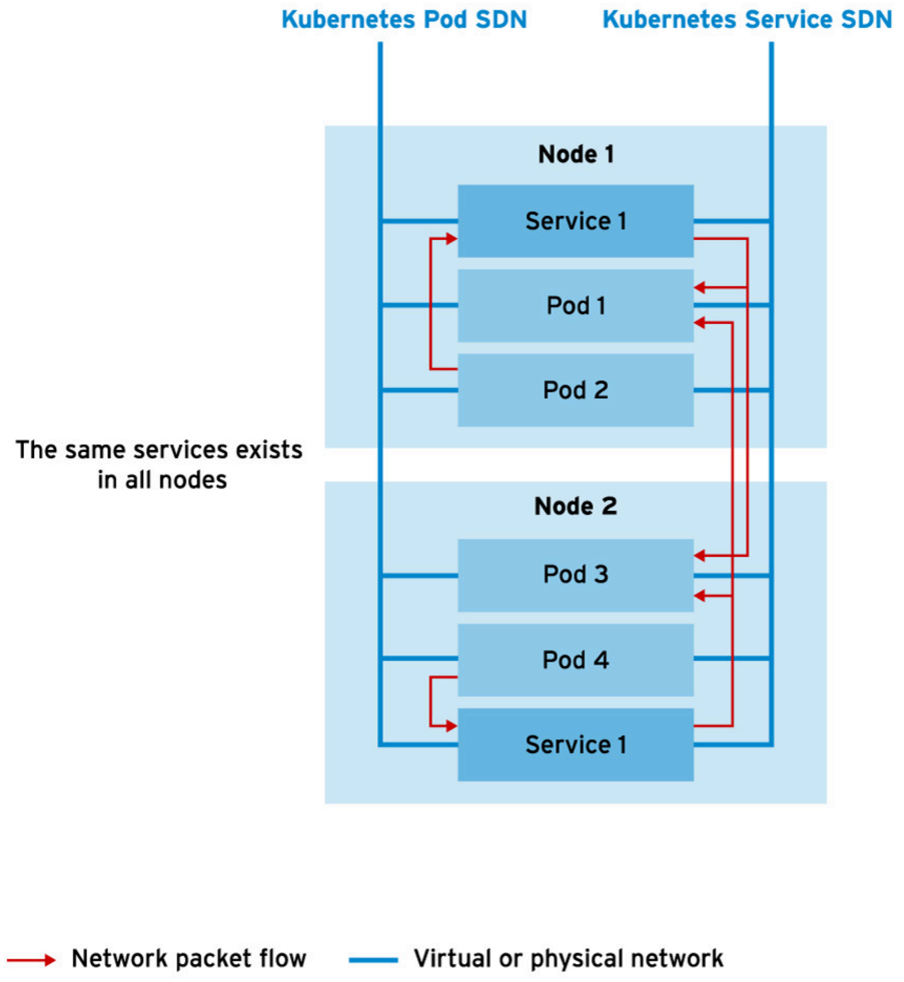


Figure 6.2: Kubernetes services networking

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", ❶
    "apiVersion": "v1",
    "metadata": {
        "name": "quotedb" ❷
    },
    "spec": {
        "ports": [ ❸
            {
                "port": 3306,
                "targetPort": 3306
            }
        ],
        "selector": {
            "name": "mysqldb" ❹
        }
    }
}
```

❶ The kind of Kubernetes resource. In this case, a *Service*.

❷ A unique name for the service.

❸ `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`.

❹ `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 OCP 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 OCP 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 OCP 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 OCP 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. OCP then proxies connections to the service via the public IP address of the node host and the port value set in **nodePort**.

2. OCP Routes: This is the preferred approach in OCP to expose services using a unique URL. You can use the **oc expose** command to expose a service for external access or expose a service from the OCP 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.

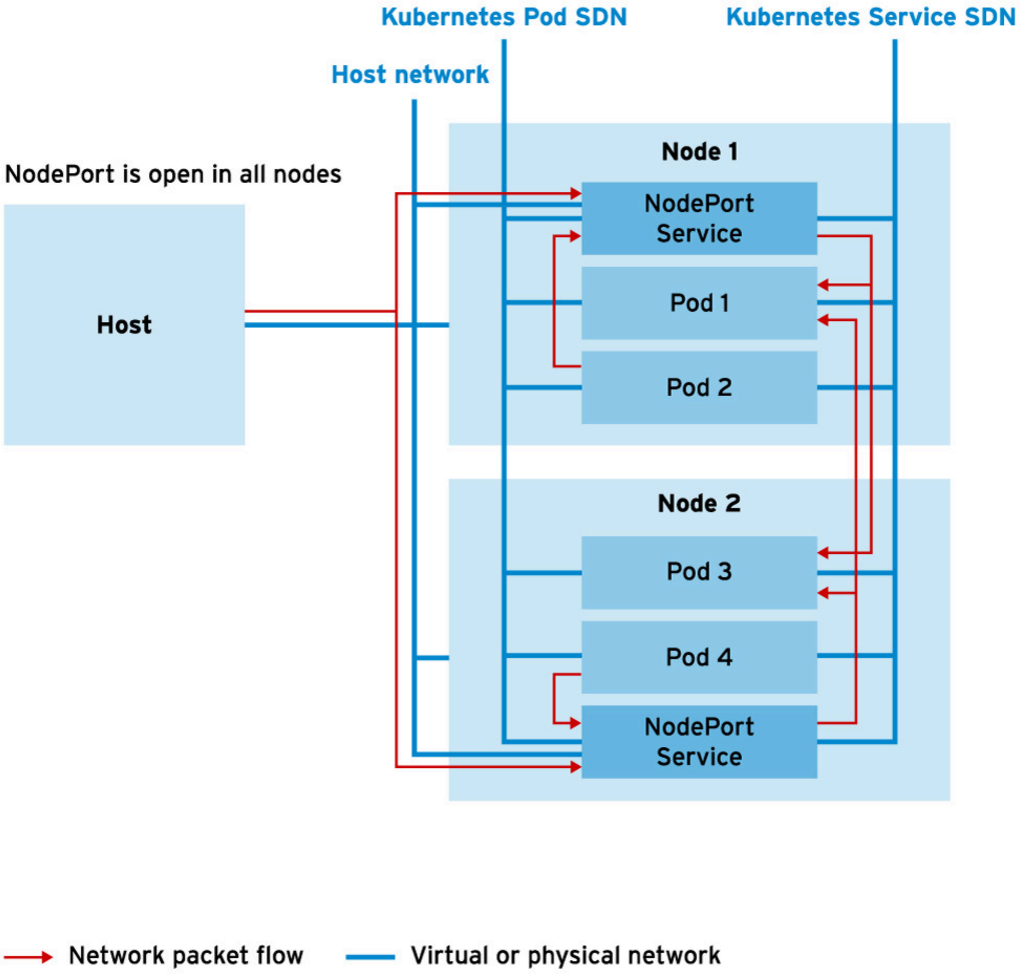


Figure 6.3: Kubernetes NodePort services

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 OCP 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 OCP. The most common options are:

To create an application based on an image from Docker Hub:

```
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:

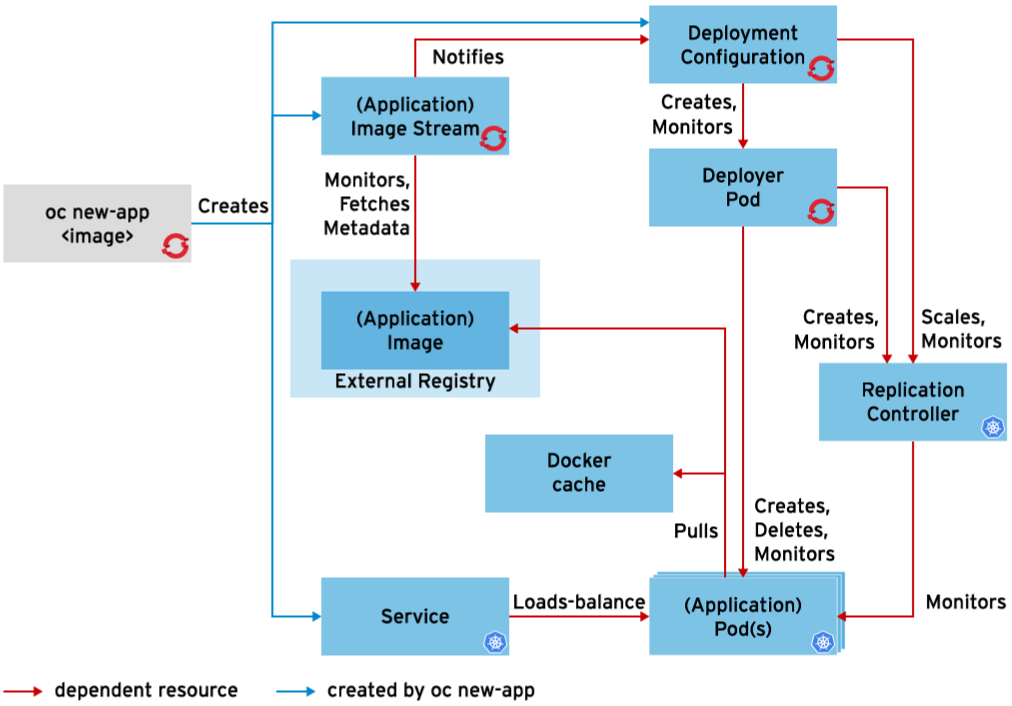


Figure 6.4: Deployment Configuration and dependent resources

To create an application based on an image from a private registry:

```
oc new-app --docker-image=myregistry.com/mycompany/myapp --name=myapp
```

To create 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, it's associated concepts, and more advanced ways to use **oc new-app** to build applications for OCP in the next section.

Useful Commands to Manage OCP Resources

There are several essential commands used to manage OpenShift resources. The following list describes these commands.

Typically, as an administrator, the tool you will most likely to use is **oc get** command. This allows a user to get 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:**

```
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 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

NAME       REVISION   DESIRED   CURRENT   TRIGGERED BY
dc/nginx   1          1         1         config,image(nginx:latest)

NAME         DESIRED   CURRENT   READY     AGE
rc/nginx-1   1         1         1         1h

NAME        CLUSTER-IP     EXTERNAL-IP   PORT(S)          AGE
svc/nginx   172.30.72.75   <none>        80/TCP,443/TCP   1h

NAME                READY     STATUS    RESTARTS   AGE
po/nginx-1-ypp8t   1/1       Running   0          1h
```

oc describe RESOURCE 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
...
No events
```

oc export

This command can be used to export a definition of a resource. 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 OCP 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 non-interactive batch commands as part of a script.

Demonstration: Creating Basic Kubernetes Resources

1. Open an SSH session to the **ocp** VM and start the OpenShift cluster, if not running.

   ```
   [student@workstation ~]$ ssh ocp
   [student@ocp ~]$ ./ocp-up.sh
   [student@ocp ~]$ Ctrl+D
   [student@workstation ~]$
   ```

2. Log in to OCP as the **`developer`** user on the **`workstation`** VM:

   ```
   [student@workstation ~]$ oc login -u developer -p developer \
   https://ocp.lab.example.com:8443
   ```

3. Create a new project for the resources you will create during this demonstration:

   ```
   [student@workstation ~]$ oc new-project basic-kubernetes
   ```

4. 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.

Change the default security policy to allow containers to run as root:

```
[student@workstation ~]$ oc adm policy add-scc-to-user anyuid -z default
```

5. Create a new application from the **nginx** container image using the **oc new-app** command.

   Use the **--docker-images** option from **oc new-app** and 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=infrastructure.lab.example.com:5000/nginx:latest \
        --insecure-registry=true --name=nginx
--> Found Docker image 5e69fe4 (9 days old) from infrastructure.lab.example.com:5000
 for "infrastructure.lab.example.com:5000/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"
    * Ports 443/tcp, 80/tcp will be load balanced by service "nginx"
      * Other containers can access this service through the hostname "nginx"
    * WARNING: Image "infrastructure.lab.example.com:5000/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
    Run 'oc status' to view your app.
```

6. Run the **oc status** command to view the status of the new application, and to check if the deployment of the Nginx image was successful:

```
[student@workstation ~]$ oc status
In project basic-kubernetes on server https://10.0.2.15:8443

svc/nginx - 172.30.72.75 ports 80, 443
  dc/nginx deploys istag/nginx:latest
    deployment #1 deployed 2 minutes ago - 1 pod
```

7. List the pods in this project to check 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
```

8. 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
Security Policy:     anyuid
Node:                10.0.2.15/10.0.2.15
Start Time:          Thu, 06 Apr 2017 13:03:37 +0000
Labels:              app=nginx
```

```
                    deployment=nginx-1
                    deploymentconfig=nginx
Status:             Running
IP:                 172.17.0.3
...
```

9. List the services in this project and check if a service to access the Nginx pod was created:

```
[student@workstation ~]$ oc get svc
NAME      CLUSTER-IP     EXTERNAL-IP   PORT(S)          AGE
nginx     172.30.72.75   <none>        80/TCP,443/TCP   4m
```

10. Describe the **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
Selector:               app=nginx,deploymentconfig=nginx
Type:                   ClusterIP
IP: 172.30.72.75
Port:                   80-tcp 80/TCP
Endpoints:              172.17.0.3:80
Port:                   443-tcp 443/TCP
Endpoints:              172.17.0.3:443
Session Affinity:       None
No events.
```

11. Open an SSH session to the **ocp** VM and access the default Nginx home page using the service IP:

```
[student@workstation ~]$ ssh ocp curl -s http://172.30.72.75
<!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 the **ocp** VM is the machine running the all-in-one OCP cluster.

12. 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/3.5/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/3.5/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/DO180/labs/mysql-openshift**
Resources:	Red Hat Software Collections official MySQL 5.6 image **(rhscl/mysql-56-rhel7)**

Outcomes

You should be able to create and deploy a MySQL database pod on OpenShift.

Before you begin

This exercise has no prerequisites.

Steps

1. Start an OpenShift cluster in the **ocp** VM.

   1.1. Open an SSH session to the **ocp** VM and verify that the OpenShift cluster is running.

   ```
   [student@workstation ~]$ ssh ocp
   [student@ocp ~]$ ./ocp-up.sh
   ```

   The **ocp-up.sh** checks that the OCP cluster is running and, if true, exits without displaying anything, otherwise it starts the cluster.

   1.2. Log out from the **ocp** VM and return to the **workstation** VM.

   ```
   [student@ocp ~]$ Ctrl+D
   Connection to ocp closed.
   [student@workstation ~]$
   ```

2. Log in to OCP as a developer user and create a new project for this exercise.

   2.1. From the **workstation** VM, log in to OCP as the **developer** user:

   ```
   [student@workstation ~]$ oc login -u developer -p developer \
   https://ocp.lab.example.com:8443
   ```

   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.

2.2. Create a new project for the resources you will create during this exercise:

```
[student@workstation ~]$ oc new-project mysql-openshift
```

3. Create a new application from the **`rhscl/mysql-56-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`**) using the **`-e`** option.

   Use the **`--docker-images`** option for **`oc new-app`** and 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=infrastructure.lab.example.com:5000/rhscl/mysql-56-rhel7:latest \
--insecure-registry=true --name=mysql-openshift \
-e MYSQL_USER=user1 -e MYSQL_PASSWORD=mypa55 -e MYSQL_DATABASE=testdb \
-e MYSQL_ROOT_PASSWORD=r00tpa55
--> Found Docker image 71ac5db (6 weeks old) from
 infrastructure.lab.example.com:5000 for "infrastructure.lab.example.com:5000/rhscl/
mysql-56-rhel7:latest"
...
--> Creating resources ...
    imagestream "mysql-openshift" created
    deploymentconfig "mysql-openshift" created
    service "mysql-openshift" created
--> Success
    Run 'oc status' to view your app.
```

4. Verify if the MySQL pod was created successfully and view details about the pod and it's service.

4.1. Run the **`oc status`** command to view the status of the new application, and to check if the deployment of the MySQL image was successful:

```
[student@workstation ~]$ oc status
In project mysql-openshift on server https://10.0.2.15:8443

svc/mysql-openshift - 172.30.192.199:3306
  dc/mysql-openshift deploys istag/mysql-openshift:latest
    deployment #1 deployed 21 seconds ago - 1 pod...
```

4.2. List the pods in this project to check if the MySQL pod is ready and running:

```
[student@workstation ~]$ oc get pods
NAME                         READY     STATUS    RESTARTS   AGE
mysql-openshift-1-kb98m      1/1       Running   0          2m
```

4.3. Use the **`oc describe`** command to view more details about the pod:

```
[student@workstation ~]$ oc describe pod mysql-openshift-1-kb98m
Name:                mysql-openshift-1-kb98m
Namespace:           mysql-openshift
Security Policy:     restricted
```

```
Node:               10.0.2.15/10.0.2.15
Start Time:         Mon, 10 Apr 2017 09:28:25 +0000
Labels:             app=mysql-openshift
                    deployment=mysql-openshift
                    deploymentconfig=mysql-openshift
Status:             Running
IP:                 172.17.0.4
...
```

4.4. List the services in this project and check if a service to access the MySQL pod was created:

```
[student@workstation ~]$ oc get svc
NAME              CLUSTER-IP         EXTERNAL-IP   PORT(S)    AGE
mysql-openshift   172.30.192.199   <none>        3306/TCP   5m
```

4.5. Describe the **mysql-openshift** service 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
Selector:              app=mysql-openshift,deploymentconfig=mysql-openshift
Type: ClusterIP
IP:                    172.30.192.199
Port:                  3306-tcp        3306/TCP
Endpoints:             172.17.0.4:3306
Session Affinity:      None
No events.
```

4.6. View details about the DeploymentConfig (dc) for this application:

```
[student@workstation ~]$ oc describe dc mysql-openshift
Name:            mysql-openshift
Namespace:       mysql-openshift
Created:         About a minute ago
Labels:          app=mysql-openshift
...
Deployment #1 (latest):
 Name:            mysql-openshift-1
 Created:         about a minute 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
...
```

5. For external clients to access the database service from outside the OCP cluster, you can expose the service through a port on the host machine (in this environment, the workstation VM) where the pod is running. This can be achieved by changing the default type of the service to the **NodePort** type.

Note

Exposing a service to the outside world using the **NodePort** type is the kubernetes approach, which is no longer the preferred way to achieve this objective. OpenShift has a simpler and more elegant approach, using the *Routes* concept, which is covered later in this chapter.

5.1. Change the type of the **mysql-openshift** service from the default **ClusterIP** type to **NodePort**. Edit the service using the **oc edit svc** command:

```
[student@workstation ~]$ oc edit svc mysql-openshift
```

5.2. The **oc edit** command opens up a **vi** buffer, allowing you to edit the service attributes. Change the **type** of the service to **NodePort**, and add a new attribute called **nodePort** to the **ports** array with a value of **30306** for the attribute:

```
# Please edit the object below. Lines beginning with a '#' will be ignored,
# and an empty file will abort the edit. If an error occurs while saving this
 file will be
# reopened with the relevant failures.
#
apiVersion: v1
kind: Service
metadata:
...
  ports:
  - name: 3306-tcp
    port: 3306
    protocol: TCP
    targetPort: 3306
    nodePort: 30306
  selector:
    app: mysql-openshift
    deploymentconfig: mysql-openshift
  sessionAffinity: None
  type: NodePort
status:
...
```

Save the contents of the buffer and exit from the editor.

5.3. Verify your changes by running the **oc describe svc** command again:

```
[student@workstation ~]$ oc describe svc mysql-openshift
Name:                  mysql-openshift
Namespace:             mysql-openshift
Labels:                app=mysql-openshift
Selector:              app=mysql-openshift,deploymentconfig=mysql-openshift
Type:                  NodePort
IP:                    172.30.192.199
Port:                  3306-tcp      3306/TCP
NodePort:              3306-tcp      30306/TCP
Endpoints:             172.17.0.4:3306
Session Affinity:      None
```

```
No events.
```

6. Connect to the MySQL database server and verify if the database was created successfully.

   6.1. Open an SSH session to the **ocp** VM.

```
[student@workstation ~]$ ssh ocp
[student@ocp ~]$
```

   6.2. Connect to the MySQL server using the MySQL client with the public IP of the **ocp** VM and the **nodePort**:

```
[student@ocp ~]$ mysql -hocp -P30306 -uuser1 -pmypa55
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)]>
```

   6.3. Verify if the **testdb** database has been created:

```
MySQL [(none)]> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| testdb             |
+--------------------+
2 rows in set (0.00 sec)
```

   6.4. Exit from the MySQL prompt and terminate the SSH session:

```
MySQL [(none)]> exit
Bye
[student@ocp ~]$ Ctrl+D
Connection to ocp closed.
[student@workstation ~]$
```

7. Verify that the database was correctly set up. Run the following grading script from a terminal window:

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

8. Delete the project and 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 OCP.

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`** command when the argument is an application source code repository. Notice that a Deployment Configuration and all its dependent resources are also created.

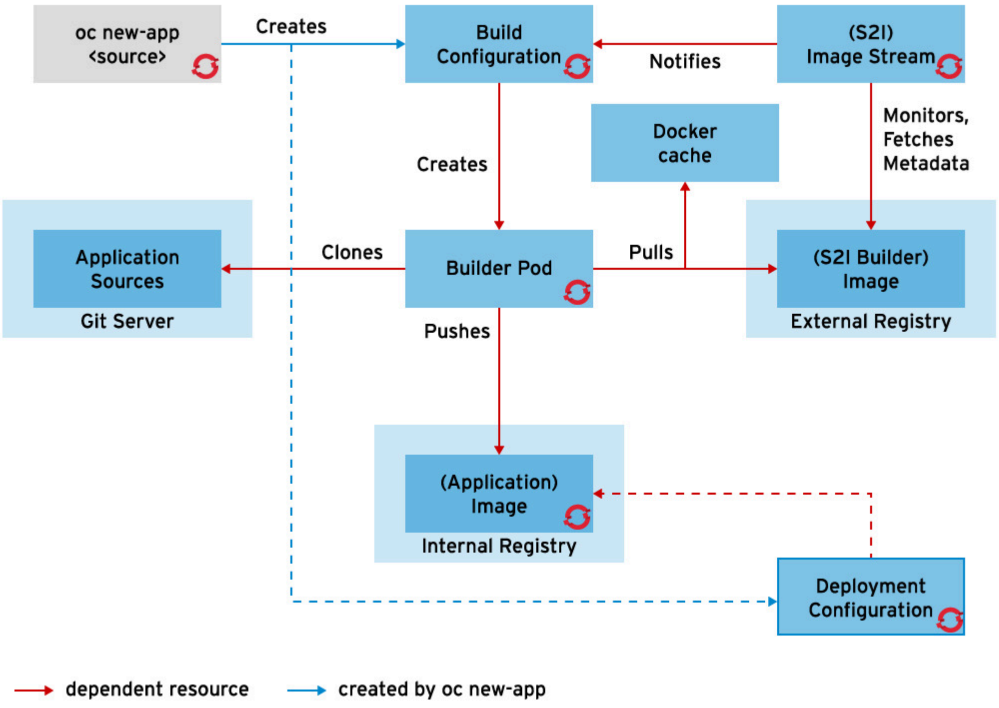


Figure 6.5: Deployment Configuration and dependent resources

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 + 1 more...

ruby                                  172.30.0.103:5000/openshift/ruby
                                   2.3,2.2,2.0
```

Note

Your OCP instance may have more or fewer image streams depending on local additions and OCP 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 OCP CLI.

An application can be created using the S2I process with the **oc new-app** command from the CLI.

```
$ oc new-app ❶php~http://infrastructure.lab.example.com/app❷ --name=myapp❸
```

❶ The image stream used in the process appears to the left of the tilde (~).

❷ The URL after the tilde indicates the location of the source code's Git repository.

❸ Sets the application name.

The **oc new-app** command allows for 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 .
```

Warning

When using a local Git repository, the repository must have a remote origin that points to a URL accessible by the OpenShift instance.

It is also possible to create an application using a remote Git repository and a context subdirectory:

```
$ oc new-app https://github.com/openshift/sti-ruby.git \
  --context-dir=2.0/test/puma-test-app
```

Finally, it is possible to create an application using a remote Git repository with a specific branch reference:

```
$ oc new-app https://github.com/openshift/ruby-hello-world.git#beta4
```

If an image stream is not specified in the command, **new-app** attempts to determine which language builder to use based on the presence of certain files in the root of the repository:

Language	Files
ruby	**Rakefile**, **Gemfile**, **config.ru**

Language	Files
Java EE	**pom.xml**
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://infrastructure.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", ❶
    "apiVersion": "v1",
    "metadata": {
        "name": "myapp", ❷
        "creationTimestamp": null
        "labels": {
                    "app": "myapp"
                },
                "annotations": {
                    "openshift.io/generated-by": "OpenShiftNewApp"
                }
    },
    "spec": {},
    "status": {
        "dockerImageRepository": ""
    }
},
...
```

❶ Define a resource type of image stream.

❷ 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", ❶
            "apiVersion": "v1",
            "metadata": {
                "name": "myapp", ❷
                "creationTimestamp": null,
                "labels": {
                    "app": "myapp"
                },
                "annotations": {
                    "openshift.io/generated-by": "OpenShiftNewApp"
                }
            },
            "spec": {
                "triggers": [
                    {
                        "type": "GitHub",
                        "github": {
                            "secret": "S5_4BZpPabM6KrIuPBvI"
                        }
                    },
                    {
                        "type": "Generic",
                        "generic": {
                            "secret": "3q8K8JNDoRzhjoz1KgMz"
                        }
                    },
                    {
                        "type": "ConfigChange"
                    },
                    {
                        "type": "ImageChange",
                        "imageChange": {}
                    }
                ],
                "source": {
                    "type": "Git",
                    "git": {

                        "uri": "http://infrastructure.lab.example.com/app" ❸
                    }
                },
                "strategy": {

                    "type": "Source", ❹
                    "sourceStrategy": {
                        "from": {
                            "kind": "ImageStreamTag",
                            "namespace": "openshift",

                            "name": "php:5.5" ❺
                        }
                    }
                },
                "output": {
                    "to": {
                        "kind": "ImageStreamTag",

                        "name": "myapp:latest" ❻
                    }
                },
                "resources": {},
                "postCommit": {},
                "nodeSelector": null
            },
            "status": {
                "lastVersion": 0
            }
        },
...
```

❶ Define a resource type of **`BuildConfig`**.

❷ Name the **`BuildConfig`** as **`myapp`**.

❸ Define the address to the source code Git repository.

❹ Define the strategy to use S2I.

❺ Define the builder image as the **php:latest** image stream.

❻ Name the output image stream as **`myapp:latest`**.

The third resource is the deployment configuration that is responsible for customizing the deployment process into OpenShift. They 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 deployments.

- 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",  ❺
                                "ports": [  ❻
                                    {
                                        "containerPort": 8080,
                                        "protocol": "TCP"
                                    }
                                ],
                                "resources": {}
                            }
                        ]
                    }
                }
            },
            "status": {}
        },
...
```

❶ Define a resource type of **DeploymentConfig**.

❷ Name the **DeploymentConfig** as **myapp**.

❸ A configuration change trigger causes a new deployment to be created any time the replication controller template changes.

❹ 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.

❺ Define that the **library/myapp:latest** container image should be deployed.

❻ 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 to viewing of the build logs. The following command shows the last few lines of the build log:

```
$ oc logs build/myapp-1
```

Warning

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/openshift_container_platform/3.5/html/architecture/

Additional information about the S2I build process is available in 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 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/DO180/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 ~/DO180/labs/s2i
```

Steps

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/5.6
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
```

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 and create a new project named **`s2i`**:

```
[student@workstation s2i]$ oc login -u developer https://ocp.lab.example.com:8443
Logged into "https://172.25.250.11:8443" as "developer" using existing credentials.
...
[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 **`developer`**.

3. Create a new PHP application using Source-to-Image from the Git repository at **`http://infrastructure.lab.example.com/php-helloworld`**.

3.1. Use the **`oc new-app`** command to create the PHP application.

Important

The following example has been split into two lines for printing, type the entire command on one line.

```
[student@workstation s2i]$ oc new-app \
 php:5.6~http://infrastructure.lab.example.com/php-helloworld
```

3.2. 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://infrastructure.lab.example.com/php-helloworld
From Image:       ImageStreamTag openshift/php:5.6
Output to:        ImageStreamTag php-helloworld:latest
...
Build                  Status  Duration      Creation Time
php-helloworld-1 complete 3s 2017-04-09 17:18:29 +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.

3.3. 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
Cloning "http://infrastructure.lab.example.com/php-helloworld" ...
 Commit: ecb93d1e41f2eb8f9a0ba59c1317ec998be17c0f (change test)
 Author: Jim Rigsbee <jrigsbee@redhat.com>
 Date: Sat Apr 8 23:49:21 2017 +0000
---> Installing application source...
Pushing image 172.30.7.74:5000/s2i/php-helloworld:latest ...
Pushed 0/9 layers, 1% complete
Pushed 1/9 layers, 11% 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.

3.4. 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(php-helloworld@latest, auto=true)
Strategy:         Rolling
Template:
  Labels:         app=php-helloworld
                  deploymentconfig=php-helloworld
...
```

```
  Containers:
   php-helloworld:
    Image:                 172.30.7.74:5000/s2i/php-helloworld...
    Port:                  8080/TCP
    Volume Mounts:         <none>
    Environment Variables: <none>
  No volumes.

Deployment #1 (latest):
 Name:          php-helloworld-1
 Created:       11 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
...
```

3.5. Review the **service** for this application:

```
[student@workstation s2i]$ oc describe svc/php-helloworld
Name:              php-helloworld
Namespace:         s2i
Labels:            app=php-helloworld
Selector:          app=php-helloworld,deploymentconfig=php-helloworld
Type:              ClusterIP
IP:                172.30.161.172
Port:              8080-tcp 8080/TCP
Endpoints:         172.17.0.4:8080
Session Affinity:  None
No events.
```

3.6. 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@ocp curl -s 172.30.161.172:8080
Hello, World! php version is 5.6.25
```

4. 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.

4.1. Clone the project locally using **git**:

```
[student@workstation s2i]$ git clone \
http://infrastructure.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
```

4.2. 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.

4.3. 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 eb438ba] Changed index page contents.
 1 file changed, 1 insertion(+), 1 deletion(-)
[student@workstation php-helloworld]$ git push origin master
...
Counting objects: 5, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 287 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To http://infrastructure.lab.example.com/php-helloworld
   ecb93d1..eb438ba  master -> master
[student@workstation php-helloworld]$ cd ..
```

4.4. 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
```

4.5. Test that your changes are served by the application:

```
[student@workstation s2i]$ ssh student@ocp curl -s 172.30.161.172:8080
Hello, World! php version is 5.6.25
A change is a coming!
```

5. Grade the lab:

```
[student@workstation s2i]$ lab s2i grade
Accessing PHP web application............. SUCCESS
```

6. 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 OCP instance, routes allow for network access to pods from users and applications outside the OCP instance.

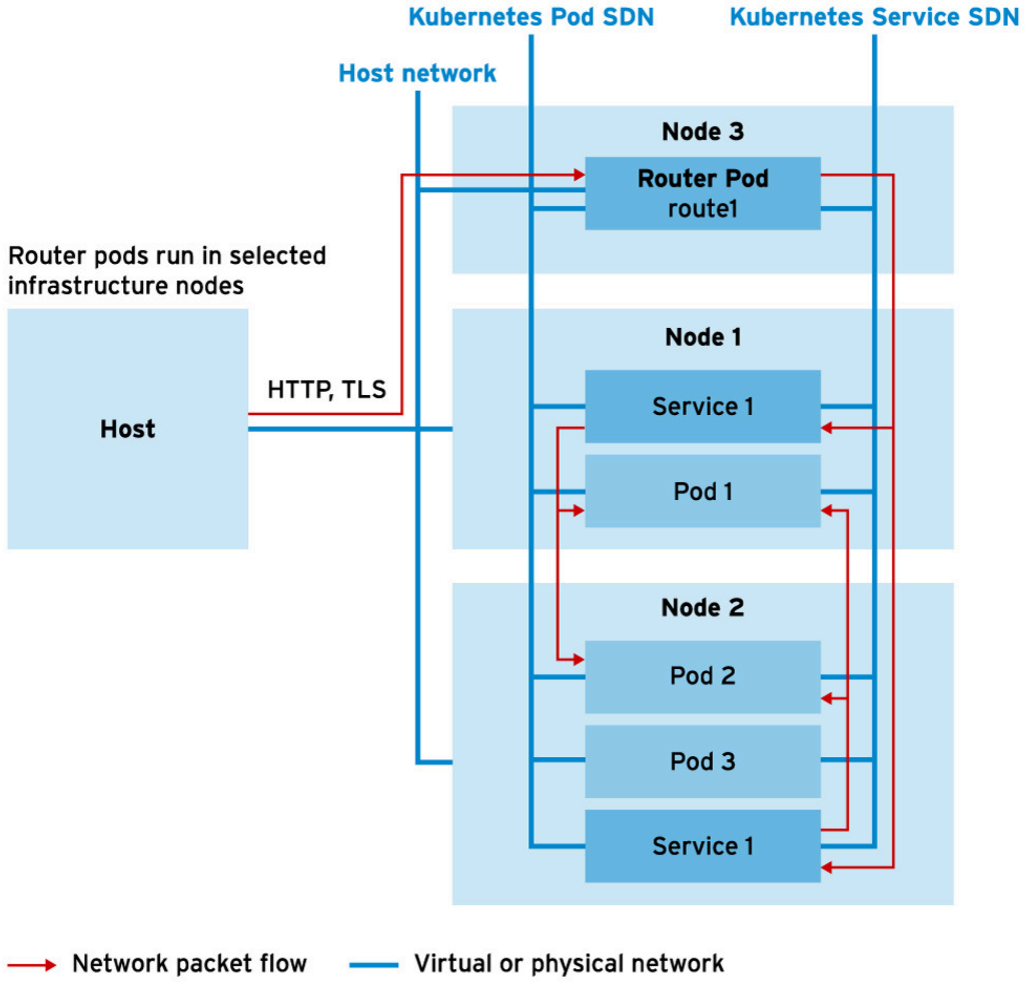


Figure 6.6: OpenShift routes and Kubernetes services

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 OCP router connects directly to the pods using the internal pod software-defined network (SDN), using the service only to find the end points, that is, the pods exposed by the service.

OCP routes are implemented by a shared router service, which runs as pods inside the OCP 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 OCP 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.cloudapps.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` attributes 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 OCP router IP address.

- **`to`** is an object stating the **`"kind"`** of resource this route points to, which in this case is an OCP Service and the **`"name"`** of that resource.

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 OCP 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 OCP 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 OCP.

For example, creating route **quote** in project **test** from an OCP instance where the wildcard domain is **cloudapps.example.com** results in the FQDN **quote-test.cloudapps.example.com**.

Insight

The DNS server that hosts the wild-card domain knows nothing about route host names. It simply resolves any name to the configured IPs. Only the OCP 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 OCP 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
```

When using the **oc cluster up** command to run the OpenShift cluster as we do in this course, this configuration file can be found at **/var/lib/origin/openshift.local.config/master/master-config.yaml**.

References

Additional information about the architecture of routes in OCP is available in the *Routes* section of the OpenShift Container Platform documentation:

> Architecture
> https://access.redhat.com/documentation/en-us/openshift_container_platform/3.5/html/architecture/

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/openshift_container_platform/3.5/html/developer_guide

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.cloudapps.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
```

Steps

1. Login to OpenShift and create a new project named **`route`**:

```
[student@workstation ~]$ oc login -u developer https://ocp.lab.example.com:8443
Logged into "https://ocp.lab.example.com:8443" as "developer" using existing
 credentials.
...
[student@workstation ~]$ oc new-project route
Now using project "route" on server "https://ocp.lab.example.com: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.
```

You may have to enter credentials for the **`developer`** account. The password is **`developer`**. The current project in your command output may differ from the listing above.

2. Create a new PHP application using Source-to-Image from the Git repository at **`http://infrastructure.lab.example.com/php-helloworld`**.

2.1. Use the **`oc new-app`** command to create the PHP application.

Important

The following example has been split into two lines for printing. Type the entire command on one line.

```
[student@workstation ~]$ oc new-app \
 php:5.6~http://infrastructure.lab.example.com/php-helloworld
```

2.2. 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
```

2.3. 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
Selector:          app=php-helloworld,deploymentconfig=php-helloworld
Type:              ClusterIP
IP:                172.30.161.172
Port:              8080-tcp 8080/TCP
Endpoints:         172.17.0.4:8080
Session Affinity:  None
No events.
```

The IP address displayed in the output of the command may differ.

3. 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       TERMINATION
php-helloworld   php-helloworld-route.cloudapps.lab.example.com             php-
helloworld
      8080-tcp
[student@workstation ~]$ curl php-helloworld-route.cloudapps.lab.example.com
```

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** to verify that the application can be accessed with its route address.

4. Replace this route with a route named **xyz**.

4.1. Delete the current route:

```
[student@workstation ~]$ oc delete route/php-helloworld
route "php-helloworld" deleted
```

4.2. 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.cloudapps.lab.example.com             php-helloworld
      8080-tcp
```

Note the new FQDN that was generated.

4.3. Make an HTTP request using the FQDN on port 80:

```
[student@workstation ~]$ curl xyz-route.cloudapps.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.

5. Grade your lab progress:

```
[student@workstation ~]$ lab route grade
Accessing PHP web application............. SUCCESS
```

6. Clean up the lab environment by deleting the project:

```
[student@workstation ~]$ oc delete project route
project "route" deleted
```

This concludes this 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 created within those projects, and application resources examined and manipulated as needed.

Accessing the Web Console

The web console runs in a web browser. The URL is of the format **`https://{hostname of OCP master}:8443/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. In a cluster started by the **`oc cluster up`** command, the user name **`developer`** is created with a password of **`developer`**. The URL of the web console can be retrieved by issuing this command:

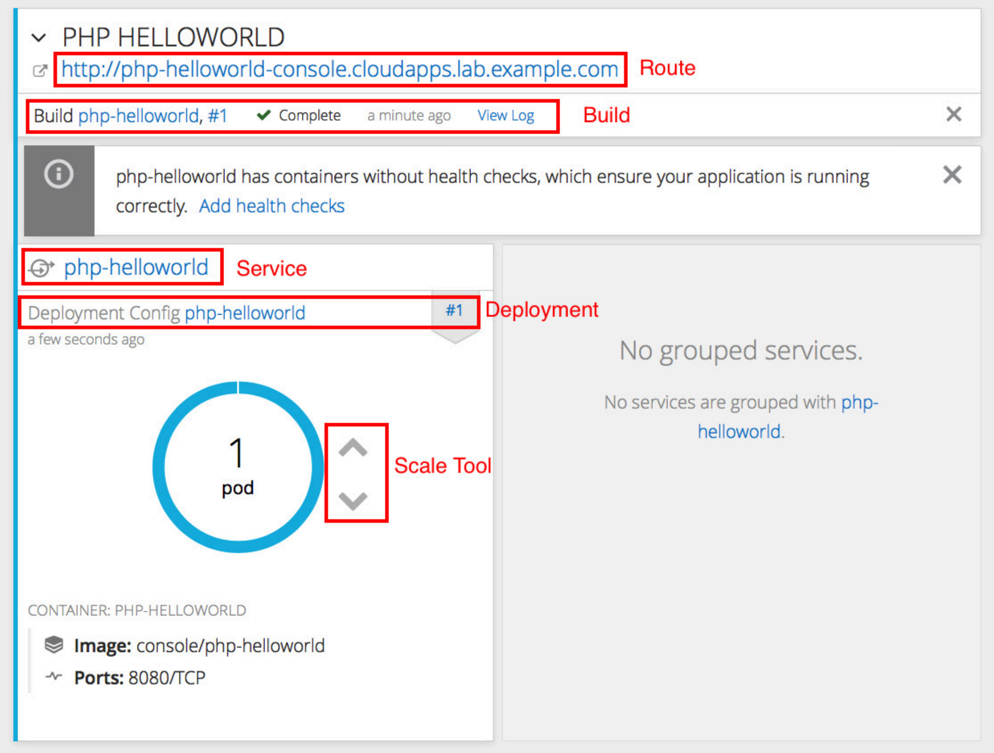
```
$ oc cluster status
The OpenShift cluster was started 4 minutes ago

Web console URL: https://workstation.lab.example.com:8443
...
```

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.



PHP HELLOWORLD

http://php-helloworld-console.cloudapps.lab.example.com

Build php-helloworld, #1 ✓ Complete a minute ago View Log

php-helloworld has containers without health checks, which ensure your application is running correctly. Add health checks

php-helloworld

Deployment Config php-helloworld #1

a few seconds ago

1 pod

CONTAINER: PHP-HELLOWORLD

Image: console/php-helloworld

Ports: 8080/TCP

No grouped services.

No services are grouped with php-helloworld.

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 a 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.cloudapps.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.

Steps

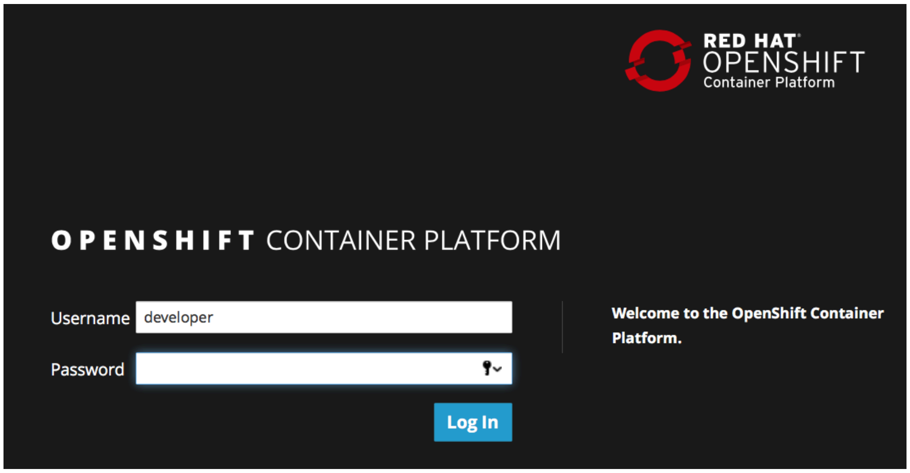
1. Access the OpenShift Web Console via a browser. Log in and create a new project.

   1.1. Find the web console address:

   ```
   [student@workstation ~]$ ssh student@ocp oc cluster status
   The OpenShift cluster was started 4 minutes ago

   Web console URL: https://ocp.lab.example.com:8443
   ...
   ```

   1.2. Enter the web console URL in the browser and trust the self-signed certificate generated by OpenShift when the cluster was established with the **`oc cluster up`** command. The cluster generates certificates every time it is started with a new configuration store. In this course, the cluster uses a persistent configuration store at **`/var/lib/origin`**. You should only have to trust this certificate once.

   1.3. Login with **`developer`** as the user name and **`developer`** as the password.



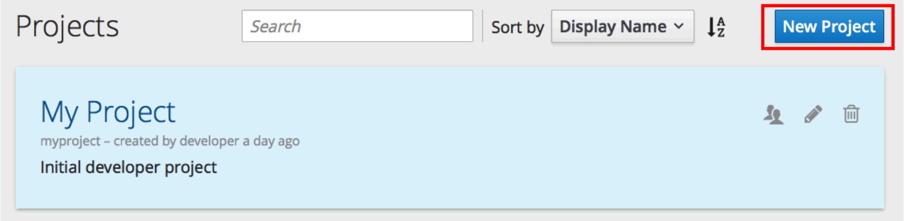
Username: developer

Password:

Welcome to the OpenShift Container Platform.

Figure 6.8: Web console login

1.4. Create a new project named **console**. You may type any values you wish in the other fields.

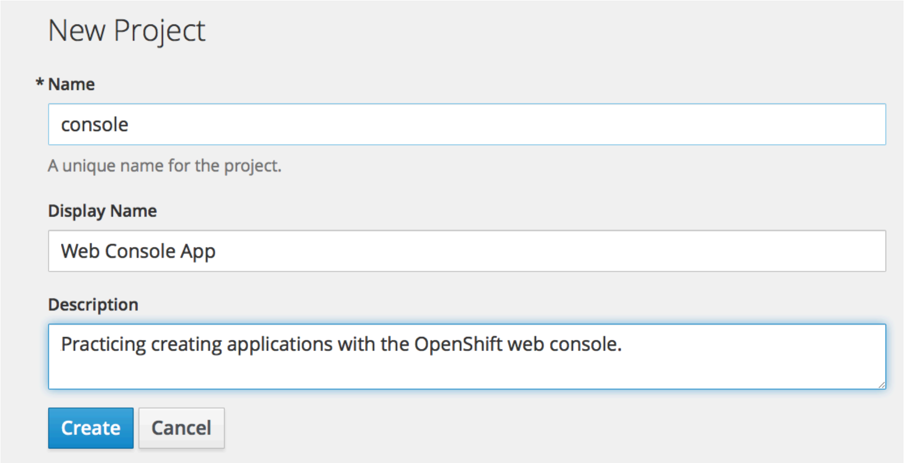


My Project

myproject – created by developer a day ago

Initial developer project

Figure 6.9: Create a new project - step 1



* Name: console

A unique name for the project.

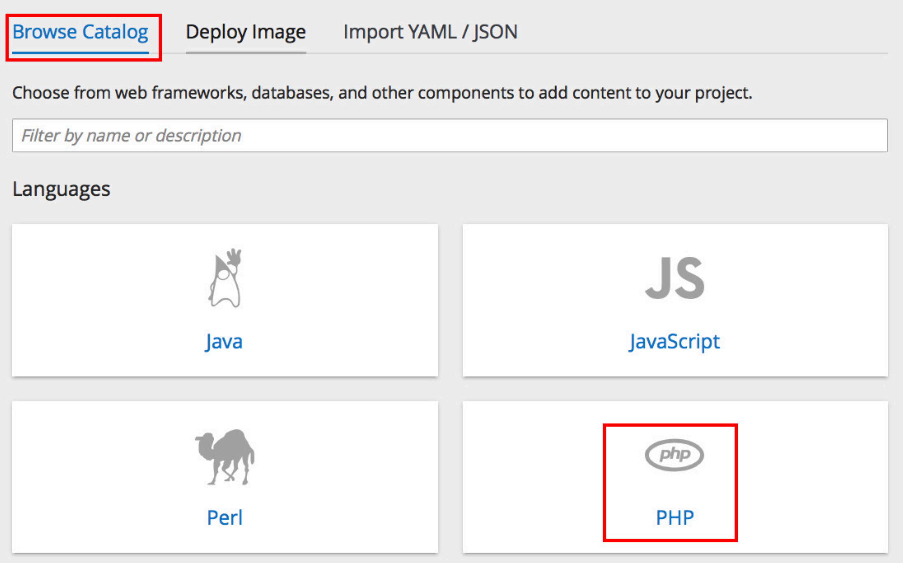
Display Name: Web Console App

Description: Practicing creating applications with the OpenShift web console.

Figure 6.10: Create a new project - step 2

2. Create the new **php-helloworld** application with a PHP template.

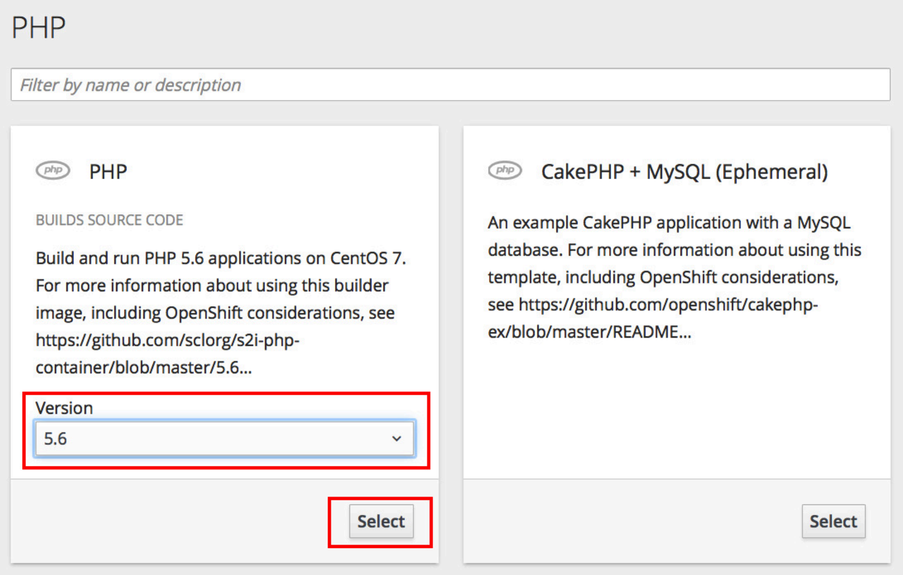
   2.1. Select the PHP Template.



Choose from web frameworks, databases, and other components to add content to your project.

Figure 6.11: Select the PHP template

2.2. Select PHP version 5.6.



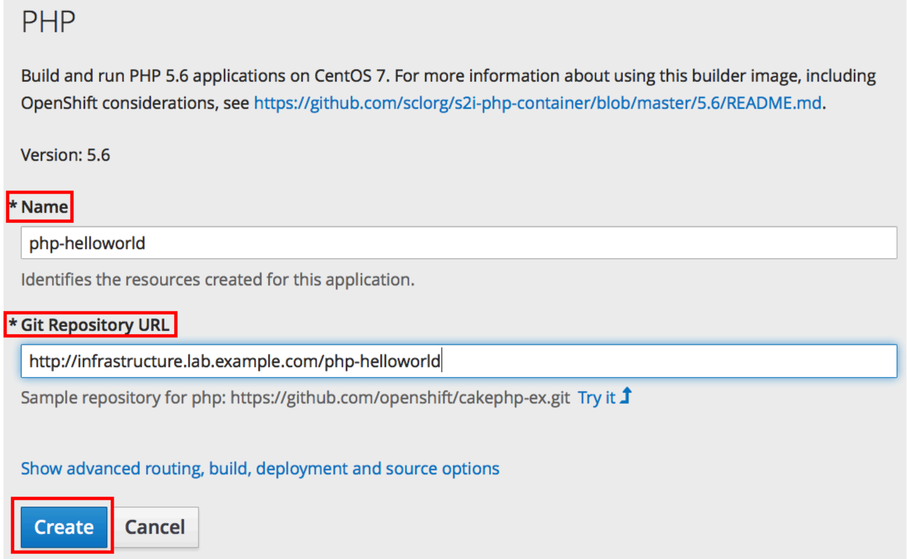
Build and run PHP 5.6 applications on CentOS 7. For more information about using this builder image, including OpenShift considerations, see https://github.com/sclorg/s2i-php-container/blob/master/5.6...

Version: 5.6

An example CakePHP application with a MySQL database. For more information about using this template, including OpenShift considerations, see https://github.com/openshift/cakephp-ex/blob/master/README...

Figure 6.12: Select PHP version 5.6

2.3. Enter **`php-helloworld`** as the name for the application and the location of the source code git repository: **`http://infrastructure.lab.example.com/php-helloworld`**.



PHP

Build and run PHP 5.6 applications on CentOS 7. For more information about using this builder image, including OpenShift considerations, see https://github.com/sclorg/s2i-php-container/blob/master/5.6/README.md.

Version: 5.6

* Name

php-helloworld

Identifies the resources created for this application.

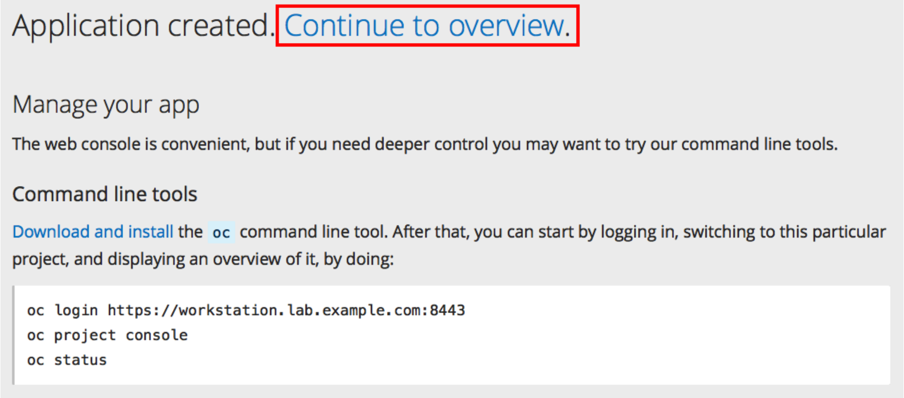
* Git Repository URL

http://infrastructure.lab.example.com/php-helloworld

Sample repository for php: https://github.com/openshift/cakephp-ex.git Try it

Figure 6.13: PHP application details

2.4. On the confirmation page, click on the **Continue to overview** link.



Application created. Continue to overview.

Manage your app

The web console is convenient, but if you need deeper control you may want to try our command line tools.

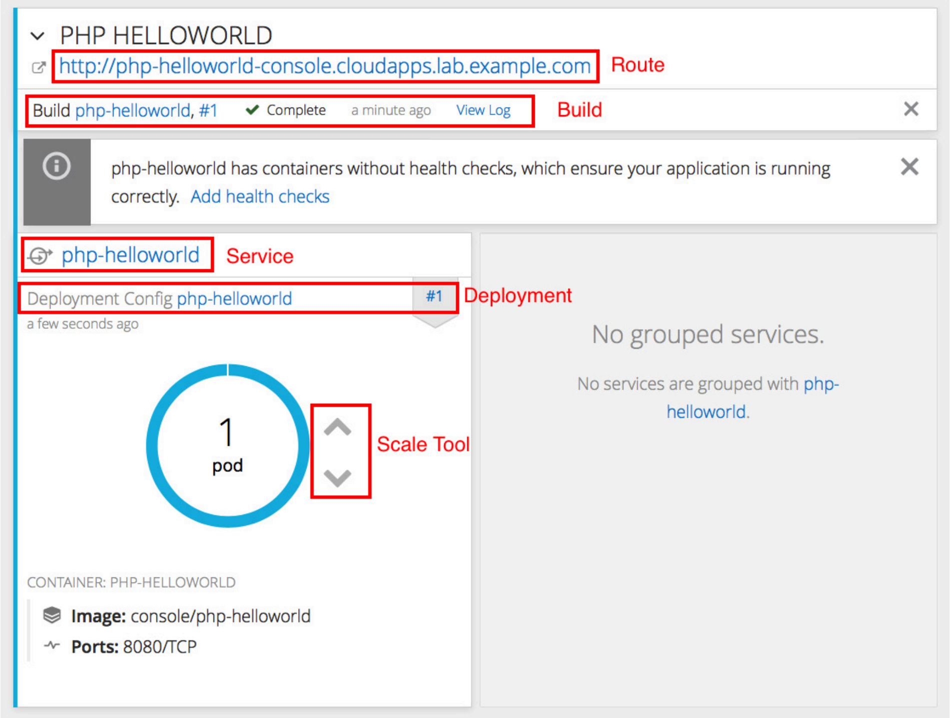
Command line tools

Download and install the `oc` command line tool. After that, you can start by logging in, switching to this particular project, and displaying an overview of it, by doing:

```
oc login https://workstation.lab.example.com:8443
oc project console
oc status
```

Figure 6.14: 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.



PHP HELLOWORLD

http://php-helloworld-console.cloudapps.lab.example.com

Build php-helloworld, #1 ✓ Complete a minute ago View Log

php-helloworld has containers without health checks, which ensure your application is running correctly. Add health checks

php-helloworld

Deployment Config php-helloworld #1

a few seconds ago

1 pod

CONTAINER: PHP-HELLOWORLD

Image: console/php-helloworld

Ports: 8080/TCP

No grouped services.

No services are grouped with php-helloworld.

Figure 6.15: 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. In the build section of the Overview page, 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 beside the label **Deployment Config**. 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 service section. See the circled number 3 in the previous image. 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 under 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://infrastructure.lab.example.com/php-helloworld
Cloning into 'php-helloworld'...
remote: Counting objects: 12, done.
remote: Compressing objects: 100% (8/8), done.
remote: Total 12 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (12/12), done.
[student@workstation ~]$ cd php-helloworld
```

4.2. Change or 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]$ 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://infrastructure.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** in the build section. Click the upper-right menu, and select **Start Build**. Wait for the build to complete. Examine the build log on the build page or the Overview page to determine when the build completes.

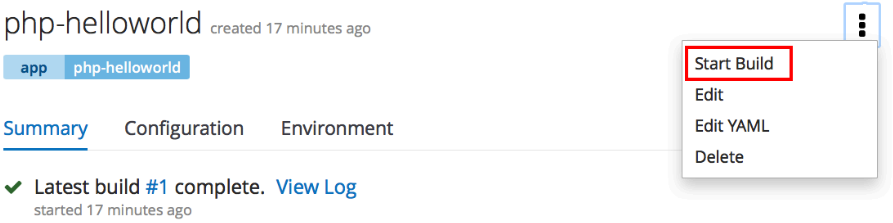


Figure 6.16: Start a new application build

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 home icon to go back to the list of projects. Click the trash can icon and enter the name of the project to confirm its deletion.

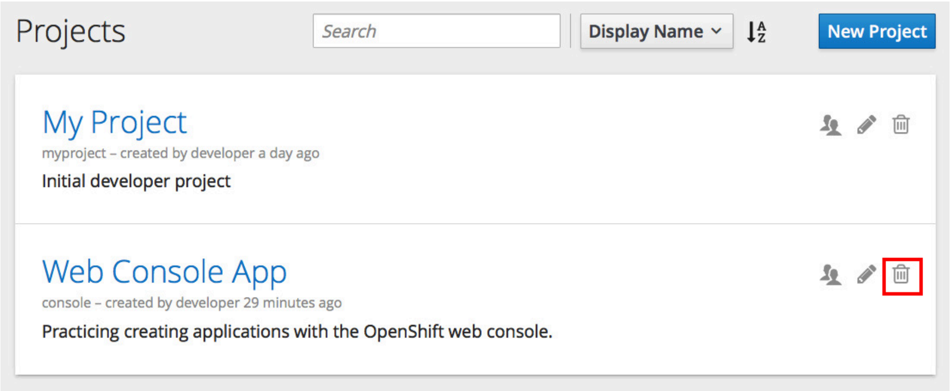

Projects
Search
Display Name
New Project
My Project
myproject – created by developer a day ago
Initial developer project
Web Console App
console – created by developer 29 minutes ago
Practicing creating applications with the OpenShift web console.

Figure 6.17: Delete the project

This concludes this guided exercise.

Lab: Deploying Containerized Applications on OpenShift

In this lab, you will create an application using the OpenShift Source-to-Image facility.

Resources	
Files:	N/A
Application URL:	**`http://temps-ocp.cloudapps.lab.example.com`**

Outcomes

You should be able to create an OpenShift application and access it through a web browser.

Before you begin

Get the lab files by executing the lab script:

```
[student@workstation ~]$ lab openshift setup
```

Steps

1. Login as **`developer`** and create the project **`ocp`**.

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://infrastructure.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 application.

Solution

In this lab, you will create an application using the OpenShift Source-to-Image facility.

Resources	
Files:	N/A
Application URL:	**`http://temps-ocp.cloudapps.lab.example.com`**

Outcomes

You should be able to create an OpenShift application and access it through a web browser.

Before you begin

Get the lab files by executing the lab script:

```
[student@workstation ~]$ lab openshift setup
```

Steps

1. Login as **`developer`** and create the project **`ocp`**.

   Issue the following commands:

   ```
   [student@workstation openshift]$ oc login -u developer \
   https://ocp.lab.example.com:8443
   [student@workstation openshift]$ oc new-project ocp
   ```

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://infrastructure.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://infrastructure.lab.example.com/temps
   [student@workstation openshift]$ oc logs -f bc/temps # wait until build is
    successful
   [student@workstation openshift]$ oc get pods -w # wait until pod is deployed
   [student@workstation openshift]$ oc expose svc/temps
   ```

3. Test the application in a web browser using the route URL.

   Discover the URL for the route.

   ```
   [student@workstation openshift]$ oc get route/temps
   ```

4. Grade your work:

   ```
   [student@workstation openshift]$ lab openshift grade
   Accessing PHP web application............ SUCCESS
   ```

5. Delete the application.

   There are two solutions:

   ```
   [student@workstation openshift]$ oc delete all -l app=temps
   ```

   or

   ```
   [student@workstation openshift]$ oc delete project ocp
   ```

Summary

In this chapter, you learned:

- The OpenShift command line client **oc** is used to perform the following tasks in an OCP cluster:
  - Logging in and out of an OCP 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 Platform organizes entities in the OCP 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 OCP 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 OCP instance, routes allow for network access to pods from users and applications outside the OCP instance.

- You can create, build, deploy and monitor applications using the OCP web console.

CHAPTER 7

DEPLOYING MULTI-CONTAINER APPLICATIONS

Overview	
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 Docker link. • 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 find the others, to use the services on which it depends.

For example, there is an application which is composed of a front-end container, a back-end container, and a database. The front-end container needs to know the IP address of the back-end container. Similarly, the back-end container needs to know the IP address of the database container. Additionally, the IP address could change if a container was to be 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 will be covered later in the section.

Injecting Environment Variables into a Docker Container

It is a well-known recommended practice to parameterize application connection information to outside services, and one common way to do that is using operating system (OS) 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. But it does not show environment variables inherited by the container by the OS 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 for managing the container environment. Using the **ENV** instruction, you can define an environment variable that is available 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

Even though using environment variables to share information between containers with Docker makes service discovery technically possible, there are still some limitations, and lots of manual work involved in ensuring that all environment variables stay in sync, especially when you are working with many containers. Kubernetes provides an approach to solve this problem when you create 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 ordinarily 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 a 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 will be 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 will be listed.
- **<SERVICE_NAME>_PORT**: Represents the address, port, and protocol provided by the service for external access.

- **<SERVICE_NAME>_PORT_<PORT_NUMBER>_<PROTOCOL>**: 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**: Alias for the **<SERVICE_NAME>_SERVICE_PORT**.
- **<SERVICE_NAME>_PORT_<PORT_NUMBER>_<PROTOCOL>_ADDR**: Alias for the **<SERVICE_NAME>_SERVICE_HOST**.

Notice the variables following the convention **<SERVICE_NAME>_PORT_*** emulate the variables created by docker for linked containers.

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

If no protocol is provided, Kubernetes will consider a TCP port.

Examining the To Do List Application

Many labs from this course are based on a To Do List application. This application is architected in three tiers as illustrated by the following figure:

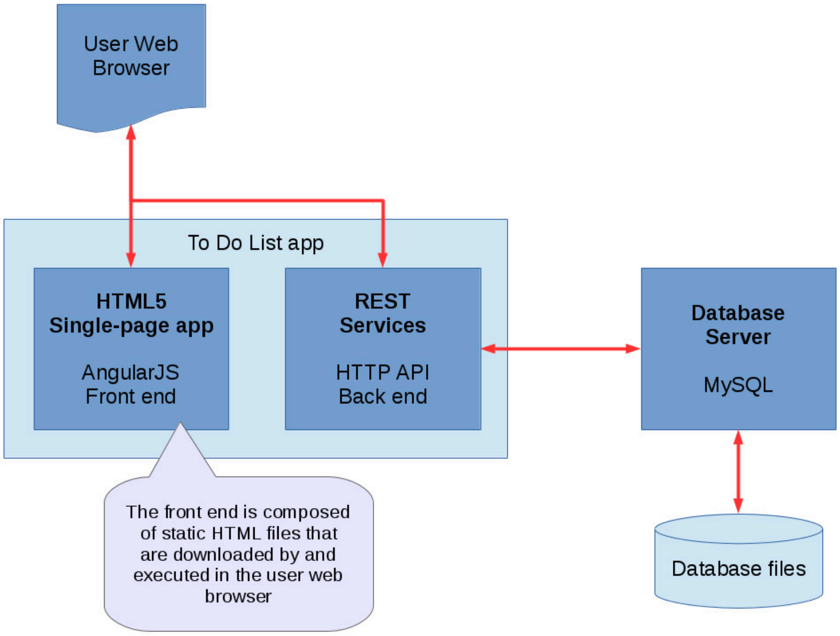
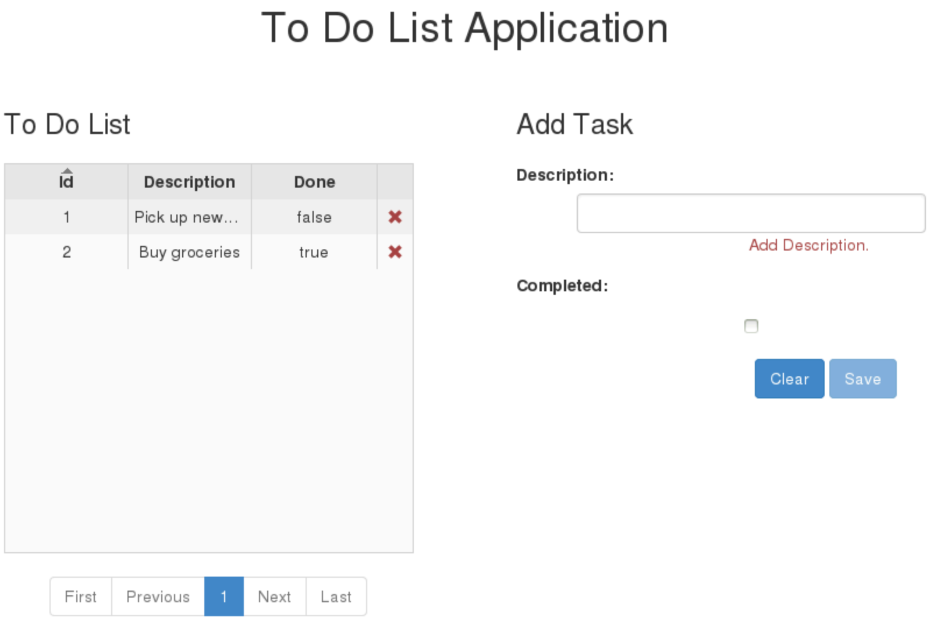


Figure 7.1: To Do List application logical architecture

- 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	✖

Add Task

Description:

Add Description.

Completed: □

Figure 7.2: To Do List application screen shot

On the left is a table with items to do, and on the right a form to add a new item or change an existing item.

The classroom materials server provides the application in two versions:

- One represents how a typical developer would create the application as a single unit, without caring to break it into tiers or services. Its sources are available at:

  `http://materials.example.com/todoapp/apps/nodejs`

- The other shows the changes needed to break the application presentation and business tiers so they can be deployed into different containers. Its sources are available at:

  `http://materials.example.com/todoapp/apps/nodejs_api`

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 intra-container communication.
   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 app can be scaled independently of the others, conserving resources.
   d. Monitoring, troubleshooting, and debugging are simplified when a container only provides a single purpose.
   e. Applications leveraging 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 to ensure the need environment variables are available.
   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

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 intra-container communication.
   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 app can be scaled independently of the others, conserving resources.**
   d. **Monitoring, troubleshooting, and debugging are simplified when a container only provides a single purpose.**
   e. Applications leveraging 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 to ensure the need environment variables are available.
   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 Docker link.
- Describe how environment variables are shared between linked containers.

Using Docker Linking to Share Environment Variables Between Containers

Environment variables should be enough to start an application composed of multiple containers. However, this is not a completely sufficient solution because:

- It is very easy to make mistakes when passing multiple **-e** options to the **docker run** command, and frequently multiple containers need the same environment variables to match exactly.
- It is still not possible for one container to get the IP address of another container, for example, an application container that needs to connect to a database container.

The **linked containers** feature from Docker solves both issues. It automatically copies all environment variables defined within a container to another container. It also defines environment variables based on the other container IP address and exposed ports.

Using linked containers is done by simply adding the option **--link container:alias** to the **docker run** command. For example, to link to a container named **mysql** using the **db** alias, the command would be:

```
$ docker run --link mysql:db --name my_container_name my_image_name
```

The new container (named **my_container_name** in the previous example) would then get all variables defined from the linked container (named **mysql** in the previous example). Those variable names are prefixed by **DB_ENV_** so that they do not conflict with the new container's own environment variables.

Note

The alias name uses uppercase to follow shell script conventions for environment variable names.

For example, the RHSCL MySQL container image from previous chapters defines the variable **MYSQL_USER** to provide the database user name with permissions to access the database. Any application container linked to a database container created from this MySQL image, as in the previous example, gets a variable named **DB_ENV_MYSQL_USER**.

The variables providing the container IP address and port follow a different naming convention. Four variables are created, but just two of them are sufficient for most applications. In the following example, *alias* is replaced with the alias given in the **`--link container:alias`** option and the *exposed-port* refers to the port the linked container exposed with the **`-p`** option when executing **`docker run`**:

- ***`{alias}`*****`_PORT_`*****`{exposed-port}`*****`_TCP_ADDR`**

- ***`{alias}`*****`_PORT_`*****`{exposed-port}`*****`_TCP_PORT`**

Continuing with the RHSCL MySQL image example, the application container would get the following variables:

- **`DB_PORT_3306_TCP_ADDR`**

- **`DB_PORT_3306_TCP_PORT`**

If the linked container exposes multiple ports, each of them generates a set of environment variables.

Guided Exercise: Linking the Web Application and MySQL Containers

In this lab, you will create a script that runs and links a Node.js application container and the MySQL container.

Resources	
Files:	**`/home/student/DO180/labs/linking-containers`**
Application URL:	**`http://127.0.0.1:30080/todo/`**
Resources:	RHSCL MySQL 5.6 image (**`rhscl/mysql-56-rhel7`**) Custom MySQL 5.6 image (**`do180/mysql-56-rhel7`**) RHEL 7.3 image (**`rhel7.3`**) Custom Node.js 4.0 image (**`do180/nodejs`**)

Outcomes

You should be able to link containers to create a multitiered application.

Before you begin

The workstation needs the To Do List application source code and lab files available. To achieve this goal, in a new terminal window run the following command:

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

1. Build the MySQL image.

   A custom MySQL 5.6 image is used for this exercise. It is configured to automatically run any scripts in **`/var/lib/mysql/init`** directory, in order to load the schema and some sample data into the database for the To Do List application on container start up.

   1.1. Review the Dockerfile.

   Using your preferred editor, open and examine the completed Dockerfile located at **`/home/student/DO180/labs/linking-containers/images/mysql/Dockerfile`**.

   1.2. Build the MySQL database image.

   Examine the **`/home/student/DO180/labs/linking-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 ~/DO180/labs/linking-containers/images/
   [student@workstation images]$ cd mysql
   [student@workstation nodejs]$ ./build.sh
   ```

   1.3. Wait for the build to complete, and then run the following command to verify the image built successfully:

```
[student@workstation mysql]$ docker images
REPOSITORY                                                 TAG
 IMAGE ID            CREATED              SIZE
do180/mysql-56-rhel7                                       latest
 5eb10a245ff7        About a minute ago   366.3 MB
infrastructure.lab.example.com:5000/rhscl/httpd-24-rhel7   latest
 533e496998ca        5 weeks ago          438 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/DO180/labs/linking-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 **$HOME** directory for application execution.
- The **ONBUILD** command is used to define actions which should be run when any child container image is built from this image. In this case, the **COPY** command copies the **run.sh** file and the **build** directory into **$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 the internet.

2.2. Build the Parent image.

Examine the **/home/student/DO180/labs/linking-containers/images/nodejs/build.sh** script to see how the image is built. To build the base image, run the **build.sh** script:

```
[student@workstation images]$ cd ~/DO180/labs/linking-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 built successfully:

```
[student@workstation nodejs]$ docker images
REPOSITORY                                                 TAG
 IMAGE ID            CREATED             SIZE
do180/nodejs                                               latest
 8587efed5e92        About a minute ago   465.7 MB
do180/mysql-56-rhel7                                       latest
 5eb10a245ff7        9 minutes ago        366.3 MB
infrastructure.lab.example.com:5000/rhscl/httpd-24-rhel7   latest
 533e496998ca        5 weeks ago          438 MB
...
```

3. Build the To Do App Child Image using the Node.js Dockerfile.

Warning

Execute the following commands because they are creating a different container.

3.1. Review the Dockerfile.

Using your preferred editor, open and examine the completed Dockerfile located at **/home/student/DO180/labs/linking-containers/deploy/nodejs/Dockerfile**.

3.2. Build the Child Image.

Examine the **/home/student/DO180/labs/linking-containers/deploy/nodejs/build.sh** script to see how the image is built. Run the following in order to build the child image:

```
[student@workstation nodejs]$ cd ~/DO180/labs/linking-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 the image built successfully:

```
[student@workstation nodejs]$ docker images
REPOSITORY                                                 TAG
 IMAGE ID            CREATED             SIZE
do180/todonodejs                                           latest
 91e4ae09d39e        14 seconds ago      490.4 MB
```

```
do180/nodejs                                               latest
 8587efed5e92        3 minutes ago       465.7 MB
do180/mysql-56-rhel7                                       latest
 5eb10a245ff7        11 minutes ago      366.3 MB
infrastructure.lab.example.com:5000/rhscl/httpd-24-rhel7   latest
 533e496998ca        5 weeks ago         438 MB
...
```

4. Explore the Environment Variables.

   Take a closer look at the environment variables that allow the Node.js REST API container to communicate with the MySQL container.

   4.1. View the file **`/home/student/DO180/labs/linking-containers/deploy/nodejs/nodejs-source/models/db.js`** that holds the database configuration:

   ```
   module.exports.params = {
       dbname: process.env.MYSQL_ENV_MYSQL_DATABASE,
       username: process.env.MYSQL_ENV_MYSQL_USER,
       password: process.env.MYSQL_ENV_MYSQL_PASSWORD,
       params: {
           host: process.env.MYSQL_PORT_3306_TCP_ADDR,
           port: process.env.MYSQL_PORT_3306_TCP_PORT,
           dialect: 'mysql'
       }
   };
   ```

   4.2. Take note of the variables being utilized by the REST API.

   These variables are created and populated by the **`docker run --link`** option when running the container, and their names are based on the alias of the container being linked to. The prefixes for these variables **`MYSQL_PORT_3306`** assume that the container that is being linked to has an alias **`mysql`** and that it exposes port 3306:

   - **`MYSQL_PORT_3306_TCP_PORT`**
   - **`MYSQL_PORT_3306_TCP_ADDR`**

   The remaining variables are passed in with values at the time that the container being linked to is run. In this lab, the MySQL container runs with these values passed in and then run the API container runs with a **`link`** to the MySQL container, which automatically defines these variables with the "MYSQL_ENV_" prefix, assuming that the alias of the container is "**`mysql`**".

   - **`MYSQL_ENV_MYSQL_DATABASE`**
   - **`MYSQL_ENV_MYSQL_USER`**
   - **`MYSQL_ENV_MYSQL_PASSWORD`**

5. Write a script to link the containers.

   In this step, you will write a script to start the MySQL container and then start the application container while linking to the MySQL container.

5.1. Edit the **run.sh** file located at **/home/student/DO180/labs/linking-containers/deploy/nodejs/linked/**.

> **Note**
>
> The existing code in the **run.sh** script is for providing an SQL script to initialize the MySQL database when the container is run. In the following steps, append any commands to the end of the file.

5.2. First, append a **docker run** command in order to run the MySQL container:

```
docker run -d --name mysql -e MYSQL_DATABASE=items -e MYSQL_USER=user1 \
-e MYSQL_PASSWORD=mypa55 -e MYSQL_ROOT_PASSWORD=r00tpa55 \
-v $PWD/work/data:/var/lib/mysql/data \
-v $PWD/work/init:/var/lib/mysql/init -p 30306:3306 do180/mysql-56-rhel7
```

In the previous command, the **MYSQL_DATABASE**, **MYSQL_USER**, and **MYSQL_PASSWORD** are populated with the credentials to access the MySQL database. While these are required environment variables for the MySQL container, the variable names are also hard coded into our API.

5.3. Append a **docker run** command to start the API container with the **--link** option to link with the MySQL container. The **--link** option takes the following parameters:

```
--link name:alias
```

The **name** refers to the name of the container being linked to, while the **alias** is the prefix used for the generated environmental variables. In this case, use **mysql** for both values.

Append the following run command to the **run.sh** script:

```
docker run -d --link mysql:mysql --name todoapi -p 30080:30080 \
     do180/todonodejs
```

5.4. After each **docker run** that was inserted into the **run.sh** script, make sure there is also a **sleep 9** command.

5.5. Verify that your **run.sh** script matches the solution script located at **/home/student/DO180/solutions/linking-containers/deploy/nodejs/linked/run.sh**.

5.6. Save the file and exit the editor.

6. Run the linked containers.

6.1. Use the following command to execute the script that you wrote to run the MySQL container, run the Node.js REST API container and link it to the MySQL container, and then run the Apache front-end container:

```
[student@workstation ~]$ cd \
/home/student/DO180/labs/linking-containers/deploy/nodejs/linked
```

```
[student@workstation linked]$ ./run.sh
```

6.2. Verify that the containers all started correctly:

```
[student@workstation linked]$ docker ps
CONTAINER ID        IMAGE                  COMMAND                  CREATED
        STATUS              PORTS                      NAMES
11eac06126f7        do180/todonodejs       "scl enable rh-nodejs"   28 seconds
 ago      Up 26 seconds       0.0.0.0:30080->30080/tcp   todoapi
8371dae9cc68        do180/mysql-56-rhel7   "container-entrypoint"   37 seconds
 ago      Up 36 seconds       0.0.0.0:30306->3306/tcp    mysql
```

7. Examine the environment variables inside the API container.

Run the following command to explore the environment variables that are in the API container:

```
[student@workstation linked]$ docker exec -it todoapi env
```

The following is an example of the expected output:

```
...
MYSQL_PORT=tcp://172.17.0.2:3306
MYSQL_PORT_3306_TCP=tcp://172.17.0.2:3306
MYSQL_PORT_3306_TCP_ADDR=172.17.0.2
MYSQL_PORT_3306_TCP_PORT=3306
MYSQL_PORT_3306_TCP_PROTO=tcp
MYSQL_NAME=/todoapi/mysql
MYSQL_ENV_MYSQL_DATABASE=items
MYSQL_ENV_MYSQL_USER=user1
MYSQL_ENV_MYSQL_PASSWORD=mypa55
MYSQL_ENV_MYSQL_ROOT_PASSWORD=r00tpa55
MYSQL_ENV_container=docker
MYSQL_ENV_MYSQL_VERSION=5.6
...
```

8. Test the application.

8.1. Run a **curl** to test the REST API for the To Do List application:

```
[student@workstation linked]$ curl 127.0.0.1:30080/todo/api/items/1
{"description": "Pick up newspaper", "done": false, "id":1}
```

8.2. Open Firefox on workstation and point your browser to **http://127.0.0.1:30080/todo/** and you should see the To Do List application. The slash (**/**) at the end of the URL is necessary.

8.3. Verify that the correct images were built and that the application is running correctly:

```
[student@workstation linked]$ lab linking-containers grade
```

9. Clean up.

9.1. Stop the running containers:

```
[student@workstation linked]$ cd ~
[student@workstation ~]$ docker stop todoapi mysql
```

9.2. Remove the stopped containers:

```
[student@workstation ~]$ docker rm todoapi mysql
```

9.3. Remove the container images:

```
[student@workstation ~]$ docker rmi do180/todonodejs
[student@workstation ~]$ docker rmi do180/nodejs
[student@workstation ~]$ docker rmi do180/mysql-56-rhel7
```

10. Remove left-over containers and container images from the docker builds:

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

11. 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 new applications using *templates*. A template can include any OCP resource, provided 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 backed by a database. 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, either specifying parameters or allowing them to be automatically generated (for example, a unique DB password) in order to instantiate the list of resources in the template as a cohesive application.

The OCP installer creates several templates by default after installation in the **`openshift`** namespace. View these pre-installed templates by using the **`oc get templates`** command with the **`-n openshift`** option:

```
$ oc get templates -n openshift
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", ❶
        "creationTimestamp": null,
        "annotations": {
            "description": "MySQL database service, with persistent storage...
            "iconClass": "icon-mysql-database",
            "openshift.io/display-name": "MySQL (Persistent)",
            "tags": "database,mysql" ❷
        }
    },
    "message": "The following service(s) have been created ...",
    "objects": [
        {
            "apiVersion": "v1",
            "kind": "Service",
            "metadata": {
                "name": "${DATABASE_SERVICE_NAME}" ❸
            },
            ... Service attributes omitted ...
        },
        {
            "apiVersion": "v1",
            "kind": "PersistentVolumeClaim",
            "metadata": {
                "name": "${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", ❹
            "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}",  5
            "required": true
        },
        {
            "name": "MYSQL_DATABASE",
            "displayName": "MySQL Database Name",
            "description": "Name of the MySQL database accessed.",
            "value": "sampledb",
            "required": true
        },
...
}
```

1. Defines the template name.
2. A list of arbitrary tags that this template will have in the UI.
3. Defines that the name of the service will be the value assigned by the parameter **DATABASE_SERVICE_NAME**.
4. Defines the default value for the parameter **DATABASE_SERVICE_NAME**.
5. 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 OCP 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 templates created under the **openshift** namespace is available in the web console on the "add to project" page.

Parameters

Templates define a set of *parameters*, which are assigned values. OCP 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 be 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
         GENERATOR VALUE
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.    sampledb
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
```

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 above 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

Remember that 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 **-v** option followed by a comma-separated list of **<name>=<value>** pairs:

```
$ oc process -f mysql.json \
     -v MYSQL_USER=dev -v MYSQL_PASSWORD=$P4SSD -v MYSQL_DATABASE=bank \
     -v 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 \
    -v MYSQL_USER=dev,MYSQL_PASSWORD=$P4SSD,MYSQL_DATABASE=bank,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 \
    -v MYSQL_USER=dev -v MYSQL_PASSWORD=$P4SSD -v MYSQL_DATABASE=bank \
    -v VOLUME_CAPACITY=10Gi | oc create -f -
```

The previous command returns an error:

```
error processing the template "mysql-persistent": User "regularUser" cannot create
 processedtemplates in project "openshift"
```

One way to solve this problem is to export the template to a file and then process the template using the file:

```
$ oc -o json export template mysql-persistent \
    -n openshift > mysql-persistent-template.json
```

```
$ oc process -f mysql-persistent-template.json \
    -v MYSQL_USER=dev -v MYSQL_PASSWORD=$P4SSD -v MYSQL_DATABASE=bank \
    -v 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 \
    -v MYSQL_USER=dev -v MYSQL_PASSWORD=$P4SSD -v MYSQL_DATABASE=bank \
    -v 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
```

Note

While the **oc process** command uses **-v** option for parameter values, the **oc new-app** command uses **-p**.

R

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/DO180/labs/openshift-template`
Application URL:	`http://todoapi-template.cloudapps.lab.example.com/todo/`

Outcomes

You should be able to build and deploy an application to OpenShift Container Platform using a provided JSON template.

Before you begin

The workstation needs the To Do List application source code and lab files available and both Docker and the OpenShift cluster need to be running. To download the project 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/DO180/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 ~/DO180/labs/openshift-template/images/mysql
   [student@workstation mysql]$ ./build.sh
   ```

   1.2. Push the image to the private registry.

   To make the image available for OpenShift to use in the template, push it up 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-56-rhel7 \
   infrastructure.lab.example.com:5000/do180/mysql-56-rhel7
   [student@workstation mysql]$ docker push \
   infrastructure.lab.example.com:5000/do180/mysql-56-rhel7
   The push refers to a repository [infrastructure.lab.example.com:5000/do180/
   mysql-56-rhel7]
   b3838c109ba6: Pushed
   a72cfd1d969d: Mounted from rhscl/mysql-56-rhel7
   9ca8c628d8e7: Mounted from rhscl/mysql-56-rhel7
   827264d42df6: Mounted from rhscl/mysql-56-rhel7
   ```

```
latest: digest:
 sha256:170e2546270690fded13f3ced0d575a90cef58028abcef8d37bd62a166ba436b size:
 1156
```

2. Build the parent image for the To Do List app using the Node.js Dockerfile.

   To build the base image, run the **build.sh** script:

   ```
   [student@workstation mysql]$ cd ~/DO180/labs/openshift-template/images/nodejs
   [student@workstation nodejs]$ ./build.sh
   ```

3. Build the To Do List app child image using the Node.js Dockerfile.

> ## Warning
>
> Execute the following commands because they are creating a different container.

   3.1. Build the child image.

   Run the following in order to build the child image:

   ```
   [student@workstation nodejs]$ cd ~/DO180/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, push it to the private registry. To tag and push the image run the following commands in the terminal window:

   ```
   [student@workstation nodejs]$ docker tag do180/todonodejs \
   infrastructure.lab.example.com:5000/do180/todonodejs
   [student@workstation nodejs]$ docker push \
   infrastructure.lab.example.com:5000/do180/todonodejs
   The push refers to a repository [infrastructure.lab.example.com:5000/do180/
   todonodejs]
   8a8bf0b2e036: Pushed
   6d0ce5c0fc9c: Pushed
   22ad2b4aa931: Pushed
   911719963abb: Pushed
   464c51e50340: Pushed
   9ca8c628d8e7: Mounted from do180/mysql-56-rhel7
   827264d42df6: Mounted from do180/mysql-56-rhel7
   latest: digest:
    sha256:fe1d4c2d3834814a5fed91955394e079f2def51b4017e57fb1d96b27992c9512 size:
    1785
   ```

4. Create the persistent volume.

   4.1. Log in as a system 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 that need it.

```
[student@workstation nodejs]$ oc login -u developer -p developer \
https://ocp.lab.example.com:8443
```

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 ~/DO180/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.

Create a new project in OpenShift to use for this exercise. Run the following command to create the project:

```
[student@workstation openshift-template]$ oc new-project template
Now using project "template" on server "https://ocp.lab.example.com:8443".
```

5.2. Set the security policy for the project.

To allow the container to run as the correct user, 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 completed template located at **/home/student/DO180/labs/openshift-template/todo-template.json**. Notice the following resources defined in the template and review their configurations:

- **todoapi** pod to host the running Node.js application
- **mysql** pod to host the MySQL database
- **todoapi** service to provide connectivity to the Node.js application pod
- **mysql** service to provide connectivity to the MySQL database pod
- **dbinit** persistent volume claim for the MySQL **/var/lib/mysql/init** volume
- **db-volume** persistent volume claim for 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 pipe the result into the **oc create** command to create 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. Check the deployment.

Check the status of the deployment using the **oc get pods** command with the **-w** option to continue to monitor the pod status. Wait until the containers are both 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 OCP router and available behind the 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 where the application is available using the **oc status** command by running the following command in the terminal window 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://ocp.lab.example.com:8443

svc/mysql - 172.30.194.34:3306
  pod/mysql runs infrastructure.lab.example.com:5000/do180/mysql-56-rhel7
```

```
http://todoapi-template.cloudapps.lab.example.com to pod port 30080 (svc/
todoapi)
  pod/todoapi runs infrastructure.lab.example.com:5000/do180/todonodejs

2 warnings 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 \
http://todoapi-template.cloudapps.lab.example.com/todo/api/items/1
{"description": "Pick up newspaper", "done": false, "id":1}
```

7.3. Open Firefox on workstation and point your browser to **`http://todoapi-template.cloudapps.lab.example.com/todo/`** and you should see the To Do List 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 your 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 commands 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

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/DO180/labs/deploy-multicontainer`**
Application URL:	**`http://quote-php-deploy.cloudapps.lab.example.com/`**

Outcomes

You should be able to create a multicontainer OpenShift application and access it through a web browser.

Before you begin

The workstation needs the PHP application source code and lab files available and both Docker and the OpenShift cluster need to be running. To download the project 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 ~/DO180/labs/deploy-multicontainer
```

Steps

1. Log in to OCP as the developer user and create a new project for this exercise.

   1.1. From the **`workstation`** VM, log in to OCP as the **`developer`** user.

   1.2. Create a new project in OpenShift, named **`deploy`**, to use for this lab.

   1.3. Relax the default cluster security policy.

   Change the default security policy to allow containers to run as root using the provided **`set-policy.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, 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, push it to the private registry.

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.

> ### 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.

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 then pipe the result into the **`oc create`** command to create an application from the template.

   7.2. Check the deployment.

   Check the status of the deployment using the **`oc get pods`** command with the **`-w`** option to continue to tail the pod status. Wait until the containers are both 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 OCP router and available behind the public FQDN, 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 by running the following command in the terminal window and note the FQDN for the app.

   9.2. Use **`curl`** to test the REST API for the PHP Quote application

   9.3. Verify that the correct images were built and that the application is running correctly:

   ```
   [student@workstation deploy-multicontainer]$ lab deploy-multicontainer grade
   ```

10. Clean up.

    10.1. Delete the project used by this exercise.

10.2. Delete the persistent volumes using the provided shell script.

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

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/DO180/labs/deploy-multicontainer`
Application URL:	`http://quote-php-deploy.cloudapps.lab.example.com/`

Outcomes

You should be able to create a multicontainer OpenShift application and access it through a web browser.

Before you begin

The workstation needs the PHP application source code and lab files available and both Docker and the OpenShift cluster need to be running. To download the project 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 ~/DO180/labs/deploy-multicontainer
```

Steps

1. Log in to OCP as the developer user and create a new project for this exercise.

   1.1. From the **`workstation`** VM, log in to OCP as the **`developer`** user.

   Run the following command in your terminal window:

   ```
   [student@workstation deploy-multicontainer]$ oc login -u developer \
   -p developer https://ocp.lab.example.com:8443
   ```

   If the **`oc login`** command prompts about using insecure connections, answer **`y`** (yes).

   1.2. Create a new project in OpenShift, named **`deploy`**, to use 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://ocp.lab.example.com:8443".
   ```

   1.3. Relax the default cluster security policy.

   Change the default security policy to allow containers to run as root using the provided **`set-policy.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, 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-56-rhel7 \
 infrastructure.lab.example.com:5000/do180/mysql-56-rhel7
[student@workstation mysql]$ docker push \
 infrastructure.lab.example.com:5000/do180/mysql-56-rhel7
The push refers to a repository [infrastructure.lab.example.com:5000/do180/
mysql-56-rhel7]
...
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, 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 \
 infrastructure.lab.example.com:5000/do180/quote-php
[student@workstation quote-php]$ docker push \
 infrastructure.lab.example.com:5000/do180/quote-php
The push refers to a repository [infrastructure.lab.example.com:5000/do180/
quote-php]
9e09d226684b: Pushed
a1f11f24380d: Pushed
db28059b7958: Pushed
c528c3fe2e43: Pushed
d921bad4d903: Pushed
```

```
99c7fad34019: Pushed
b9d38d1b3407: Pushed
9ca8c628d8e7: Layer already exists
827264d42df6: Layer already exists
latest: digest:
 sha256:b150a4c46119c412133f7eab3039b653d15fb903d2f181b9efc71b19366c6236 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** 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 then pipe the result into 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. Check the deployment.

Check the status of the deployment using the **oc get pods** command with the **-w** option to continue to tail the pod status. Wait until the containers are both 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 OCP router and available behind the public FQDN, 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 by running the following command in the terminal window 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
quote-php   quote-php-deploy2.cloudapps.lab.example.com             quote-php
 8080
```

9.2. Use **curl** to test the REST API for the PHP Quote application

```
[student@workstation ~]$ curl http://quote-php-deploy.cloudapps.lab.example.com
Do not take life too seriously. You will never get out of it alive.
```

9.3. Verify that the correct images were built and that the application is running correctly:

```
[student@workstation deploy-multicontainer]$ lab deploy-multicontainer grade
```

10. Clean up.

10.1. Delete the project used by this exercise.

Run the following commands in your terminal window:

```
[student@workstation deploy-multicontainer]$ oc delete project deploy
```

10.2.Delete the persistent volumes using the provided shell script.

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 $(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 **`--link`** option from the **`docker run`** command injects into a container environment variables from other containers as arguments. Containers must be manually started in the correct order and with the correct values for the **`--link`** option.

- 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

Overview	
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.

The S2I Process

S2I is a simple way to create images based on programming languages in OpenShift. Unfortunately, problems may arise during the S2I image creation process, either by the programming language characteristics or the run time environment that requires both developer and administrators to work together.

Initially, 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:

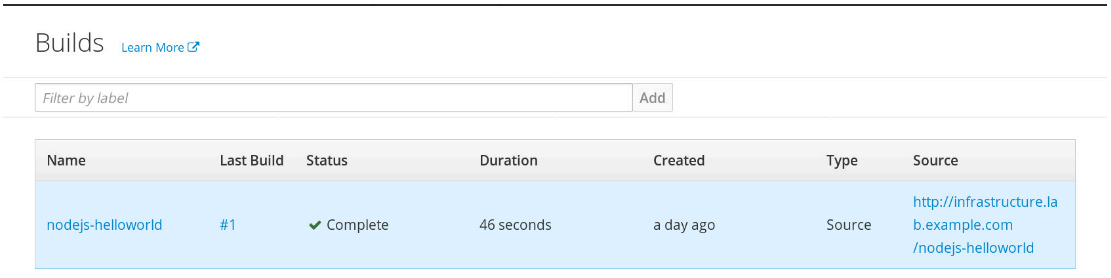
- *Build step*: Responsible for compiling source code, downloading library dependencies, and packaging the application as a Docker image. Furthermore, the build step pushes the image to the OpenShift registry for the deployment step. The build step is managed by the BuildConfig (BC) from OpenShift.
- *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) from OpenShift.

In S2I, every application has its own BuildConfig (BC) and DeploymentConfig (DC). The BuildConfig (BC) and DeploymentConfig (DC) names match the application name. The deployment process is aborted if the build fails.

The S2I process starts each step in a separate pod. Therefore, the build step creates a pod named **`<application-name>-build-<number>-<string>`**. On each build attempt, the whole build step is executed and a log is stored. After a successful execution, the application is started on a separate pod, normally named as **`<application-name>-<string>`**.

OpenShift has a rich web console listing the isolated steps mentioned previously by project.

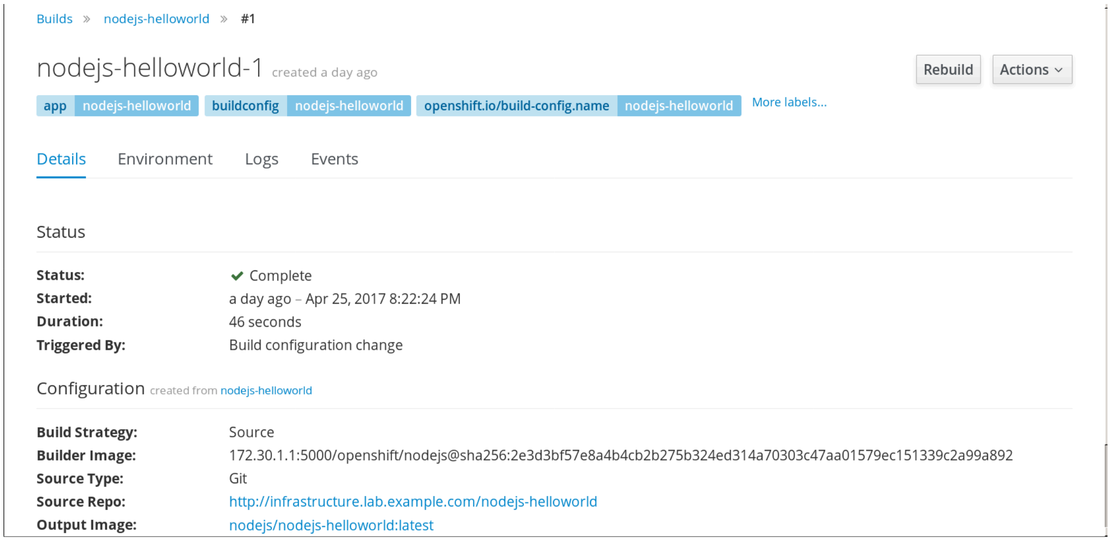
To identify any build issues, the logs from a build can be evaluated using the **Builds** link from the left panel depicted as follows.



Name	Last Build	Status	Duration	Created	Type	Source
nodejs-helloworld	#1	✓ Complete	46 seconds	a day ago	Source	http://infrastructure.lab.example.com/nodejs-helloworld

Figure 8.1: Detailed vision of a project

On each attempt to build, a history of the build is provided for administrator evaluation.


Builds » nodejs-helloworld » #1
nodejs-helloworld-1 created a day ago
Rebuild
Actions
app nodejs-helloworld
buildconfig nodejs-helloworld
openshift.io/build-config.name nodejs-helloworld
More labels...
Details
Environment
Logs
Events

Status

Status: ✓ Complete
Started: a day ago – Apr 25, 2017 8:22:24 PM
Duration: 46 seconds
Triggered By: Build configuration change

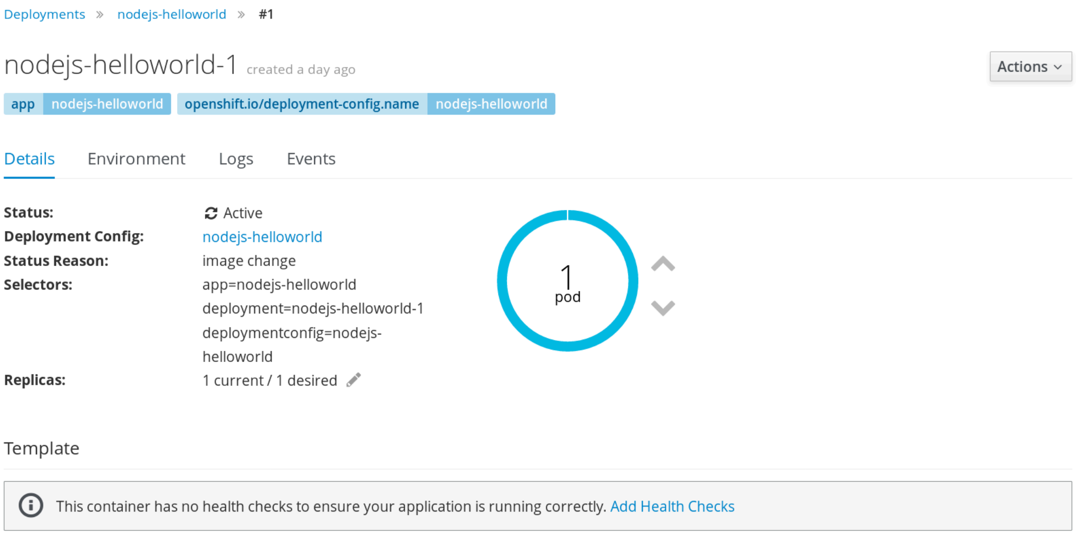
Configuration created from nodejs-helloworld

Build Strategy: Source
Builder Image: 172.30.1.1:5000/openshift/nodejs@sha256:2e3d3bf57e8a4b4cb2b275b324ed314a70303c47aa01579ec151339c2a99a892
Source Type: Git
Source Repo: http://infrastructure.lab.example.com/nodejs-helloworld
Output Image: nodejs/nodejs-helloworld:latest

Figure 8.2: Detailed vision of a build config

On the other hand, to identify issues during the deployment step, the Applications link from the left panel can be used.

On each pod deployed, the logs can be obtained accessing the **Applications** > **Deployments** from the left panel


Deployments » nodejs-helloworld » #1
nodejs-helloworld-1 created a day ago
Actions
app nodejs-helloworld
openshift.io/deployment-config.name nodejs-helloworld
Details
Environment
Logs
Events
1 pod
Template

Status: Active
Deployment Config: nodejs-helloworld
Status Reason: image change
Selectors: app=nodejs-helloworld deployment=nodejs-helloworld-1 deploymentconfig=nodejs-helloworld
Replicas: 1 current / 1 desired

This container has no health checks to ensure your application is running correctly. Add Health Checks

Figure 8.3: Detailed vision of a deployment config

To identify problems using the **oc** command-line interface, the logs command may be used. Likewise in the web interface, it has a set of commands which provides information about each step. For example, to get the logs from the a build configuration, the following command may be used:

```
$ oc logs bc/<application-name>
```

If a build fails, the bc must be restarted. To request a new build, the following command may be used:

```
$ oc start-build <application-name>
```

A new pod with the build process is automatically started.

After a successful build, a pod is started, in which the application and the deployment process is executed.

Common Problems

Sometimes, the source code requires some customization that may not be available in containerized environments. For instance, database credentials, file system access, and message queue information, normally provided as internal environment variables, may need to be changed.

The **oc logs** command provides important outputs about the build, deploy, and run process of an application during the execution of a pod. The logs normally indicate missing values or options that must be enabled, incorrect parameters, invalid flags, or environment incompatibilities.

Note

Application logs must be clearly labeled to identify problems quickly without the need to learn the container internals.

Permission issues

OpenShift runs S2I containers using RHEL as the base image. so any run time difference may imply in a run time error. The usual problem identified is the permission denied error due to wrong permission used by developers or incorrect environment permissions set by administrators. For example, S2I images enforce the use of a different user than root to access file systems and external resources. Also, RHEL7 works with an enforcing SELinux policy that blocks any access to file system resources, network port access, or process access.

Important

SELinux on the Container Development Kit (CDK) VM is set as permissive which is different from the RHEL 7 and OCP standard policies.

Some containers may require a specific user ID, but S2I is designed to run containers using a random user as per the default OpenShift security policy. To fix this condition, relax the OpenShift project security with the command **`oc adm policy`**. As an example, see the **`setpolicy.sh`** script used on some of the previous labs allows the user defined in the **`Dockerfile`** file to run the application.

Invalid parameters

Multi-container applications may share parameters, such as login credentials. Verify that the same values for parameters are passed to all containers in the application. For example, a Python application running in one container links with a container running a database. Both containers must use the same user name and password for the database. Normally, logs from the application pod provide a clear idea of these problems and how to solve them.

Volume mount errors

When redeploying an application that uses a persistent volume on a local file system, a pod may have problems allocating a persistent volume claim even though the persistent volume indicates that the claim is released. To solve it, the persistent volume claim and the persistent volume must be deleted, in this order. Then the persistent volume must be recreated.

Obsolete images

OpenShift pulls images from the source named in an image stream unless it finds 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`**. Check also the **`oc adm prune`** command for more automated ways 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:

> Creating Images
> https://docs.openshift.com/container-platform/3.5/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.cloudapps.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

OpenShift must be installed and running on the **ocp** VM.

Retrieve the lab files and verify that Docker and OpenShift are up and running, by executing the lab script:

```
[student@workstation ~]$ lab bc-and-dc setup
```

Steps

1. Log in to OpenShift and create a new project named **nodejs**:

```
[student@workstation ~]$ oc login -u developer https://ocp.lab.example.com:8443
Logged into "https://ocp.lab.example.com:8443" as "developer" using existing
 credentials.
...
[student@workstation ~]$ oc new-project nodejs
Now using project "nodejs" on server "https://ocp.lab.example.com:8443".
...
```

   You may have to enter credentials for the **developer** account. The password is **developer**. The current project in your command output may differ from the listing above.

2. Create a new Node.js application using Source-to-Image from the git repository at **http://infrastructure.lab.example.com/nodejs-helloworld**.

   2.1. Use 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://infrastructure.lab.example.com:8081/nexus/content/groups/nodejs/ \
 nodejs:4~http://infrastructure.lab.example.com/nodejs-helloworld
```

   The **--build-env** option defines an environment variable to the builder pod that makes the **npm** command, which is run as part of the build, to fetch NPM modules from the Nexus server in the **infrastructure** VM.

Important

In the previous command, there should be no spaces between **registry=** and \. There should also be no spaces before **http:**. But there is an space before **nodejs:4**

2.2. Wait until the application finishes building by monitoring the progress with the **oc get pods -w** command:

```
[student@workstation ~]$ oc get pods -w
NAME                         READY     STATUS    RESTARTS   AGE
nodejs-helloworld-1-build    0/1       Error     0          35s
^C
```

The build process failed and therefore no application is running. Build failures are usually consequence 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 is started. By default, the OpenShift S2I process creates a bc named **nodejs-helloworld**, which is responsible for triggering the build process. Run this command in a terminal window to confirm 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://infrastructure.lab.example.com/nodejs-helloworld" ...
 Commit: 87edb8c3266d4d04a5948476350a9b1fdb6ee439 (fixed)
...
---> Installing application source ...
---> Building your Node application from source
...
npm ERR! notarget No compatible version found: express@'>=4.14.2 <4.15.0'
npm ERR! notarget Valid install targets:
npm ERR! notarget ["0.14.0","...,"4.14.0",...]
...
error: build error: non-zero (13) exit code from 172.30.1.1:5000/openshift/
nodejs@sha256:2e3d3bf57e8a4b4cb2b275b324ed314a70303c47aa01579ec151339c2a99a892
```

As the output indicates the format used by the express dependency is not valid.

3. Fix the build process from the project.

The developer uses a non-standard 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** > **Favorites** > **Terminal**) and run the following command:

```
[student@workstation ~]$ git clone \
http://infrastructure.lab.example.com/nodejs-helloworld
```

The source code from the application is cloned locally in the **`~/nodejs-helloworld`** folder.

3.2. Evaluate the **`package.json`**.

Using your preferred editor, open the **`~/nodejs-helloworld/package.json`** file. Check the dependencies versions provided by the developers. It uses an incorrect version for the Express dependency, which is incompatible with the supported version provided by the company (~4.14.2). Update the dependency version to become:

```
    "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 ~]$ 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 http://infrastructure.lab.example.com/nodejs-helloworld
   87edb8c..53d87dc  master -> master
```

4. Restart 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 using the following command:

```
[student@workstation nodejs-helloworld]$ oc logs -f bc/nodejs-helloworld
```

The following output is displayed:

```
Cloning "http://infrastructure.lab.example.com/nodejs-helloworld" ...
```

```
...
Push successful
```

The build was successful. However, it does not mean the application is started.

4.2. Evaluate the status of the current build process. Run the 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 was successful, but the application pod had errors during start up.

```
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. Check the log 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 from the previous step.

The expected output is:

```
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 failed to start up because a script is missing.

5. Fix the problem with starting the application pod.

5.1. Fix the **package.json** file to define a the start up command.

From the previous output, the **~/nodejs-helloworld/package.json** file is missing the **scripts** attribute the **start** field. According to the developers, a minimal effort was made to adapt the application to run as a Docker image, by running working with the community Docker image. To run the community Docker image locally, use the following command:

docker run -it --rm --name my-app -v "$PWD":/usr/src/app -w /usr/src/app node:4 node app.js

According to the command provided by the developers, the **node app.js** must be added to the start field.

To fix the problem, add to the **package.json** file the following attribute:

```
...
  "main": "app.js",
  "scripts": {
    "start": "node app.js"
  },
...
```

5.2. 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 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 http://infrastructure.lab.example.com/nodejs-helloworld
   cbeb4f1..20b2fe8  master -> master
```

Continue the deploy step from the S2I process.

5.3. To restart the build step, execute the following command.

```
[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 check 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 was successful, and this time the application was started without errors.

```
NAME                          READY     STATUS      RESTARTS   AGE
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. Check the log 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 from the previous step.

The expected output is:

```
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!
...
```

This time, the application is running on port 8080.

6. Test the application.

   6.1. Run the following command to expose the application:

   ```
   [student@workstation apps]$ oc expose svc/nodejs-helloworld
   route "nodejs-helloworld" exposed
   ```

   6.2. Check the address associated with this application:

   ```
   [student@workstation ~]$ oc get route
   NAME                HOST/PORT                                        PATH
     SERVICES
   nodejs-helloworld   nodejs-helloworld-nodejs.cloudapps.lab.example.com
     nodejs-helloworld
   ```

   6.3. Access the application from the workstation VM.

   From the command line access the workstation:

   ```
   [student@workstation ~]$ curl \
   http://nodejs-helloworld-nodejs.cloudapps.lab.example.com
   ```

   The expected output is:

   ```
   Hello world!
   ```

7. Verify that the application was correctly set up. Run the following grading script from a terminal window:

   ```
   [student@workstation ~]$ lab bc-and-dc grade
   ```

8. Delete the project, which 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 users have the port-forwarding feature provided by the **-p** option from **docker run**. 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 a port-forwarding mapping from the host to a database server running inside a container:

```
$ docker run --name db -p 30306:3306 mysql
```

The above command maps host port 30306 to port 3306 on the container named **db**. This container was created from the **mysql** image, which starts a MySQL server that accepts network connections on port 3306.

OpenShift provides the **oc port-forward** command that forwards 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, while a port-forwarding mapping forwards connections to a single pod.

Here is an example of the **oc port-forward** command syntax:

```
$ oc port-forward db 30306 3306
```

The above command forwards port 30306 from the developer workstation to port 3306 on the **db** pod, where a MySQL server (inside a container) accepts network connections.

Note

Be sure to leave this terminal window running. Closing the window or cancelling the process will stop the port-forwarding mapping.

While the **docker run -p** port-forwarding mapping can only be configured when the container is started, the **oc port-forward** 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 using **docker run -p**. But most administrators prefer not using **NodePort** with databases to avoid exposing the database server to direct connections from users. In this case, a port-forwarding mapping is considered a more secure alternative.

Using oc port-forward for Debugging Java 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"
```

Once the server is started with the debugger listening on port 8787, a **port-forwarding** map needs to be created to forward connections from a local unused TCP port to the 8787 port 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 a image previously configured to enable remote debugging:

```
$ oc port-forward jappserver 8787:8787
```

Once the debugger is enabled and the port forwarding-mapping is running, 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 just 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 get 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 lone container and return the output.

Docker and OpenShift Events

Some developers find Docker and OpenShift logs are too low-level, and digging for useful troubleshooting information may not be trivial. Fortunately, both provide a higher-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, for example:

```
$ docker events --since=10m
```

The **--since** command option allows specifying a time stamp as an absolute date and time string or as a time interval. The previous example shows events generated during the last 10 minutes.

OpenShift events

To show OpenShift events, use the **get** verb with the **ev** resource type for the **oc** command, for example:

```
$ 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 like **grep** can help, but there is a more focused way to show OCP cluster events: describing an OCP resource shows events related to that resource only.

For example, to list only events related to a pod named **mysql**, use:

```
$ oc describe pod mysql
```

Accessing Running Containers

While using the **docker logs** and **oc logs** commands can be useful for viewing any output sent by a container, it does not necessarily display all of the available debugging 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; for example, to verify external network connectivity.

Both Docker and OpenShift provide an **exec** command that allows creating new processes inside a running container, and have these process standard output and input redirected to the user terminal. The following is the general syntax for the **docker exec** command:

```
$ docker exec [options] container command
```

This is the general syntax for the **oc exec** command:

```
$ oc exec [options] pod [-c container] command
```

To execute a single interactive command or start a shell, add the **-it** options. The following example starts a Bash shell on a pod named **myhttpdpod**:

```
$ oc exec -it myhttpdpod bash
```

Users can use this command to access application logs saved to disk (as part of the container ephemeral storage). For example, the following command displays the Apache error log from a container named **apache-container**:

```
$ 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. This is done to keep the images smaller and allows running more containers per host.

One technique to temporarily provide some of these missing commands, such as **ping** and **dig**, 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** does not require matching **VOLUME** instructions to be present in the image **Dockerfile**.

Note

To obtain a similar effect using OpenShift, it would be necessary to change the pod resource definition to add more **volumeMounts** and **volumeClaims**. It would also be necessary to create PV resources of kind **hostPath**. As any container image that runs as a Kubernetes pod could be run as a simple Docker container, there will be no OpenShift specific examples for this topic.

The following command starts a container, overriding the image **/bin** folder with the one from the host, and starts an interactive shell inside that container:

```
$ docker run -it -v /bin:/bin image /bin/bash
```

Note

Which directory of binaries you need to override depends on the base OS image. For example, some commands require shared libraries from **`/lib`**. Another example: some Linux distributions have different contents in **`/bin`** and **`/usr/bin`**, or **`/lib`** and **`/usr/lib`**, requiring multiple **`-v`** options.

An alternative to mounting host binaries folders is to provide troubleshooting commands as part of the container image. This can be done by adding instructions to install the desired commands to the image **`Dockerfile`**. For example, examine the following partial **`Dockerfile`**, which is a child of the **`rhel7.3`** image used throughout this course and adds commonly used network troubleshooting commands:

```
FROM rhel7.3

RUN yum install -y \
less \
dig \
ping \
iputils && \
 yum clean all
```

When built and run, this container will be identical to a **`rhel7.3`** container, but it will also have several additional troubleshooting tools available.

Getting Files Into and Out of Containers

When troubleshooting or managing an application, it may be necessary to move files into and out of running containers, such as configuration files or log files. There are several ways to move files into and out of containers:

`docker cp`

As of Docker version 1.8, 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`**, the syntax looks like the following:

```
$ 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 .
```

This **`docker cp`** alternative has the advantage of working with containers that were already started, while the next 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 using volume mounts. Users can mount a local directory to copy data into a container. For example, the following command sets the host **`/conf`** directory as the volume to use for the Apache configuration directory. This creates a simple 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
```

Using the same concept, it is possible to pull data from a running container and place it onto the host machine. A useful example of this is using the **mysqldump** utility to create a backup of a MySQL database inside a container. For example:

```
$ 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' \
 > items_dump.sql
```

The previous command uses the container environment variables to connect to the MySQL server and execute the **mysqldump** 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 MySQL administration commands on the host.

The **oc rsync** command provides functionality similar to **docker cp** for containers running under OpenShift pods.

Demonstration: Forwarding Ports

1. Open an SSH session to the **ocp** VM and verify that the OpenShift cluster is running:

```
[student@workstation ~]$ ssh ocp
[student@ocp ~]$ ./ocp-up.sh
```

The **ocp-up.sh** script checks that the OCP cluster is running and, if true, exits without displaying anything, else it starts the cluster.

2. Log out from the **ocp** VM and return to the workstation VM:

```
[student@ocp ~]$ Ctrl+D
Connection to ocp closed.
[student@workstation ~]$
```

3. Log in to OCP as the **developer** user and create a new project for this demonstration:

```
[student@workstation ~]$ oc login -u developer -p developer \
https://ocp.lab.example.com:8443
[student@workstation ~]$ oc new-project port-forward
```

If the **oc login** command prompts about using insecure connections, answer **y** (yes).

4. Create a new application from the **rhscl/mysql-56-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**) using the **-e** option.

   Use the **--docker-images** option for **oc new-app** and 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=infrastructure.lab.example.com:5000/rhscl/mysql-56-rhel7:latest \
--insecure-registry=true --name=port-forwarding \
-e MYSQL_USER=user1 -e MYSQL_PASSWORD=mypa55 -e MYSQL_DATABASE=testdb \
-e MYSQL_ROOT_PASSWORD=r00tpa55
```

5. Run the **oc status** command to view the status of the new application, and to check if the deployment of the MySQL image was successful:

```
[student@workstation ~]$ oc status
In project port-forward on server https://ocp.lab.example.com:8443

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
```

6. To allow clients running in the developer's machine (in this environment, the **workstation** VM) to access the database server from outside the OCP cluster, forward a local port to the pod. This can be achieved 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
Forwarding from [::1]:13306 -> 3306
```

Important

The **oc port-forward** command does not return to the Bash shell. Leave this terminal open, and then open another terminal for the next steps.

7. Open a new terminal window on the **workstation** VM and 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
...
Server version: 5.6.34 MySQL Community Server (GPL)
...
```

8. Verify if the **testdb** database has been created:

```
MySQL [(none)]> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| testdb             |
+--------------------+
2 rows in set (0.00 sec)
```

9. Exit from the MySQL prompt:

```
MySQL [(none)]> exit
Bye
```

10. Press **Ctrl**+**C** in the terminal window where you started the **oc port-forward** command. This stops the port-forwarding and prevents further access to the MySQL database from the developer's machine.

11. 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:

> Architecture
> https://access.redhat.com/documentation/en-us/openshift_container_platform/3.5/html/architecture/

The CLI commands for port-forwarding are available in the *Port Forwarding* chapter 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: 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/DO180/labs/debug-httpd**
Application URL:	NA
Resources	CentOS httpd image (**httpd:2.4**)

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

The workstation should have docker running. To check if this is true and to download the required lab files, run the following command from a terminal window:

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

Steps

1. Configure Apache httpd to send log messages to stdout and change the default log level.

   1.1. The default log level for the CentOS Apache httpd container image is **warn**. You can change the default level for httpd, and re-direct log messages to **stdout** by overriding the default **httpd.conf** file. You can do this by creating a custom image on the host workstation VM.

   Briefly review the custom **httpd.conf** file at **/home/student/DO180/labs/debug-httpd/conf/httpd.conf**. Observe the **ErrorLog** directive in this file:

   ```
   ErrorLog /dev/stdout
   ```

   This directive sends the httpd error log messages to the container's **stdout**.

   Observe the **LogLevel** directive in this file:

   ```
   LogLevel debug
   ```

   This directive changes the default log level to **debug**.

   Observe the **CustomLog** directive in this file:

   ```
   CustomLog /dev/stdout common
   ```

   This directive redirects the httpd access log messages to the container's **stdout**.

2. Build a custom container to add the configuration file to the container.

   2.1. From the terminal window, run the following commands:

   ```
   [student@workstation ~]$ cd DO180/labs/debug-httpd
   [student@workstation DO180/labs/debug-httpd]$ ./build.sh
   ```

   2.2. Verify that the image was created.

   From the terminal window, run the following command:

   ```
   [student@workstation DO180/labs/debug-httpd]$ docker images
   ```

   The new image must be available in the local docker cache.

   ```
   REPOSITORY      TAG  IMAGE ID            ...
   debug-httpd  latest  c86936c8e791        ...
   ```

3. Start the custom httpd container:

   ```
   [student@workstation ~]$ docker run \
        --name debug-httpd -d \
        -p 10080:80 debug-httpd
   ```

4. View 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 debug-httpd
   [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 that debug level messages are now visible.

   4.2. Request the home page of the web server using **curl**:

   ```
   [student@workstation ~]$ curl http://127.0.0.1:10080
   <html><body><h1>It works!</h1></body></html>
   ```

   4.3. Use the **docker logs** command again to view the access log of the httpd web server:

```
[student@workstation ~]$ docker logs debug-httpd
[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. Verify the latest events from the docker daemon, related to previous lab steps using the **`docker events`** command.

Adjust the time interval specified as an argument for the **`--since`** option to the approximate time elapsed since you started this lab:

```
[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-httpd:latest)
2017-04-28T07:41:51.835352650-04:00 container create ... image=debug-httpd,
 name=debug-httpd)
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-httpd,
 name=debug-httpd)
```

Note

Your output may be different compared to the above. The **`docker events`** command will NOT return to the prompt unless you kill the command with **`Ctrl`**+**`C`**.

5. Verify if the debug configuration for the container was correctly set up. Run the following from a terminal window:

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

6. Delete the container and image used in this lab.

```
[student@workstation ~]$ docker stop debug-httpd
[student@workstation ~]$ docker rm debug-httpd
[student@workstation ~]$ docker rmi -f httpd:2.4 debug-httpd
```

This concludes the guided exercise.

Lab: Troubleshooting Containerized Applications

In this lab, you will configure the **broken-httpd** container to send the logs to the **stdout**. This container runs an application that has a broken link to download a file.

Resources	
Files	**/home/student/DO180/labs/troubleshooting-lab**
Application URL	**http://localhost:30000**
Resources	Broken httpd container image (**do180/broken-httpd**)

Outcomes

You should be able to send the Apache server logs to the **stdout** and fix a containerized application from the container that is not working as planned.

Before you begin

The workstation should have the required files to build a broken container. To achieve this goal, run the following command from a terminal window:

```
[student@workstation ~]$ lab troubleshooting-lab setup
```

The previous command downloads the following files:

- **Dockerfile**: File responsible for building the container image.
- **training.repo**: File with the local yum repositories available for the classroom environment.
- **httpd.conf**: File responsible for configuring the Apache HTTP server to send the logs to the **stdout**.
- **src/index.html**: The application file.

1. A custom **httpd** configuration file is already provided to send the logs to the **stdout**. Check the **Dockerfile** file to confirm that the default configuration file will be replaced by the customized file. These files are available in the **/home/student/DO180/labs/troubleshooting-lab** folder.

2. Build the new container image, tagging it as **do180/broken-httpd**.

3. The **do180/broken-httpd** container image has a simple web page containing a link that should download a file from the following URL:

   http://materials.example.com/labs/archive.tgz

   Open a web browser and download the file to see that the previous URL is correct.

4. To troubleshoot the application, start a new container with the following characteristics:

   - **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 **30000** host port to **80** container port.

Note

The volume mount is responsible for sharing commands, such as **ping**, from the **/usr/bin** host folder with the container, to help in the troubleshooting.

5. Verify that the httpd daemon logs from the **broken-httpd** container were forwarded to the **stdout**.

6. Open a new web browser and navigate to this URL: **http://localhost:30000**.

Warning

The final period is not part of the URL.

Click the **Download the file** link. You should see a **server not found** message.

7. Troubleshoot the container.

   7.1. Access the container shell to start troubleshooting.

   7.2. Use the **ping** command to verify whether the host name from the HTML link is accessible.

   7.3. Edit the **index.html** file to fix the problem.

   7.4. Exit the container.

   ### Warning

   In a real-world scenario, the correct approach would be building a new container image with the fixed page. The fix inside the container is just a troubleshooting step to confirm the error.

8. Switch back to the web browser and refresh the page.

   Click the **Download the file** link. The download should start.

9. Check your work. Run the following from a terminal window:

```
[student@workstation troubleshooting-lab]$ lab troubleshooting-lab grade
```

10. Clean up: Delete all containers and images created by this lab.

This concludes the lab.

Solution

In this lab, you will configure the **broken-httpd** container to send the logs to the **stdout**. This container runs an application that has a broken link to download a file.

Resources	
Files	**/home/student/DO180/labs/troubleshooting-lab**
Application URL	**http://localhost:30000**
Resources	Broken httpd container image (**do180/broken-httpd**)

Outcomes

You should be able to send the Apache server logs to the **stdout** and fix a containerized application from the container that is not working as planned.

Before you begin

The workstation should have the required files to build a broken container. To achieve this goal, run the following command from a terminal window:

```
[student@workstation ~]$ lab troubleshooting-lab setup
```

The previous command downloads the following files:

- **Dockerfile**: File responsible for building the container image.
- **training.repo**: File with the local yum repositories available for the classroom environment.
- **httpd.conf**: File responsible for configuring the Apache HTTP server to send the logs to the **stdout**.
- **src/index.html**: The application file.

1. A custom **httpd** configuration file is already provided to send the logs to the **stdout**. Check the **Dockerfile** file to confirm that the default configuration file will be replaced by the customized file. These files are available in the **/home/student/DO180/labs/troubleshooting-lab** folder.

   The following Dockerfile is expected:

   ```
   FROM rhel:7.3
   ...
   COPY httpd.conf /etc/httpd/conf/
   ...
   ```

2. Build the new container image, tagging it as **do180/broken-httpd**.

   ```
   [student@workstation ~]$ cd DO180/labs/troubleshooting-lab
   [student@workstation troubleshooting-lab]$ docker build -t do180/broken-httpd .
   ```

3. The **do180/broken-httpd** container image has a simple web page containing a link that should download a file from the following URL:

   http://materials.example.com/labs/archive.tgz

Open a web browser and download the file to see that the previous URL is correct.

4. To troubleshoot the application, start a new container with the following characteristics:

   - **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 **`30000`** host port to **`80`** container port.

> ### Note
>
> The volume mount is responsible for sharing commands, such as **`ping`**, from the **`/usr/bin`** host folder with the container, to help in the troubleshooting.

```
[student@workstation troubleshooting-lab]$ docker run --name broken-httpd -d \
-p 30000:80 -v /usr/bin:/usr/bin do180/broken-httpd
```

5. Verify that the httpd daemon logs from the **`broken-httpd`** container were forwarded to the **`stdout`**.

```
[student@workstation troubleshooting]$ docker logs broken-httpd
```

The expected output is similar to:

```
...
[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] [lbmethod_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
...
```

6. Open a new web browser and navigate to this URL: **`http://localhost:30000`**.

> ### Warning
>
> The final period is not part of the URL.

Click the **Download the file** link. You should see a **`server not found`** message.

7. Troubleshoot the container.

   7.1. Access the container shell to start troubleshooting.

```
[student@workstation troubleshooting-lab]$ docker exec -it broken-httpd \
/bin/bash
```

7.2. Use the **ping** command to verify whether the host name from the HTML link is accessible.

This web page opened in the web browser is available in the **/var/www/html** folder.

Display the **index.html** file contents:

```
[root@9ef58f71371c /]# cat /var/www/html/index.html
<html>
<body>
<h1>Download application</h1>
<br/>
<a href="http://materiasl.example.com/troubleshooting/archive.tgz">Download the
 file</a>
</body>
</html>
```

Copy the host part of the URL and ping it:

```
[root@9ef58f71371c /]# ping materiasl.example.com
```

The following output is expected:

```
ping: materiasl.example.com: Name or service not known
```

7.3. Edit the **index.html** file to fix the problem.

The problem is related to a typo. Fix the typo replacing from **materiasl** to **materials** in the **index.html** file. The **vi** editor is available because the **/usr/bin** host folder was mounted as a volume.

```
[root@9ef58f71371c /]# vi /var/www/html/index.html
```

7.4. Exit the container.

```
[root@9ef58f71371c /]# exit
```

Warning

In a real-world scenario, the correct approach would be building a new container image with the fixed page. The fix inside the container is just a troubleshooting step to confirm the error.

8. Switch back to the web browser and refresh the page.

   Click the **Download the file** link. The download should start.

9. Check your work. Run the following from a terminal window:

```
[student@workstation troubleshooting-lab]$ lab troubleshooting-lab grade
```

10. Clean up: Delete all containers and images created by this lab.

```
[student@workstation troubleshooting-lab]$ cd ~
[student@workstation ~]$ docker stop broken-httpd
[student@workstation ~]$ docker rm broken-httpd
[student@workstation ~]$ docker rmi -f $(docker images -q)
```

This concludes the lab.

Summary

In this chapter, you learned:

- S2I images requires that security concerns must be addressed to minimize build and deployment issues.
- Development and sysadmin teams must work together to identify and mitigate problems due 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

Overview	
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 OCP.
- 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 Docker link.
- 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: **system: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 most 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 you mount any volumes on the Docker and OpenShift host, you need 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 using Docker, and deploy on an OpenShift cluster.

Outcomes

You should be able to:

- Construct 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.

Before you begin

Get a copy of the lab files:

```
[student@workstation ~]$ lab review setup
```

The lab files are located in the **/home/student/DO180/labs/review** directory. The solution files are located in the **solutions** 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**. This file can be retrieved from the server at **http://content.example.com/ocp3.4/x86_64/installers/{tarball name}**. A script has been provided to retrieve the tar bundle before you build the image: **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 lab file directory, **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 to the OpenShift cluster:

- Publish the container image to the classroom private registry at **infrastructure.lab.example.com:5000**.
- The **deploy/openshift** lab files 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 used to deploy the Nexus server container image.
- Several helpful scripts are located in the solutions 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 **infrastructure.lab.example.com:5000**. The container must be named **nexus** and have a tag **latest**. The OpenShift template expects this name.
- A suggestion for the OpenShift project name is **review**. Make sure you execute the **setpolicy.sh** shell script after you create the OpenShift project. This allows the container to run as the user with which it was configured to run as.
- Be 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.
- Use the grading script to test your work:

  ```
  [student@workstation review]$ lab review grade
  ```

- Clean up your project space by deleting the OpenShift project. You may want to remove the local copy of the Docker container image as well.

Evaluation

After deploying the Nexus server container image to the OpenShift cluster, you can check your work by running the lab grading script:

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

Solution

In this review, you will containerize a Nexus Server, build and test using Docker, and deploy on an OpenShift cluster.

Outcomes

You should be able to:

- Construct 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.

Before you begin

Get a copy of the lab files:

```
[student@workstation ~]$ lab review setup
```

The lab files are located in the **`/home/student/DO180/labs/review`** directory. The solution files are located in the **`solutions`** 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`**. This file can be retrieved from the server at **`http://content.example.com/ocp3.4/x86_64/installers/{tarball name}`**. A script has been provided to retrieve the tar bundle before you build the image: **`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 lab file directory, **`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 to the OpenShift cluster:

- Publish the container image to the classroom private registry at **infrastructure.lab.example.com:5000**.

- The **deploy/openshift** lab files 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 used to deploy the Nexus server container image.

- Several helpful scripts are located in the solutions 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 **infrastructure.lab.example.com:5000**. The container must be named **nexus** and have a tag **latest**. The OpenShift template expects this name.

- A suggestion for the OpenShift project name is **review**. Make sure you execute the **setpolicy.sh** shell script after you create the OpenShift project. This allows the container to run as the user with which it was configured to run as.

- Be 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.

- Use the grading script to test your work:

  ```
  [student@workstation review]$ lab review grade
  ```

- Clean up your project space by deleting the OpenShift project. You may want to remove the local copy of the Docker container image as well.

Steps

1. Develop a **`Dockerfile`** that properly containerizes the Nexus server. Use the lab working directory **`image`**. Use the comments in the existing file to guide you in the development of this container image.

    1.1. Specify the base image to use:

    ```
    FROM rhel7.3
    ```

    1.2. Enter your name and email as the maintainer:

    ```
    FROM rhel7.3
    MAINTAINER New User <user1@myorg.com>
    ```

    1.3. Set the environment variables for **`NEXUS_VERSION`** and **`NEXUS_HOME`**:

    ```
    FROM rhel7.3
    MAINTAINER Jane Doe <jdoe@myorg.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 the snippet provided in **`installjava`**:

    ```
    ...
    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
    ```

    1.5. Create the server home directory and service account/group using the snippet provided in **`createnexusacct`**:

    ```
    ...
    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}
    ```

    1.6. Install the Nexus server software at **`NEXUS_HOME`** and add the start-up script. Note that the **`ADD`** directive will extract tarballs. Create the **`nexus2`** link pointing to the Nexus server directory.

```
...
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}
```

1.7. Make the container run as the *nexus* user and make the working directory **`/opt/nexus`**:

```
...
RUN ln -s ${NEXUS_HOME}/nexus-${NEXUS_VERSION} \
          ${NEXUS_HOME}/nexus2 && \
    chown -R nexus:nexus ${NEXUS_HOME}

USER nexus
WORKDIR ${NEXUS_HOME}
```

1.8. Define a volume mount point to hold the Nexus server persistent data:

```
...
USER nexus
WORKDIR ${NEXUS_HOME}

VOLUME ["/opt/nexus/sonatype-work"]
```

1.9. Execute the Nexus server shell script. The completed **`Dockerfile`** is listed here:

```
FROM rhel7.3
MAINTAINER Jim Rigsbee <jrigsbee@redhat.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. Get the Nexus server tarball into the **image** directory using the script **image/get-nexus-bundle.sh**. Build the container image:

```
[student@workstation ~]$ cd /home/student/DO180/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 **deploy/docker/run-persistent.sh** shell script:

```
[student@workstation docker]$ ./run-persistent.sh
[student@workstation docker]$ docker logs -f {container id} # wait for Started
 message in log
[student@workstation docker]$ docker inspect {container id} # look for IP address
[student@workstation docker]$ curl {ip address}:8081/nexus/
[student@workstation docker]$ docker kill {container id}
```

4. Publish the Nexus server container image to the classroom private registry:

```
[student@workstation docker]$ docker tag nexus:latest \
infrastructure.lab.example.com:5000/nexus:latest
[student@workstation docker]$ docker push \
infrastructure.lab.example.com:5000/nexus:latest
```

5. Deploy the Nexus server container image to the OpenShift cluster using the resources in the **deploy/openshift** directory.

   5.1. Create the OpenShift project and set the security context policy:

   ```
   [student@workstation docker]$ cd /home/student/DO180/labs/review/deploy/
   openshift
   ```

   ```
   [student@workstation openshift]$ oc login -u developer -p developer \
   https://ocp.lab.example.com:8443
   [student@workstation openshift]$ oc new-project review
   [student@workstation openshift]$ ../../setpolicy.sh
   ```

   5.2. Create the persistent volume that will hold the Nexus server's persistent data:

   ```
   [student@workstation openshift]$ ./create-pv.sh
   ```

   5.3. Process the template and create the Kubernetes resources:

```
[student@workstation openshift]$ oc process -f resources/nexus-template.json \
 | oc create -f -
[student@workstation openshift]$ oc get pods -w
```

5.4. Expose the service (create a route):

```
[student@workstation openshift]$ oc expose svc/nexus
[student@workstation openshift]$ oc get route
```

5.5. Use a browser to connect to the Nexus server web application using **`http://{routeaddress}/nexus/`**.

6. Run the grading script to check your work:

```
[student@workstation review]$ lab review grade
```

7. Clean up your OpenShift project:

```
[student@workstation review]$ oc delete project review
[student@workstation review]$ ./delete-pv.sh
```

This concludes this lab.

Evaluation

After deploying the Nexus server container image to the OpenShift cluster, you can check your work by running the lab grading script:

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

APPENDIX A

IMPLEMENTING MICROSERVICES ARCHITECTURE

Overview	
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.
Sections	• Implementing Microservices Architectures (with Guided Exercise)

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. Bigger applications are often built and deployed in this monolithic fashion.

When a monolithic application is broken into tiers, it usually has to go though 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:

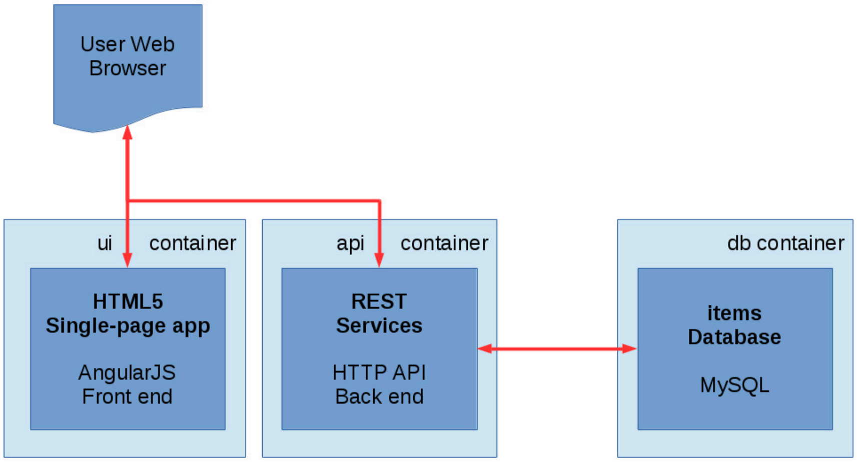


Figure A.1: To Do List application broken into tiers and each deployed as containers

Comparing the source code of the original monolithic application with the new one re-factored into services. The 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/DO180/labs/todoapp** **/home/student/DO180/labs/appendix-microservices**
Application URL	**http://127.0.0.1:30080**
Resources	MySQL 5.6 image (**mysql:5.6**), RHEL 7.3 Image (**rhel7.3**), To Do API image (**do180/todoapi_nodejs**), To Do front-end image (**do180/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 ~/DO180/labs/appendix-microservices/apps/html5/src
```

Steps

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 ~/DO180/labs/appendix-microservices/apps/html5/
   [student@workstation html5]$ mv \
   ~/DO180/labs/appendix-microservices/apps/nodejs/todo/* \
   ~/DO180/labs/appendix-microservices/apps/html5/src/
   ```

   1.2. The current front-end application interacts with the API service using a relative URL. Because the API and front-end code will now run in separate containers, the front-end needs to be adjusted to point to the absolute URL of the To Do List application API.

Open the **/home/student/DO180/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:

```
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 that was created in the previous chapter:

```
[student@workstation ~]$ cd /home/student/DO180/labs/appendix-microservices/
[student@workstation appendix-microservices]$ cd images/apache
[student@workstation apache]$ ./build.sh
```

   2.2. Verify that the image 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/DO180/labs/appendix-microservices/
[student@workstation appendix-microservices]$ cd deploy/html5
[student@workstation html5]$ ./build.sh
```

   2.4. Verify that the image 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/DO180/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_ENV_MYSQL_DATABASE,
    username: process.env.MYSQL_ENV_MYSQL_USER,
    password: process.env.MYSQL_ENV_MYSQL_PASSWORD,
    params: {
        host: process.env.MYSQL_PORT_3306_TCP_ADDR,
        port: process.env.MYSQL_PORT_3306_TCP_PORT,
        dialect: 'mysql'
    }
};
```

Note

This file can be copied and pasted from **/home/student/DO180/labs/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/DO180/labs/appendix-microservices/apps/nodejs/**:

```
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 is the same child image that was created in the previous guided exercise.

```
[student@workstation ~]$ cd /home/student/DO180/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 built successfully:

```
[student@workstation ~]$ docker images
REPOSITORY                           TAG                 IMAGE ID       CREATED
                  VIRTUAL SIZE
do180/todonodejs                     latest              18f48b42445d        14
minutes ago       518.3 MB
do180/httpd                          latest              ebc1d069d189        49
minutes ago       493.7 MB
do180/todo_frontend                  latest              46a3c5521828        50
minutes ago       503.3 MB
registry.access.redhat.com/rhel7.3   latest              6c3a84d798dc         5
weeks ago         201.7 MB
```

5. Run the Containers

   5.1. Use the **run** script to run the containers:

```
[student@workstation ~]$ cd /home/student/DO180/labs/appendix-microservices
[student@workstation appendix-microservices]$ cd deploy/nodejs/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                  COMMAND                  CREATED
                STATUS              PORTS                     NAMES
a97c5e1ab922        do180/todo_frontend    "httpd -D FOREGROUND"    About
 a minute ago   Up About a minute   0.0.0.0:30000->80/tcp     todoui
a9920830b53f        do180/todoapi_nodejs   "./run.sh"               About
 a minute ago   Up About a minute   0.0.0.0:30080->30080/tcp  todoapi
984d636591b8        mysql                 "container-entrypoint"   About
 a minute ago   Up About a minute   0.0.0.0:30306->3306/tcp   mysql
```

6. Test the Application

   6.1. Use a **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 the workstation 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
```

   This concludes the guided exercise.

7. Clean Up

   7.1. Stop the running containers:

```
[student@workstation linked]$ docker stop todoapi todoui mysql
```

7.2. Remove the stopped containers:

```
[student@workstation linked]$ docker rm todoapi todoui mysql
```

7.3. Remove the images created during the appendix:

```
[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 and the connection information is dynamic. The *linked containers* feature in Docker resolves this issue.
- Environment variables created by the **`--link`** ***`container`*****`:`*****`alias`*** option are based on the alias given in the command.