



Red Hat Training and Certification

Student Workbook (ROLE)

OCP 4.5 DO288

Red Hat OpenShift Development II: Containerizing Applications

Edition 2

Red Hat OpenShift Development II: Containerizing Applications



OCP 4.5 DO288
Red Hat OpenShift Development II: Containerizing
Applications
Edition 2 20200911
Publication date 20200911

Authors: Zach Guterman, Richard Allred, Ricardo Jun, Ravishankar Srinivasan,
Fernando Lozano, Ivan Chavero, Dan Kolepp, Jordi Sola Alaball,
Manuel Aude Morales, Eduardo Ramírez Martínez

Editor: David O'Brien, Seth Kenlon

Copyright © 2019 Red Hat, Inc.

The contents of this course and all its modules and related materials, including handouts to audience members, are
Copyright © 2019 Red Hat, Inc.

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

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

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

Red Hat, Red Hat Enterprise Linux, the Red Hat logo, JBoss, Hibernate, Fedora, the Infinity logo, and RHCE are
trademarks of Red Hat, Inc., registered in the United States and other countries.

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

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

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

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

All other trademarks are the property of their respective owners.

Contributors: Grega Bremec, Sajith Sugathan, Dave Sacco, Rob Locke, Bowe Strickland, Rudolf
Kastl, Chris Tusa

Document Conventions	ix
Introduction	xi
Red Hat OpenShift Development II: Containerizing Applications	xi
Orientation to the Classroom Environment	xii
Internationalization	xv
1. Deploying and Managing Applications on an OpenShift Cluster	1
Introducing OpenShift Container Platform 4	2
Quiz: Introducing OpenShift 4	7
Guided Exercise: Configuring the Classroom Environment	11
Deploying an Application to an OpenShift Cluster	17
Guided Exercise: Deploying an Application to an OpenShift Cluster	25
Managing Applications with the Web Console	32
Guided Exercise: Managing an Application with the Web Console	37
Managing Applications with the CLI	46
Guided Exercise: Managing an Application with the CLI	51
Lab: Deploying and Managing Applications on an OpenShift Cluster	60
Summary	68
2. Designing Containerized Applications for OpenShift	69
Selecting a Containerization Approach	70
Quiz: Selecting a Containerization Approach	74
Building Container Images with Advanced Dockerfile Instructions	80
Guided Exercise: Building Container Images with Advanced Dockerfile Instructions	88
Injecting Configuration Data into an Application	96
Guided Exercise: Injecting Configuration Data into an Application	104
Lab: Designing Containerized Applications for OpenShift	111
Summary	121
3. Publishing Enterprise Container Images	123
Managing Images in an Enterprise Registry	124
Guided Exercise: Using an Enterprise Registry	132
Allowing Access to the OpenShift Registry	138
Guided Exercise: Using the OpenShift Registry	142
Creating Image Streams	146
Guided Exercise: Creating an Image Stream	153
Lab: Publishing Enterprise Container Images	157
Summary	166
4. Building Applications	167
Describing the OpenShift Build Process	168
Quiz: The OpenShift Build Process	172
Managing Application Builds	178
Guided Exercise: Managing Application Builds	181
Triggering Builds	187
Guided Exercise: Triggering Builds	190
Implementing Post-commit Build Hooks	196
Guided Exercise: Implementing Post-Commit Build Hooks	198
Lab: Building Applications	203
Summary	211
5. Customizing Source-to-Image Builds	213
Describing the Source-to-Image Architecture	214
Quiz: Describing the Source-to-Image Architecture	220
Customizing an Existing S2I Base Image	224
Guided Exercise: Customizing S2I Builds	227
Creating an S2I Builder Image	232

Guided Exercise: Creating an S2I Builder Image	238
Lab: Customizing Source-to-Image Builds	249
Summary	263
6. Creating Applications from OpenShift Templates	265
Describing the Elements of an OpenShift Template	266
Quiz: Describing the Elements of an OpenShift Template	271
Creating a Multicontainer Template	273
Guided Exercise: Creating a Multicontainer Template	278
Lab: Creating Applications from OpenShift Templates	289
Summary	300
7. Managing Application Deployments	301
Monitoring Application Health	302
Guided Exercise: Activating Probes	307
Selecting the Appropriate Deployment Strategy	313
Guided Exercise: Implementing a Deployment Strategy	317
Managing Application Deployments with CLI Commands	324
Guided Exercise: Managing Application Deployments	329
Lab: Managing Application Deployments	336
Summary	348
8. Implementing Continuous Integration and Continuous Deployment Pipelines in OpenShift	349
Describing CI/CD Concepts	350
Quiz: CI/CD Concepts	354
Implementing Jenkins Pipelines on OpenShift	358
Guided Exercise: Run a Simple Jenkins Pipeline	364
Writing Custom Jenkins Pipelines	375
Guided Exercise: Create and Run a Jenkins Pipeline	380
Lab: Implementing Continuous Integration and Continuous Deployment Pipelines in OpenShift	393
Summary	410
9. Building Applications for OpenShift	411
Integrating External Services	412
Guided Exercise: Integrating an External Service	414
Containerizing Services	418
Guided Exercise: Deploying a Containerized Nexus Server	425
Deploying Applications with Red Hat OpenShift Application Runtimes	435
Guided Exercise: Deploying an Application with Red Hat OpenShift Application Runtimes	441
Lab: Building Cloud-Native Applications for OpenShift	451
Summary	461
10. Comprehensive Review: Red Hat OpenShift Development II: Containerizing Applications	463
Comprehensive Review	464
Lab: Designing a Container Image for OpenShift	466
Lab: Containerizing and Deploying a Service	477
Lab: Building and Deploying a Multicontainer Application	490
A. Creating a GitHub Account	507
Creating a GitHub Account	508
B. Creating a Quay Account	511
Creating a Quay Account	512
C. Useful Git Commands	515

Document Conventions



References

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



Note

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



Important

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



Warning

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

Introduction

Red Hat OpenShift Development II: Containerizing Applications

Red Hat OpenShift Container Platform, based on container technology and Kubernetes, provides developers an enterprise-ready solution for developing and deploying containerized software applications.

Red Hat OpenShift Development I: Containerizing Applications (DO288), the second course in the OpenShift development track, teaches students how to design, build, and deploy containerized software applications on an OpenShift cluster. Whether writing native container applications or migrating existing applications, this course provides hands-on training to boost developer productivity powered by Red Hat OpenShift Container Platform.

Course Objectives

- Design, build, and deploy containerized applications on an OpenShift cluster.

Audience

- Software Developers
- Software Architects

Prerequisites

- Either have completed the Introduction to Containers, Kubernetes, and Red Hat OpenShift course (DO180), or have equivalent knowledge.
- RHCSA or higher is helpful for navigation and usage of the command line, but it is not required.

Orientation to the Classroom Environment

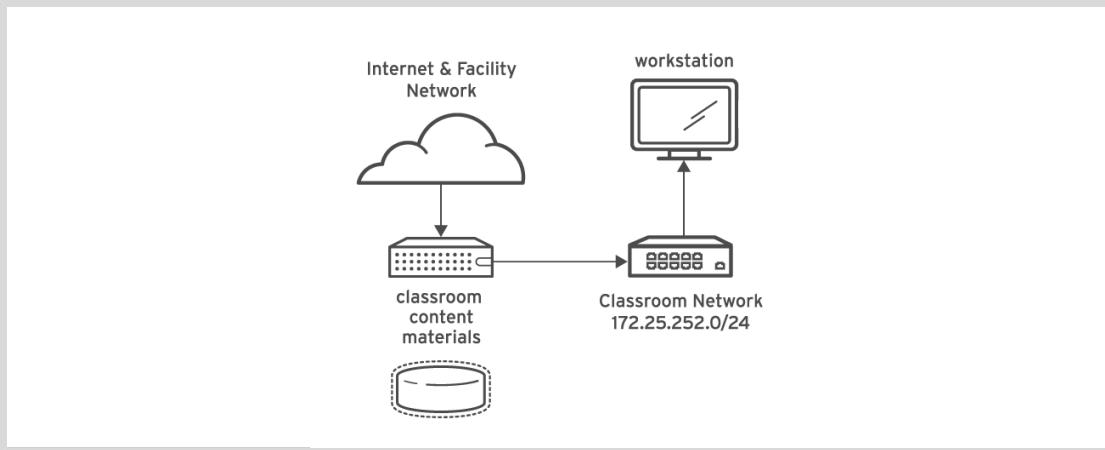


Figure 0.1: Classroom environment

In this course, the main computer system used for hands-on learning activities is **workstation**. This is a virtual machine (VM) named **workstation.lab.example.com**.

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

Classroom Machines

Machine name	IP addresses	Role
content.example.com, materials.example.com, classroom.example.com	172.25.252.254, 172.25.253.254, 172.25.254.254	Classroom utility server
workstation.lab.example.com	172.25.250.254, 172.25.252.1	Student graphical workstation

Several systems in the classroom provide supporting services. Two servers, **content.example.com** and **materials.example.com**, are sources for software and lab materials used in hands-on activities. Information on how to use these servers is provided in the instructions for those activities.

Students use the **workstation** machine to access a shared OpenShift cluster hosted externally in AWS. Students do not have cluster administrator privileges on the cluster, but that is not necessary to complete the DO288 content.

Students are provisioned an account on a shared OpenShift 4 cluster when they provision their environments in the Red Hat Online Learning interface. Cluster information such as the API endpoint, and cluster-ID, as well as their username and password are presented to them when they provision their environment.

Introduction

Students also have access to a MySQL and a Nexus server hosted by either the OpenShift cluster or by AWS. Hands-on activities in this course provide instructions to access these servers when required.

Hands-on activities in DO288 also require that students have personal accounts on two public, free internet services: GitHub and Quay.io. Students need to create these accounts if they do not already have them (see Appendix) and verify their access by signing in to these services before starting the class.

Controlling Your Systems

Students are assigned remote computers in a Red Hat Online Learning classroom. They are accessed through a web application hosted at rol.redhat.com [<http://rol.redhat.com>]. Students should log in to this site using their Red Hat Customer Portal user credentials.

Controlling the Virtual Machines

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

Machine States

Virtual Machine State	Description
STARTING	The virtual machine is in the process of booting.
STARTED	The virtual machine is running and available (or, when booting, soon will be).
STOPPING	The virtual machine is in the process of shutting down.
STOPPED	The virtual machine is completely shut down. Upon starting, the virtual machine boots 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 is available.

Classroom/Machine Actions

Button or Action	Description
PROVISION LAB	Create the ROL classroom. Creates all of the virtual machines needed for the classroom and starts them. This can take several minutes to complete.
DELETE LAB	Delete the ROL classroom. Destroys all virtual machines in the classroom. Caution: Any work generated on the disks is lost.
START LAB	Start all virtual machines in the classroom.

Button or Action	Description
SHUTDOWN LAB	Stop all virtual 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 virtual machine and run commands. In most cases, students should log in to the workstation virtual machine and use ssh to connect to the other virtual machines.
ACTION → Start	Start (power on) the virtual machine.
ACTION → Shutdown	Gracefully shut down the virtual machine, preserving the contents of its disk.
ACTION → Power Off	Forcefully shut down the virtual machine, preserving the contents of its disk. This is equivalent to removing the power from a physical machine.
ACTION → Reset	Forcefully shut down the virtual machine and reset the disk to its initial state. Caution: Any work generated on the disk is lost.

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

At the start of an exercise, if instructed to reset all virtual machines, click **ACTION → Reset**

If you want to return the classroom environment to its original state at the start of the course, you can click **DELETE LAB** to remove the entire classroom environment. After the lab has been deleted, click **PROVISION LAB** to provision a new set of classroom systems.



Warning

The **DELETE LAB** operation cannot be undone. Any work you have completed in the classroom environment up to that point will be lost.

The Autostop Timer

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

To adjust the timer, click **MODIFY** to display the **New Autostop Time** dialog box. Set the number of hours and minutes until the classroom should automatically stop. Note that there is a maximum time of ten hours. Click **ADJUST TIME** to apply this change to the timer settings.

Internationalization

Per-user Language Selection

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

Language Settings

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

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

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

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



Note

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

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

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

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

```
[user@host ~]$ LANG=fr_FR.utf8 date  
jeu. avril 25 17:55:01 CET 2019
```

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

Input Method Settings

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

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

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

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

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

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



Note

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

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

System-wide Default Language Settings

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

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

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

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

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



Important

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

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

Language Packs

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

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

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

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

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

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

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



Important

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

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

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



References

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

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

Language Codes Reference



Note

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

Language Codes

Language	Langpacks Suffix	\$LANG value
English (US)	en	en_US.utf8

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

Chapter 1

Deploying and Managing Applications on an OpenShift Cluster

Goal

Deploy applications using various application packaging methods to an OpenShift cluster and manage their resources.

Objectives

- Describe the architecture and new features in OpenShift 4.
- Deploy an application to the cluster from a Dockerfile with the CLI.
- Deploy an application from a container image and manage its resources using the web console.
- Deploy an application from source code and manage its resources using the command-line interface.

Sections

- Introducing OpenShift 4 (and Quiz)
- Deploying an Application to an OpenShift Cluster (and Guided Exercise)
- Managing Applications with the Web Console (and Guided Exercise)
- Managing Applications with the CLI (and Guided Exercise)

Lab

Deploying and Managing Applications on an OpenShift Cluster

Introducing OpenShift Container Platform 4

Objectives

After completing this section, you should be able to describe the architecture and new features in OpenShift Container Platform 4.

OpenShift Container Platform 4 Architecture

Red Hat OpenShift Container Platform 4 (RHOCP 4) is a set of modular components and services built on top of Red Hat CoreOS and Kubernetes. OpenShift adds *platform as a service* (*PaaS*) capabilities such as remote management, increased security, monitoring and auditing, application life-cycle management, and self-service interfaces for developers. It provides orchestration services and simplifies the deployment, management, and scaling of containerized applications.

An OpenShift cluster can be managed the same way as any other Kubernetes cluster, but it can also be managed using the management tools provided by OpenShift, such as the command-line interface or the web console. This additional tooling allows for more productive workflows and makes everyday tasks much more manageable.

One of the main advantages of using OpenShift is that it uses several nodes to ensure the resiliency and scalability of its managed applications. OpenShift forms a cluster of node servers that run containers and are centrally managed by a set of master servers. A single host can act as both a master and a node, but typically you should segregate those roles for increased stability and high-availability.

The following diagram illustrates the high-level logical overview of the OpenShift Container Platform 4 architecture.

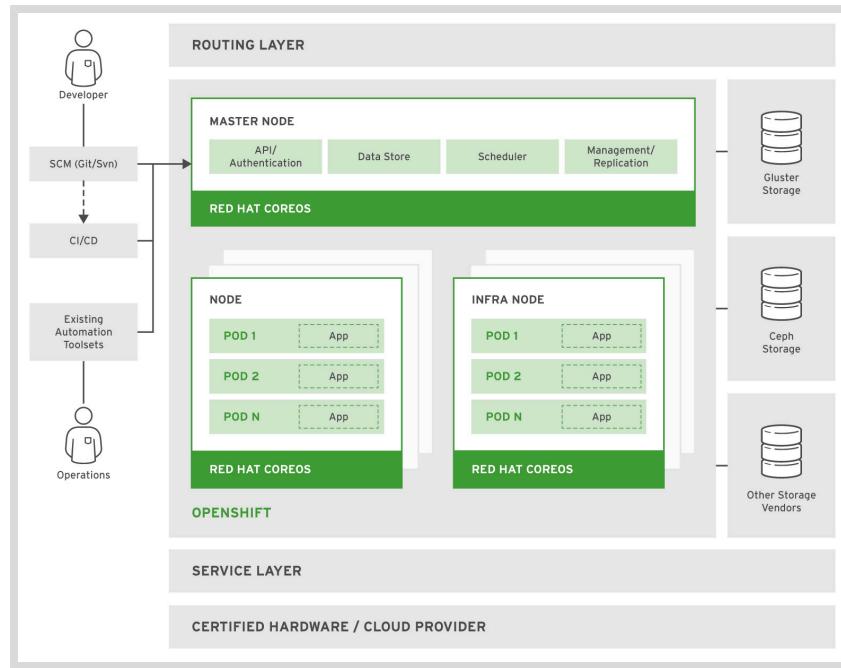


Figure 1.1: OpenShift 4 architecture

The following diagram illustrates the OpenShift Container Platform stack.

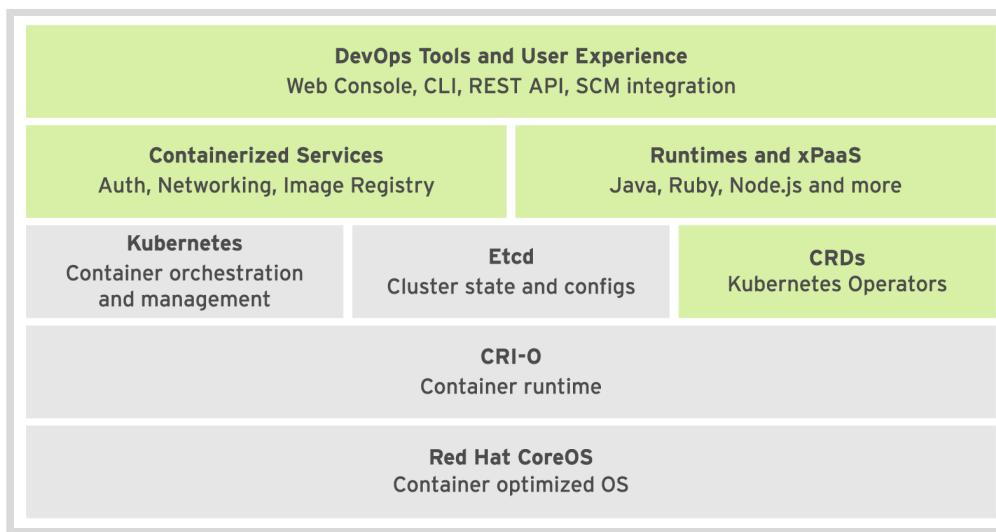


Figure 1.2: OpenShift component stack

From bottom to top, and from left to right, this shows the basic container infrastructure, integrated and enhanced by Red Hat:

Red Hat CoreOS

Red Hat CoreOS is the base OS on top which OpenShift runs. Red Hat CoreOS is a Linux distribution focused on providing an immutable operating system for container execution.

CRI-O

CRI-O is an implementation of the Kubernetes *Container Runtime Interface (CRI)* to enable using *Open Container Initiative (OCI)* compatible runtimes. CRI-O can use any container runtime that satisfies CRI, such as: **runc** (used by the Docker service) or **rkt** (from CoreOS).

Kubernetes

Kubernetes manages a cluster of hosts, physical or virtual, running containers. It uses resources that describe multicontainer applications composed of multiple resources, and how they interconnect.

Etcd

Etcd is a distributed key-value store, used by Kubernetes to store configuration and state information about the containers and other resources inside the Kubernetes cluster.

Custom Resource Definitions (CRDs)

Custom Resource Definitions (CRDs) are resource types stored in Etcd and managed by Kubernetes. These resource types form the state and configuration of all resources managed by OpenShift.

Containerized Services

Containerized services fulfill many PaaS infrastructure functions, such as networking and authorization. RHOPC uses the basic container infrastructure from Kubernetes and the underlying container runtime for most internal functions. That is, most RHOPC internal services run as containers orchestrated by Kubernetes.

Runtimes and xPaaS

Runtimes and xPaaS are base container images ready for use by developers, each preconfigured with a particular runtime language or database. The xPaaS offering is a set of base images for Red Hat middleware products such as JBoss EAP and ActiveMQ.

Red Hat OpenShift Application Runtimes (RHOAR) are a set runtimes optimized for cloud native applications in OpenShift. The application runtimes available are Red Hat JBoss EAP, OpenJDK, Thorntail, Eclipse Vert.x, Spring Boot, and Node.js.

DevOps Tools and User Experience

DevOps tools and user experience: RHOPC provides web UI and CLI management tools for managing user applications and RHOPC services. The OpenShift web UI and CLI tools are built from REST APIs which can be used by external tools such as IDEs and CI platforms.

The following table lists some of the most commonly used terminology when you work with OpenShift.

OpenShift Terminology

Term	Definition
Node	A server that hosts applications in an OpenShift cluster.
Master Node	A node server that manages the control plane in an OpenShift cluster. Master nodes provide basic cluster services such as APIs or controllers.
Worker Node	Also called a Compute Node , worker nodes execute workloads for the cluster. Application pods are scheduled onto worker nodes.
Resource	Resources are any kind of component definition managed by OpenShift. Resources contain the configuration of the managed component (for example, the role assigned to a node), and the current state of the component (for example, if the node is available).
Controller	A controller is an OpenShift component that watches resources and makes changes attempting to move the current state towards the desired state.
Label	A key-value pair that can be assigned to any OpenShift resource. Selectors use labels to filter eligible resources for scheduling and other operations.
Namespace or Project	A scope for OpenShift resources and processes, so that resources with the same name can be used in different contexts.
Console	A web UI provided by OpenShift that allows developers and administrators to manage cluster resources.



Note

The latest OpenShift versions implement many controllers as *Operators*. Operators are Kubernetes plug-in components that can react to cluster events and control the state of resources. Operators and the Operator Framework are outside the scope of this course.

New Features in RHOPC 4

RHOPC 4 is a massive change from previous versions. As well as keeping backwards compatibility with previous releases, it includes new features, such as:

- CoreOS as the default operating system for all nodes, offering an immutable infrastructure optimized for containers.

- A new cluster installer, which simplifies the process of installing and updating the masters and worker nodes in the cluster.
- A self-managing platform, able to automatically apply cluster updates and recoveries without disruption.
- A redesigned web console based on the concept of “personas”, targeting both platform administrators and developers.
- An Operator SDK to build, test, and package Operators.

Describing OpenShift Resource Types

As a developer, you will work with many different kinds of resource types in OpenShift. These resources can be created and configured using a YAML or a JSON file, or using OpenShift management tools:

Pods (pod)

Collections of containers that share resources, such as IP addresses and persistent storage volumes. Pods are the basic unit of work for OpenShift.

Services (svc)

Specific IP/port combinations that provides access to a pool of pods. By default, services connect clients to pods in a round-robin fashion.

Replication Controllers (rc)

OpenShift resources that define how pods are replicated (horizontally scaled) to different nodes. Replication controllers are a basic OpenShift service to provide high availability for pods and containers.

Persistent Volumes (pv)

Storage areas to be used by pods.

Persistent Volume Claims (pvc)

Requests for storage by a pod. A **pvc** links a **pv** to a pod so its containers can make use of it, usually by mounting the storage into the container's file system.

Config Maps (cm)

A set of keys and values that can be used by other resources. ConfigMaps and Secrets are usually used to centralize configuration values used by several resources. Secrets differ from ConfigMaps maps in that Secrets are used to store sensitive data (usually encrypted), and their access is restricted to fewer authorized users.

Deployment Configs (dc)

A set of containers included in a pod, and the deployment strategies to be used. A **dc** also provides a basic but extensible continuous delivery workflow.

Build Configs (bc)

A process to be executed in the OpenShift project. The OpenShift Source-to-Image (S2I) feature uses BuildConfigs to build a container image from application source code stored in a Git repository. A **bc** works together with a **dc** to provide a basic but extensible continuous integration and continuous delivery workflows.

Routes

DNS host names recognized by the OpenShift router as an ingress point for various applications and microservices deployed on the cluster.

Image Streams (is)

An image stream and its tags provide an abstraction for referencing container images from within OpenShift Container Platform. The image stream and its tags allow you to track what images are available and ensure that you are using the specific image you need even if the image in the repository changes. Image streams do not contain actual image data, but present a virtual view of related images, similar to an image repository.



References

Kubernetes documentation website

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

OpenShift documentation website

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

CoreOS Operators and Operator Framework

<https://coreos.com/operators/>

► Quiz

Introducing OpenShift 4

Choose the correct answers to the following questions:

When you have completed the quiz, click **CHECK**. If you want to try again, click **RESET**. Click **SHOW SOLUTION** to see all of the correct answers.

► 1. Which statement is correct regarding OpenShift additions to Kubernetes?

- a. OpenShift adds features to make application development and deployment on Kubernetes easy and efficient.
- b. Container images created for OpenShift cannot be used with plain Kubernetes.
- c. To enable new features, Red Hat maintains forked versions of Kubernetes internal to the RHOC product.
- d. There are no new features for continuous integration and continuous deployment with RHOC, but you can use external tools instead.

► 2. Which statement is correct regarding persistent storage in OpenShift?

- a. Developers create a persistent volume claim to request a cluster storage area that a project pod can use to store data.
- b. A persistent volume claim represents a storage area that a pod can request to store data but is provisioned by the cluster administrator.
- c. A persistent volume claim represents the amount of memory that can be allocated to a node, so that a developer can state how much memory he requires for his application to run.
- d. A persistent volume claim represents the number of CPU processing units that can be allocated to an application pod, subject to a limit managed by the cluster administrator.
- e. OpenShift supports persistent storage by allowing administrators to directly map storage available on nodes to applications running on the cluster.

► 3. Which two statements are correct regarding 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 service is responsible for providing IP addresses for external access to pods.
- d. A route is responsible for providing a host name for external access to pods.
- e. A replication controller is responsible for monitoring and maintaining the number of pods for a particular application.

► **4. Which two statements are correct regarding the OpenShift 4 architecture? (Choose two.)**

- a. OpenShift nodes can be managed without a master. The nodes form a peer to peer network.
- b. OpenShift masters manage pod scaling and scheduling pods to run on nodes.
- c. Master nodes in a cluster must be running Red Hat CoreOS.
- d. Master nodes in a cluster must be running Red Hat Enterprise Linux 8.
- e. Master nodes in a cluster must be running Red Hat Enterprise Linux 7.

► Solution

Introducing OpenShift 4

Choose the correct answers to the following questions:

When you have completed the quiz, click **CHECK**. If you want to try again, click **RESET**. Click **SHOW SOLUTION** to see all of the correct answers.

► 1. Which statement is correct regarding OpenShift additions to Kubernetes?

- a. OpenShift adds features to make application development and deployment on Kubernetes easy and efficient.
- b. Container images created for OpenShift cannot be used with plain Kubernetes.
- c. To enable new features, Red Hat maintains forked versions of Kubernetes internal to the RHOCP product.
- d. There are no new features for continuous integration and continuous deployment with RHOCP, but you can use external tools instead.

► 2. Which statement is correct regarding persistent storage in OpenShift?

- a. Developers create a persistent volume claim to request a cluster storage area that a project pod can use to store data.
- b. A persistent volume claim represents a storage area that a pod can request to store data but is provisioned by the cluster administrator.
- c. A persistent volume claim represents the amount of memory that can be allocated to a node, so that a developer can state how much memory he requires for his application to run.
- d. A persistent volume claim represents the number of CPU processing units that can be allocated to an application pod, subject to a limit managed by the cluster administrator.
- e. OpenShift supports persistent storage by allowing administrators to directly map storage available on nodes to applications running on the cluster.

► 3. Which two statements are correct regarding 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 service is responsible for providing IP addresses for external access to pods.
- d. A route is responsible for providing a host name for external access to pods.
- e. A replication controller is responsible for monitoring and maintaining the number of pods for a particular application.

► **4. Which two statements are correct regarding the OpenShift 4 architecture? (Choose two.)**

- a. OpenShift nodes can be managed without a master. The nodes form a peer to peer network.
- b. OpenShift masters manage pod scaling and scheduling pods to run on nodes.
- c. Master nodes in a cluster must be running Red Hat CoreOS.
- d. Master nodes in a cluster must be running Red Hat Enterprise Linux 8.
- e. Master nodes in a cluster must be running Red Hat Enterprise Linux 7.

► Guided Exercise

Configuring the Classroom Environment

In this exercise, you will configure the **workstation** to access all infrastructure used by this course.

Outcomes

You should be able to:

- Configure your **workstation** to access an OpenShift cluster, a container image registry, and a Git repository used throughout the course.
- Fork this course's sample applications repository to your personal GitHub account.
- Clone this course's sample applications repository from your personal GitHub account to your **workstation** VM.

Before You Begin

To perform this exercise, ensure you have:

- Access to the DO288 course in the Red Hat Training's Online Learning Environment.
- The connection parameters and a developer user account to access an OpenShift cluster managed by Red Hat Training.
- A personal, free GitHub account. If you need to register to GitHub, see the instructions in *Appendix A, Creating a GitHub Account*.
- A personal, free Quay.io account. If you need to register to Quay.io, see the instructions in *Appendix B, Creating a Quay Account*.

► 1. Before starting any exercise, you need to configure your **workstation** VM.

For the following steps, use the values the Red Hat Training Online Learning environment provides to you when you provision your online lab environment:

The screenshot shows the 'Lab Environment' tab selected in the top navigation bar. Below it, a section titled 'Lab Controls' contains instructions to click 'CREATE' to build virtual machines and 'DELETE' to remove them. Below this are two buttons: 'DELETE' (red) and 'STOP' (teal). A tooltip icon is next to the STOP button. A red box highlights the 'OpenShift Details' section, which lists connection parameters:

Username	RHT_OCP4_DEV_USER	youruser
Password	RHT_OCP4_DEV_PASSWORD	yourpassword
API Endpoint	RHT_OCP4_MASTER_API	https://api.cluster.domain.example.com:6443
Console Web Application	https://console-openshift-console.apps.cluster.domain.example.com	
Cluster Id	your-cluster-id	

Below this is a table showing two VMs: 'workstation' and 'classroom'. Both are listed as 'active'. Each row has an 'ACTION' dropdown and an 'OPEN CONSOLE' button.

Open a terminal on your **workstation** VM and execute the following command. Answer its interactive prompts to configure your workstation before starting any other exercise in this course.

If you make a mistake, you can interrupt the command at any time using **Ctrl+C** and start over.

```
[student@workstation ~]$ lab-configure
```

1. The **lab-configure** command starts by displaying a series of interactive prompts, and will try to find some sensible defaults for some of them.

This script configures the connection parameters to access the OpenShift cluster for your lab scripts

- Enter the API Endpoint: <https://api.cluster.domain.example.com:6443> ①
- Enter the Username: *youruser* ②
- Enter the Password: *yourpassword* ③
- Enter the GitHub Account Name: *yourgituser* ④
- Enter the Quay.io Account Name: *yourquayuser* ⑤

...output omitted...

- 1 The URL to your OpenShift cluster's Master API. Type the URL as a single line, without spaces or line breaks. Red Hat Training provides this information to you when you provision your lab environment. You need this information to log in to the cluster and also to deploy containerized applications.
- 2 3 Your OpenShift developer user name and password. Red Hat Training provides this information to you when you provision your lab environment. You need to use this user name and password to log in to OpenShift. You will also use your user name as part of identifies such as route host names and project names,

to avoid collision with identifiers from other students who share the same OpenShift cluster with you.

- 4.5** Your personal GitHub and Quay.io account names. You need valid, free accounts on these online services to perform this course's exercises. If you have never used any of these online services, refer to *Appendix A, Creating a GitHub Account* and *Appendix B, Creating a Quay Account* for instructions about how to register.



Note

If you use two-factor authentication with your GitHub account you may want to create a personal access token for use from the **workstation** VM during the course. Refer to the following documentation on how to setup a personal access on your account: Creating a personal access token for the command line [<https://help.github.com/en/articles/creating-a-personal-access-token-for-the-command-line>]

- 1.2. The **lab-configure** command prints all the information that you entered and tries to connect to your OpenShift cluster:

```
...output omitted...
```

You entered:

- API Endpoint: https://api.cluster.domain.example.com:6443
- Username: youruser
- Password: yourpassword
- GitHub Account Name: yourgituser
- Quay.io Account Name: yourquayuser

```
...output omitted...
```

- 1.3. If **lab-configure** finds any issues, it displays an error message and exits. You will need to verify your information and run the **lab-configure** command again. The following listing shows an example of a verification error:

```
...output omitted...
```

Verifying your Master API URL...

ERROR:

Cannot connect to an OpenShift 4.5 API using your URL.
Please verify your network connectivity and that the URL does not point to an OpenShift 3.x nor to a non-OpenShift Kubernetes API.
No changes made to your lab configuration.

- 1.4. If everything is OK so far, the **lab-configure** tries to access your public GitHub and Quay.io accounts:

```
...output omitted...

Verifying your GitHub account name...

Verifying your Quay.io account name...

...output omitted...
```

15. Again, **lab-configure** displays an error message and exits if it finds any issues. You will need to verify your information and run the **lab-configure** command again. The following listing shows an example of a verification error:

```
...output omitted...

Verifying your GitHub account name...

ERROR:
Cannot find a GitHub account named: invalidusername.
No changes made to your lab configuration.
```

16. Finally, the **lab-configure** command verifies that your OpenShift cluster reports the expected wildcard domain.

```
...output omitted...

Verifying your cluster configuration...

...output omitted...
```

17. If all checks pass, the **lab-configure** command saves your configuration:

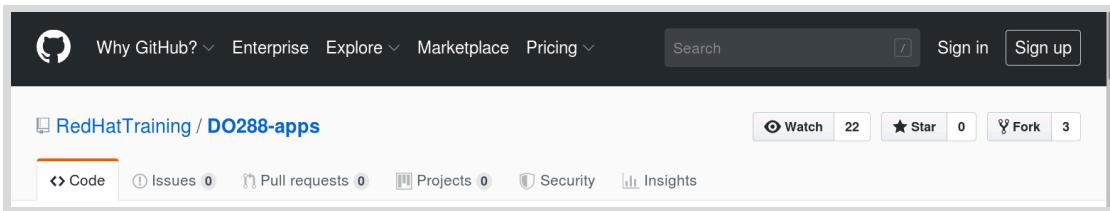
```
...output omitted...

Saving your lab configuration file...

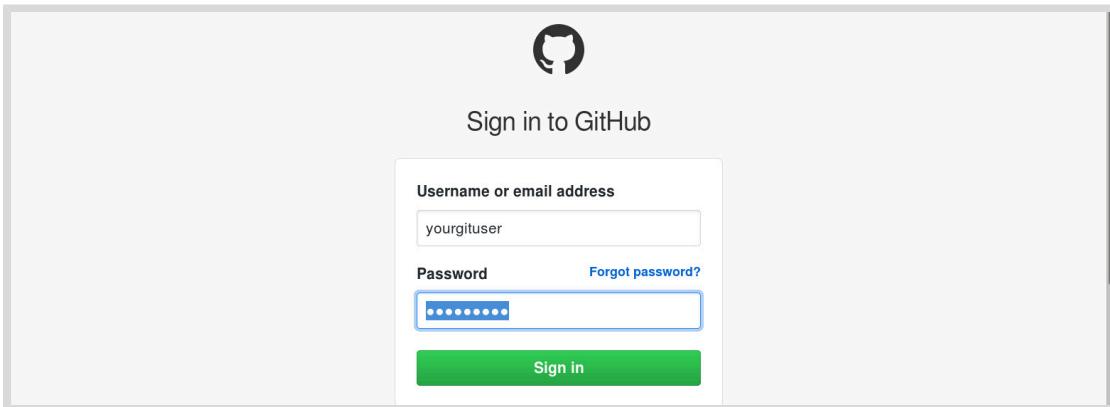
Saving your Maven settings file...

All fine, lab config saved. You can now proceed with your exercises.
```

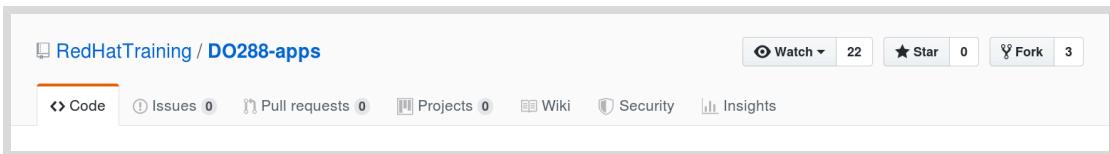
18. If there were no errors saving your configuration, you are almost ready to start any of this course's exercises. If there were any errors, do not try to start any exercise until you can execute the **lab-configure** command successfully.
- ▶ 2. Before starting any exercise, you need to fork this course's sample applications into your personal GitHub account. Perform the following steps:
 - 2.1. Open a web browser and navigate to <https://github.com/RedHatTraining/DO288-apps>. If you are not logged in to GitHub, click **Sign in** in the upper-right corner.



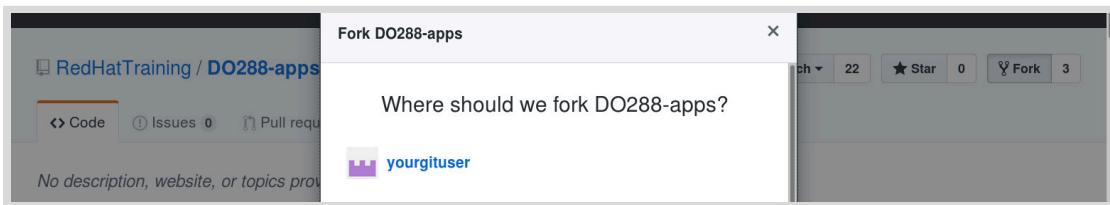
- 2.2. Log in to GitHub using your personal user name and password.



- 2.3. Return to the **RedHatTraining/DO288-apps** repository and click **Fork** in the upper-right corner.



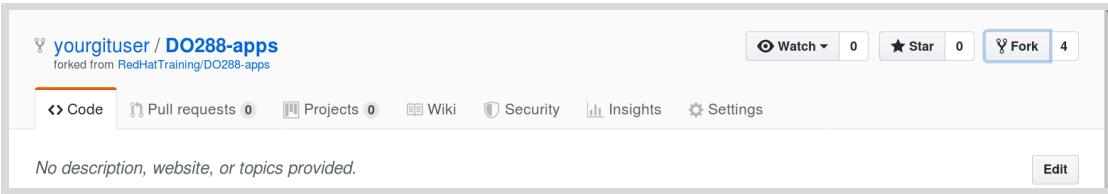
- 2.4. In the **Fork DO288-apps** window, click **yourgituser** to select your personal GitHub project.



Important

While it is possible to rename your personal fork of the <https://github.com/RedHatTraining/DO288-apps> repository, grading scripts, helper scripts, and the example output in this course assume that you retain the name **DO288-apps** when your fork the repository.

- 2.5. After a few minutes, the GitHub web interface displays your new repository **yourgituser/DO288-apps**.



- 3. Before starting any exercise, you also need to clone this course's sample applications from your personal GitHub account to your **workstation** VM. Perform the following steps:
- 3.1. Run the following command to clone this course's sample applications repository. Replace **yourgituser** with the name of your personal GitHub account:

```
[student@workstation ~]$ git clone https://github.com/yourgituser/DO288-apps
Cloning into 'DO288-apps'...
...output omitted...
```

- 3.2. Verify that **/home/student/DO288-apps** is a Git repository:

```
[student@workstation ~]$ cd DO288-apps
[student@workstation DO288-apps]$ git status
# On branch master
nothing to commit, working directory clean
```

- 3.3. Verify that **/home/student/DO288-apps** contains this course's sample applications, and change back to the **student** user's home folder.

```
[student@workstation DO288-apps]$ head README.md
# DO288 Containerized Example Applications
...output omitted...
[student@workstation DO288-apps]$ cd ~
[student@workstation ~]$
```

- 4. Now that you have a local clone of the **DO288-apps** repository on your **workstation** VM, and you have executed the **lab-configure** command successfully, you are ready to start this course's exercises.

During this course, all exercises that build applications from source start from the **master** branch of the **DO288-apps** Git repository. Exercises that make changes to source code require you to create new branches to host your changes, so that the **master** branch always contains a known good starting point. If for some reason you need to pause or restart an exercise, and need to either save or discard about changes you make into your Git branches, refer to *Appendix C, Useful Git Commands*.

This concludes the guided exercise.

Deploying an Application to an OpenShift Cluster

Objectives

After completing this section, you should be able to:

- Deploy an application to a cluster from a Dockerfile using the CLI.
- Describe resources created in a project using the **oc new-app** command and the web console.

Development Paths

Red Hat OpenShift Container Platform is designed for building and deploying containerized applications. OpenShift supports two main use cases:

- When the complete application life cycle, from initial development to production, is managed using OpenShift tools
- When existing containerized applications, built outside of OpenShift, are deployed to OpenShift



Important

In OpenShift 4.5 the **oc new-app** command now produces Deployment resources instead of DeploymentConfig resources by default. This version of DO288 does not cover the Deployment resource, only DeploymentConfigs. To create DeploymentConfig resources, you can pass the **--as-deployment-config** flag when invoking **oc new-app**. You will use the **--as-deployment-config** throughout this version of the course. For more information, see Understanding Deployments and DeploymentConfigs [<https://docs.openshift.com/container-platform/4.5/applications/deployments/what-deployments-are.html#what-deployments-are>].

The **oc new-app** command creates the resources required to build and deploy an application to OpenShift. Different resources are created, according to the desired use case:

- If you want OpenShift to manage the entire application life cycle, use the **oc new-app** command to create a build configuration to manage the build process that creates the application container image. The **oc new-app** command also creates a deployment configuration to manage the deployment process that runs the generated container image in the OpenShift cluster. In the following example you are delegating to the OpenShift cluster the entire life cycle: cloning a Git repository, building a container image, and deploying it to an OpenShift cluster.

```
[user@host ~]$ oc new-app --as-deployment-config \
> https://github.com/RedHatTraining/D0288/tree/master/apps/apache-httpd
```

- If you have an existing containerized application that you want to deploy to OpenShift, use the **oc new-app** command to create a deployment configuration to manage the deployment process that runs the existing container image in the OpenShift cluster. In the following example, you are referring to a container image using the **--docker-image** option:

```
[user@host ~]$ oc new-app --as-deployment-config \
> --docker-image=registry.access.redhat.com/rhel7-mysql57
```

The **oc new-app** command also creates some auxiliary resources, such as services and image streams. These resources are required to support the way OpenShift manages containerized applications, and are presented later in this course.

The **Add to Project** button in the web console performs the same tasks as the **oc new-app** command. A later chapter of this course covers the OpenShift web console and its usage.

Describing the **oc new-app** Command Options

The **oc new-app** command takes, in its simplest form, a single URL argument that points to either a Git repository or a container image. It accesses the URL to determine how to interpret the argument and perform either a build or a deployment.

The **oc new-app** command may not be able to make the decision you want. For example:

- If a Git repository contains both a Dockerfile and an **index.php** file, OpenShift cannot identify which approach to take unless explicitly mentioned.
- If a Git repository contains source code that targets PHP, but the OpenShift cluster supports deploying either PHP version 5.6 or 7.0, the build process fails because it is not clear which version to use.

To accommodate these and other scenarios, the **oc new-app** command provides a number of options to further specify exactly how to build the application:

Supported Options

Option	Description
--as-deployment-config	Configures the oc new-app to create a DeploymentConfig resource instead of a Deployment.
--image-stream -i	Provides the image stream to be used as either the S2I builder image for an S2I build or to deploy a container image.
--strategy	docker or pipeline or source
--code	Provides the URL to a Git repository to be used as input to an S2I build.
--docker-image	Provides the URL to a container image to be deployed.

Managing the Complete Application Life Cycle with OpenShift

OpenShift manages an application life cycle using the *Source-to-Image (S2I)* process. S2I takes application source code from a Git repository, combines it with a base container image, builds the source, and creates a container image with the application ready to run.

The **oc new-app** command takes a Git repository URL as the input argument and inspects the application source code to determine which builder image to use to create the application container image:

```
[user@host ~]$ oc new-app --as-deployment-config http://gitserver.example.com/
mygitrepo
```

The **oc new-app** command can optionally take the builder image stream name as an argument, either as part of the Git URL, prefixed by a tilde (~), or using the **--image-stream** argument (short form: **-i**).

The following two commands illustrate using a PHP S2I builder image:

```
[user@host ~]$ oc new-app --as-deployment-config php~http://gitserver.example.com/
mygitrepo
```

```
[user@host ~]$ oc new-app --as-deployment-config -i php http://
gitserver.example.com/mygitrepo
```

Optionally, follow the image stream name with a specific tag, which is usually the version number of the programming language runtime. For example:

```
[user@host ~]$ oc new-app --as-deployment-config php:7.0~http://
gitserver.example.com/mygitrepo
```

```
[user@host ~]$ oc new-app --as-deployment-config -i php:7.0 http://
gitserver.example.com/mygitrepo
```

Specifying the Image Stream Name

Some developers prefer the **-i** option to the tilde notation because the tilde character is not very readable, depending on the screen font. The following three commands yield the same results:

```
[user@host ~]$ oc new-app --as-deployment-config \
> myis~http://gitserver.example.com/mygitrepo
```

```
[user@host ~]$ oc new-app --as-deployment-config \
> -i myis http://gitserver.example.com/mygitrepo
```

```
[user@host ~]$ oc new-app --as-deployment-config -i myis --strategy source \
> --code http://gitserver.example.com/mygitrepo
```

While the **oc new-app** command aims to be a convenient way to deliver applications, developers need to be aware that the command will try to "guess" the source language of the given Git repository.

From the previous example, if *myis* is not one of the standard S2I image streams provided by OpenShift, only the first example works. The tilde notation disables the language detection functionality of the **oc new-app** command. This allows the usage of an image stream that points to a builder for a programming language not known by the **oc new-app** command.

The tilde (~) and **--image-stream (-i)** options do not work in the same way, the **-i** option requires the git client to be installed locally since the language detection needs to clone the repository so it can inspect the project and the tilde (~) notation does not.

Deploying Existing Containerized Applications to OpenShift

If you develop an application outside of OpenShift, and the application container image is available from a container image registry accessible by the OpenShift cluster, the **oc new-app** command can take that container image URL as an input argument:

```
[user@host ~]$ oc new-app --as-deployment-config \
> registry.example.com/mycontainerimage
```

Notice that there is no way to know from the previous command whether the URL refers to a Git repository or a container image inside a registry. The **oc new-app** command accesses the input URL to resolve this ambiguity. OpenShift inspects the contents of the URL and determines whether it is source code or a container image registry. To avoid ambiguity, use either the **--code** or the **--docker-image** options. For example:

```
[user@host ~]$ oc new-app --as-deployment-config \
> --code http://gitserver.example.com/mygitrepo
```

```
[user@host ~]$ oc new-app --as-deployment-config \
> --docker-image registry.example.com/mycontainerimage
```

Deploying Existing Dockerfiles with OpenShift

In many cases, you have existing container images built using Dockerfiles. If the Dockerfiles are accessible from a Git repository, the **oc new-app** command can create a build configuration that performs the Dockerfile build inside the OpenShift cluster and then pulls the resulting container image to the internal registry:

```
[user@host ~]$ oc new-app --as-deployment-config \
> http://gitserver.example.com/mydockerfileproject
```

OpenShift accesses the source URL to determine if it contains a Dockerfile. If the same project contains source files for programming languages, OpenShift might create a builder configuration for an S2I build instead of a Dockerfile build. To avoid ambiguity, use the **--strategy** option:

```
[user@host ~]$ oc new-app --as-deployment-config \
> --strategy docker \
> http://gitserver.example.com/mydockerfileproject
```

The following example illustrates using the **--strategy** option for an S2I build:

```
[user@host ~]$ oc new-app --as-deployment-config \
> --strategy source \
> http://gitserver.example.com/user/mygitrepo
```

Other options, such as **--image-stream** and **--code**, can be used in the same command with **--strategy**.

**Note**

The **oc new-app** command also provides some options to create applications from a template, or applications built by a Jenkins pipeline, but these options are out of scope for the current chapter.

Resources Created by the **oc new-app** Command

The **oc new-app --as-deployment-config** command adds the following resources to the current project to support building and deploying an application:

- A build configuration to build the application container image from either source code or a Dockerfile.
- An image stream pointing to either the generated image in the internal registry or to an existing image in an external registry.
- A deployment configuration using the image stream as input to create application pods.
- A service for all ports that the application container image exposes. If the application container image does not declare any exposed ports, then the **oc new-app** command does not create a service.

These resources start a series of processes which in turn create more resources in the project, such as application pods to run containerized applications.

The following command creates an application based on the **mysql** image with the label set to **db=mysql**:

```
[user@host ~]$ oc new-app --as-deployment-config \
> 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:

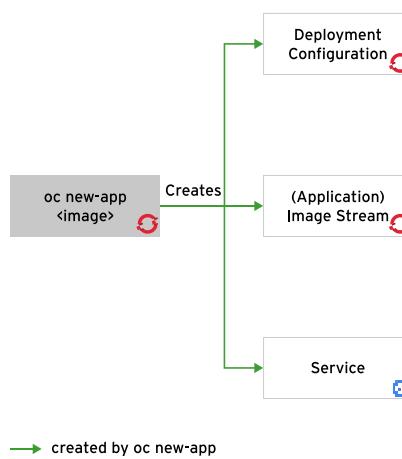


Figure 1.9: Resources created by the **oc new-app --as-deployment-config command**

The following command creates an application from source code in the PHP programming language:

```
[user@host ~]$ oc new-app --as-deployment-config \
> --name hello -i php \
> --code http://gitserver.example.com/mygitrepo
```

After the build and deployment processes complete, use the **oc get all** command to display all resources in the **test** project. The output shows a few more resources beyond those created by the **oc new-app** command:

NAME	TYPE	FROM	LATEST			
bc/hello	Source	Git	3 ①			
builds/hello-1	Source	Git@3a0af02	Complete	Started About an hour ago	DURATION 1m16s ②	
is/hello	DOCKER REPO	docker-registry.default.svc:5000/test/hello	③	TAGS	UPDATED	
dc/hello	REVISION	DESIRED	CURRENT	TRIGGERED BY config,image(hello:latest)	④	
rc/hello-1	DESIRED	CURRENT	READY	AGE 3m ⑤		
svc/hello	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE 2m31s ⑥		
po/hello-1-build	READY	STATUS	RESTARTS	AGE 2m11s ⑦		
po/hello-1-tmf1	0/1	Completed	0			
po/hello-1-tmf1	1/1	Running	0	1m23s ⑧		

- ① The build configuration created by the **oc new-app** command.
- ② The first build is triggered by the **oc new-app** command.
- ③ The image stream created by the **oc new-app** command. It points to the container image created by the S2I process.
- ④ The deployment configuration created by the **oc new-app** command.
- ⑤ The replication controller configuration created by the first deployment. Subsequent deployments might also create deployer pods.
- ⑥ The service created by the **oc new-app** command as a result of the PHP S2I builder image exposing port 8080/TCP.
- ⑦ Build pods from the most recent builds are retained by OpenShift because you might want to inspect these logs. Any deployer pods are deleted after successful termination.
- ⑧ The application pod created by the first deployment.

An application may expect a number of resources that are not created by the **oc new-app** command, such as routes, secrets, and persistent volume claims. These resources can be created using other **oc** commands before or after using the **oc new-app** command.

All resources created by the **oc new-app** command include the **app** label. The value of the **app** label matches the short name of the application Git repository or existing container image. To specify a different value for the **app** label, use the **--name** option, for example:

```
[user@host ~]$ oc new-app --as-deployment-config \
> --name test http://gitserver.example.com/mygitrepo
```

You can delete resources created by the **oc new-app** command using a single **oc delete** command and the **app** label, without resorting to deleting the entire project, and without affecting other resources that may exist in the project. The following command deletes all resources created by the previous **oc new-app** command:

```
[user@host ~]$ oc delete all -l app=test
```

Use the argument of the **--name** option to specify the base name for the resources created by the **oc new-app** command, such as build configurations and services.

The **oc new-app** command can be executed multiple times inside the same OpenShift project to create multicontainer applications one piece at a time. For example:

- Run the **oc new-app** command with the URL to a MongoDB database container image to create a database pod and a service.
- Run the **oc new-app** command with the URL to the Git repository for a Node.js application that requires access to the database, using the service created by the first invocation.

Later you can export all resources created by both commands to a template file.

If you want to inspect resource definitions without creating the resources in the current project, use the **-o** option:

```
[user@host ~]$ oc new-app --as-deployment-config \
> -o json registry.example.com/mycontainerimage
```

The resource definitions are sent to the standard output and can be redirected to a file. The resulting file can then be customized or inserted into a template definition.



Note

OpenShift provides a number of predefined templates for common scenarios such as a database plus an application. For example, the **rails-postgresql** template deploys a PostgreSQL database container image and a Ruby on Rails application built from source.

To get a complete list of options supported by the **oc new-app** command, and to see a list of examples, run the **oc new-app -h** command.

Referring to Container Images Using Image Streams and Tags

The OpenShift community recommends using *image stream* resources to refer to container images instead of using direct references to container images. An image stream resource points to a container image either in the internal registry or in an external registry, and stores metadata such as available tags and image content checksums.

Having container image metadata in an image stream allows OpenShift to perform operations, such as image caching, based on this data instead of going to a registry server every time. It also allows using either notification or pooling strategies to react to image content updates.

Build configurations and deployment configurations use image stream events to perform operations such as:

- Triggering a new S2I build because the builder image was updated.
- Triggering a new deployment of pods for an application because the application container image was updated in an external registry.

The easiest way to create an image stream is by using the **oc import-image** command with the **--confirm** option. The following example creates an image stream named **myis** for the **acme/awesome** container image that comes from the insecure registry at **registry.acme.example.com**:

```
[user@host ~]$ oc import-image myis --confirm \
> --from registry.acme.example.com:5000/acme/awesome --insecure
```

The **openshift** project provides a number of image streams for the benefit of all OpenShift cluster users. You can create your own image streams in the current project using both the **oc new-app** command as well as using OpenShift templates.

An image stream resource can define multiple *image stream tags*. An image stream tag can either point to a different container image tag or to a different container image name. This means you can use simpler, shorter names for common images, such as S2I builder images, and use different names or registries for variations of the same image. For example, the **ruby** image stream from the **openshift** project defines the following image stream tags:

- **ruby:2.5** refers to **rhel8/ruby-25** from the Red Hat Container Catalog.
- **ruby:2.6** refers to **rhel8/ruby-26** from the Red Hat Container Catalog.



References

Further information is available in the *Developer CLI commands* chapter of the *CLI reference* for Red Hat OpenShift Container Platform 4.5; at https://access.redhat.com/documentation/en-us/openshift_container_platform/4.5/html-single/cli_tools/index#cli-developer-commands

► Guided Exercise

Deploying an Application to an OpenShift Cluster

In this exercise, you will use OpenShift to build and deploy an application from a Dockerfile.

Outcomes

You should be able to create an application using the **docker** build strategy, and delete all resources from the application without deleting the project.

Before You Begin

To perform this exercise, ensure you have access to:

- A running OpenShift cluster.
- The parent image for the sample application (**ubi8/ubi**).
- The sample application in the Git repository (**ubi-echo**).

Run the following command on the **workstation** VM to validate the prerequisites and to download solution files:

```
[student@workstation ~]$ lab docker-build start
```

► 1. Inspect the Dockerfile for the sample application.

1. Enter your local clone of the **D0288-apps** Git repository and checkout the **master** branch of the course's repository to ensure you start this exercise from a known good state:

```
[student@workstation ~]$ cd D0288-apps
[student@workstation D0288-apps]$ git checkout master
...output omitted...
```

2. Create a new branch to save any changes you make during this exercise:

```
[student@workstation D0288-apps]$ git checkout -b docker-build
Switched to a new branch 'docker-build'
[student@workstation D0288-apps]$ git push -u origin docker-build
...output omitted...
 * [new branch]      docker-build -> docker-build
Branch docker-build set up to track remote branch docker-build from origin.
```

3. Review the Dockerfile for the application, inside the the **ubi-echo** folder:

```
[student@workstation D0288-apps]$ cat ubi-echo/Dockerfile
FROM registry.access.redhat.com/ubi8/ubi:8.0 ①
USER 1001 ②
CMD bash -c "while true; do echo test; sleep 5; done" ③
```

- ① The parent image is the Universal Base Image (UBI) for Red Hat Enterprise Linux 8.0 from the Red Hat Container Catalog.
- ② The user ID this container image runs as. Any nonzero value would work here. Just to make it different from standard system users, such as **apache** which are usually on the lower range of UID values.
- ③ The application runs a loop that echoes "test" every five seconds.

► 2. Build the application container image using the OpenShift cluster.

2.1. Load your classroom environment configuration.

Run the following command to load the environment variables created in the first guided exercise:

```
[student@workstation D0288-apps]$ source /usr/local/etc/ocp4.config
```

2.2. Log in to OpenShift using your developer user name:

```
[student@workstation D0288-apps]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful.
...output omitted...
```

2.3. Create a new project for the application. Prefix the project's name with your developer user name.

```
[student@workstation D0288-apps]$ oc new-project ${RHT_OCP4_DEV_USER}-docker-build
Now using project "youruser-docker-build" on server "https://
api.cluster.domain.example.com:6443".
```

2.4. Create a new application named "echo" from the Dockerfile in the **ubi-echo** folder. Use the branch you created in a previous step. It creates, among other resources, a build configuration:

```
[student@workstation D0288-apps]$ oc new-app --as-deployment-config --name echo \
> https://github.com/${RHT_OCP4_GITHUB_USER}/D0288-apps#docker-build \
> --context-dir ubi-echo
...output omitted...
--> Creating resources ...
imagestream.image.openshift.io "ubi" created
imagestream.image.openshift.io "echo" created
buildconfig.build.openshift.io "echo" created
deploymentconfig.apps.openshift.io "echo" created
--> Success
...output omitted...
```

Ignore the warnings about the base image running as root. Remember that your Dockerfile switched to a unprivileged user.

2.5. Follow the build logs:

```
[student@workstation D0288-apps]$ oc logs -f bc/echo
Cloning "https://github.com/youruser/D0288-apps#docker-build" ...
Replaced Dockerfile FROM image registry.access.redhat.com/ubi8/ubi:8.0
Caching blobs under "/var/cache/blobs".

...output omitted...
Pulling image registry.access.redhat.com/ubi8/ubi@sha256:1a2a...75b5
...output omitted...
STEP 1: FROM registry.access.redhat.com/ubi8/ubi@sha256:1a2a...75b5 ①
STEP 2: USER 1001
STEP 3: CMD bash -c "while true; do echo test; sleep 5; done"
STEP 4: ENV "OPENSHIFT_BUILD_NAME"="echo-1" ... ②
STEP 5: LABEL "io.openshift.build.commit.author"=...
STEP 6: COMMIT containers-storage:[overlay@/var/lib/containers/storage+... ③
...output omitted...
Pushing image image-registry.openshift-image-registry.svc:5000/youruser-docker-
build/echo:latest ... ④
Push successful
```

- ① The **oc new-app** command correctly identified the Git repository as a Dockerfile project and the OpenShift build performs a Dockerfile build.
- ② OpenShift appends metadata to the application container image using **ENV** and **LABEL** instructions.
- ③ OpenShift commits the application image to the node's container engine.
- ④ OpenShift pushes the application image from the node's container engine to the cluster's internal registry.

► 3. Verify that the application works inside OpenShift.

- 3.1. Wait for the application container image to deploy. Repeat the **oc status** command until the output shows a successful deployment:

```
[student@workstation D0288-apps]$ oc status
In project youruser-docker-build on server
  https://api.cluster.domain.example.com:6443

dc/echo deploys istag/echo:latest <-
  bc/echo docker builds https://github.com/youruser/D0288-apps#docker-build on
  istag/ubi:8.0
  deployment #1 deployed 6 minutes ago - 1 pod
...output omitted...
```

- 3.2. Wait for the application pod to be ready and running. Repeat the **oc get pod** command until the output is similar to the following:

```
[student@workstation D0288-apps]$ oc get pod
NAME        READY   STATUS    RESTARTS   AGE
echo-1-build 0/1     Completed  0          1m
echo-1-deploy 0/1     Completed  0          1m
echo-1-555xx  1/1     Running   0          14s
```

- 3.3. Display the application pod logs to show the application container image is producing the expected output under OpenShift. Use the application pod's name you got from the previous step.

```
[student@workstation D0288-apps]$ oc logs echo-1-555xx | tail -n 3
test
test
test
```

► 4. Inspect the build and deployment configuration to see how they relate to the image stream.

- 4.1. Review the build configuration:

```
[student@workstation D0288-apps]$ oc describe bc echo
Name:           echo
...output omitted...
Labels:         app=echo
...output omitted...
Strategy:      Docker
URL:           https://github.com/youruser/D0288-apps
Ref:           docker-build ①
ContextDir:    ubi-echo ②
From Image:    ImageStreamTag ubi:8.0 ③
Output to:     ImageStreamTag echo:latest ④
...output omitted...
```

- ① Builds start from the **docker-build** branch from the Git repository in the **URL** attribute.
- ② Builds take only the **ubi-echo** folder from the the Git repository in the **URL** attribute.
- ③ Builds take an image stream that points to the parent image from the Dockerfile so that new builds can be triggered by image changes.
- ④ Builds generate a new container image and push it to the internal registry through an image stream.

- 4.2. Review the image stream:

```
[student@workstation D0288-apps]$ oc describe is echo
Name:           echo
...output omitted...
Labels:         app=echo
...output omitted...
Image Repository: image-registry.openshift-image-registry.svc:5000/youruser-
docker-build/echo①
...output omitted...
latest
```

```
no spec tag

* image-registry.openshift-image-registry.svc:5000/youruser-docker-build/
echo@sha256:5bbf...ef0b ②
...output omitted...
```

- ① The image stream points to the OpenShift internal registry using the service DNS name.
- ② A SHA256 hash identifies the latest image. Using this hash, the image stream can detect whether the image was changed.

4.3. Review the deployment configuration:

```
[student@workstation D0288-apps]$ oc describe dc echo
Name:           echo
...output omitted...
Labels:         app=echo
Triggers:       Config, Image(echo@latest, auto=true) ①
...output omitted...
Pod Template:
...output omitted...
Containers:
echo:
  Image:      docker-registry.default.svc:5000/youruser-docker-build/
echo@sha256:5bbf...ef0b ②
...output omitted...
Deployment #1 (latest):
...output omitted...
```

- ① The deployment configuration has a trigger in the image stream. If the image stream changes, then a new deployment is performed.
- ② The pod template inside the deployment configuration specifies the SHA256 hash of the container image in order to support deployment strategies such as rolling upgrades.

▶ 5. Change the application.

5.1. Edit the **CMD** instruction in the Dockerfile at **~/D0288-apps/ubi-echo/Dockerfile** to display a counter. The final Dockerfile contents should be as follows:

```
FROM registry.access.redhat.com/ubi8/ubi:8.0
USER 1001
CMD bash -c "while true; do (( i++ )); echo test \$i; sleep 5; done"
```

5.2. Commit and push the changes to the Git server.

```
[student@workstation D0288-apps]$ cd ubi-echo
[student@workstation ubi-echo]$ git commit -a -m 'Add a counter'
...output omitted...
[student@workstation ubi-echo]$ git push
...output omitted...
[student@workstation ubi-echo]$ cd ~
[student@workstation ~]$
```

- 6. Rebuild the application and verify that OpenShift deploys the new container image.

6.1. Start a new OpenShift build:

```
[student@workstation ~]$ oc start-build echo  
build.build.openshift.io/echo-2 started
```

6.2. Follow the new build logs and wait for the build to finish:

```
[student@workstation ~]$ oc logs -f bc/echo  
...output omitted...  
Push successful
```

6.3. Verify that OpenShift starts a new deployment after the build finishes:

```
[student@workstation ~]$ oc status  
...output omitted...  
dc/echo deploys istag/echo:latest <-  
  bc/echo docker builds https://github.com/youruser/D0288-apps#docker-build on  
  istag/ubi:8.0  
    deployment #2 deployed 50 seconds ago - 1 pod  
    deployment #1 deployed 26 minutes ago  
...output omitted...
```

6.4. Wait until the new application pod is ready and running:

```
[student@workstation ~]$ oc get pod  
NAME        READY   STATUS    RESTARTS   AGE  
echo-1-build 0/1     Completed  0          27m  
echo-1-deploy 0/1     Completed  0          27m  
echo-2-build  0/1     Completed  0          1m  
echo-2-deploy 0/1     Completed  0          1m  
echo-2-p1hg   1/1     Running   0          1m
```

6.5. Display the application pod logs to show that it is running the new container image.
Use the pod's name from the previous step:

```
[student@workstation ~]$ oc logs echo-2-p1hg | head -n 3  
test 1  
test 2  
test 3
```

- 7. Compare the status of the image stream before and after rebuilding the application.

Inspect the current status of the image stream:

```
[student@workstation ~]$ oc describe is echo  
Name:      echo  
...output omitted...  
Labels:    app=echo  
...output omitted...  
latest
```

```
no spec tag

* image-registry.openshift-image-registry.svc:5000/youruser-docker-build/
echo@sha256:025a...542f ①
    2 minutes ago
image-registry.openshift-image-registry.svc:5000/youruser-docker-build/
echo@sha256:5bbf...ef0b ②
...output omitted...
```

- ① This is the new image. Notice that its SHA256 hash is different from the old image.
- ② This is the old image.

► 8. Delete all application resources.

- 8.1. Use the **oc delete** command with the application label generated by the **oc new-app** command:

```
[student@workstation ~]$ oc delete all -l app=echo
pod "echo-2-pl1hg" deleted
replicationcontroller "echo-1" deleted
replicationcontroller "echo-2" deleted
deploymentconfig.apps.openshift.io "echo" deleted
buildconfig.build.openshift.io "echo" deleted
build.build.openshift.io "echo-1" deleted
build.build.openshift.io "echo-2" deleted
imagestream.image.openshift.io "echo" deleted
imagestream.image.openshift.io "ubi" deleted
```

- 8.2. Verify that there are no resources left in the project. All resources from the single application in the project should be deleted. If the output shows a pod in the **Terminating** status, repeat the command until the pod is gone.

```
[student@workstation ~]$ oc get all
No resources found.
```

Finish

On **workstation**, run the **lab docker-build finish** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab docker-build finish
```

This concludes the guided exercise.

Managing Applications with the Web Console

Objectives

After completing this section, you should be able to use the web console to:

- Deploy an application from a binary image and manage its resources.
- View pod and build logs.
- Edit resource definitions.

Overview of the OpenShift Web Console

The OpenShift web console is a browser-based user interface that provides a graphical alternative to most common tasks required to manage OpenShift projects and applications. The functionality that the web console provides mostly focuses on developer tasks and workflow. The web console does not provide full cluster administration functionality. That functionality typically requires the use of the **oc** command.

To access the web console, use the OpenShift API URL, which for single-master clusters is usually an HTTPS URL to the master public host name.

The home page of the web console shows a list of projects that the current user can access. From the home page, you can create new projects, delete existing ones, and navigate to a project overview page.

Name	Display Name	Status	Requester
your-project	No display name	Active	youruser

Figure 1.10: Web console project listing

Expect to spend most of your time using project overview pages and the many project resource's pages. The project overview page displays summary information about applications inside the project and the status of all application pods. From the project overview page, you can navigate to resource details pages and add applications to the project.

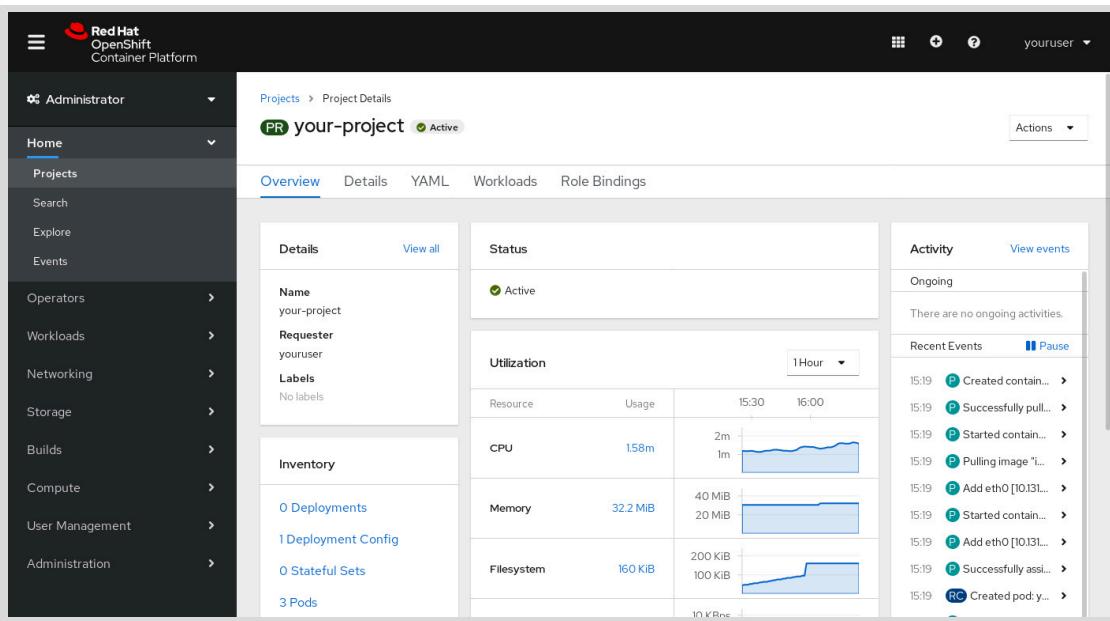


Figure 1.11: Web console project overview

Applications in OpenShift

The OpenShift web console defines an application as a set of resources that have the same value for the `app` label. The `oc new-app` command adds this label to all application resources it creates. The **+Add** section in the **Developer** perspective provides features similar to the `oc new-app` command, including adding the `app` label to resources.

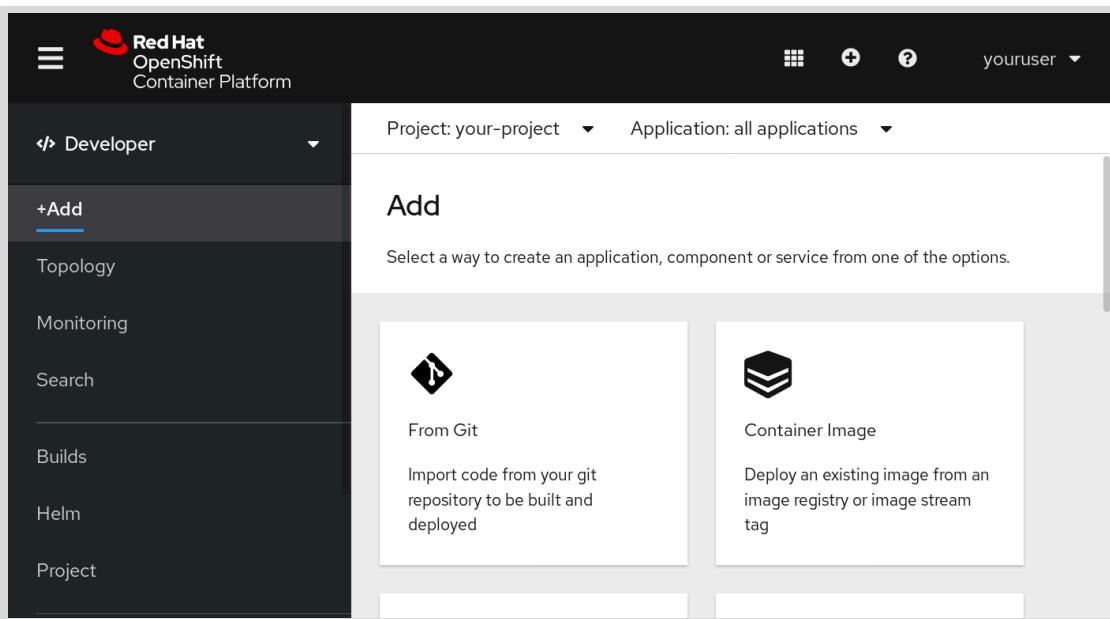


Figure 1.12: Actions available from the +Add section

The **Add** section is the entry point to an assistant that allows you to choose between S2I builder images, templates, and container images to deploy an application to the OpenShift cluster as part of a specific project. The assistant categorizes images and templates in the catalog according to labels in the container images and annotations in the templates and image stream resources.

Resource Detail Pages

Most resource detail pages list all project resources of a given kind. They provide links to delete specific resources and access the details page for each resource.

The details page for a single resource provides customized status information for each resource kind using multiple tabs. For example:

- A build details page shows the history, configuration, environment, and logs for each build.
- A deployment details page shows the history, configuration, environment, events, and logs for each deployment.
- A service details page shows the set of pods that are load-balanced by the service, and also the routes (if any) that point to the service.
- A pod details page shows the status, configuration, environment, logs, and events for each pod. It also allows opening a terminal session running a shell inside any container from the pod.

The resource details page usually lists all labels associated with that resource. OpenShift uses labels to record relationships between resources. For example, all pods created by a deployment configuration have the **deploymentconfig** label with the name of the deployment configuration.

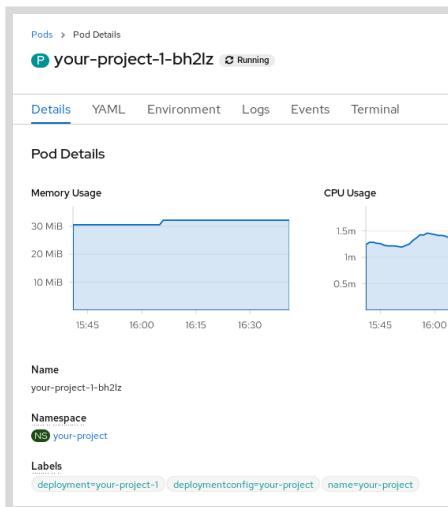


Figure 1.13: Labels at the bottom of a pod details page

Usually, when the web console displays a resource name, it is a link to that resource's details page. The navigation bar on the left side of the web console provides access to the details page for all supported kinds of resources. At the top of the navigation bar, a home icon provides access to the projects list page.

Accessing Logs with the Web Console

The Pod details pages in the web console include a **Logs** tab, which displays the logs from the pod. The container runtime collects the standard output of the containers inside a pod and stores them as pod logs.

**Note**

Only logs written to standard output inside the container are visible using the **oc logs** command or in the web console. If a containerized application saves its log events to log files instead, either in ephemeral container storage or a persistent volume, OpenShift does not display those logs in the web console, nor with the **oc logs** command.

The **Logs** tab automatically updates with the latest log entries. This tab also provides the following actions:

- Use the **Download** link to download and save the logs to a local file.
- Use the **Expand** link to make the pod logs consume the entire screen for easier viewing.

Builds and deployments are operations performed by OpenShift using builder and deployer pods. The **Build Details** page captures and stores the logs from the builder pod. Use this page to view these builder pod logs. OpenShift does not store the logs from a deployment unless there are errors during deployment.

The screenshot shows the 'Builds > Build Details' page for a build named 'your-project-1' which is 'Complete'. The page has tabs for 'Details', 'YAML', 'Environment', 'Logs' (which is selected), and 'Events'. Below the tabs, it says 'Log stream ended.' and shows a log output area with a 'Download' and 'Expand' button. The log output is as follows:

```

56 lines
Copying config sha256:c8fb8cb622514af059a359153cbf76b7db3270816d4b7ef5a48b2a0080f1dbb0
Writing manifest to image destination
Storing signatures
--> c8fb8cb622514af059a359153cbf76b7db3270816d4b7ef5a48b2a0080f1dbb0

Pushing image image-registry.openshift-image-registry.svc:5000/your-project/your-project:latest ...
Getting image source signatures
Copying blob sha256:02cb2ff840727e51a3c232637b7496e15a9a671885a64fd5e4a9dc2c1674fd
Copying blob sha256:9e7a6dc796f0a75c560158a9f9e30fb8b5a90cb53edce9ffbd5778406e4de39
Copying blob sha256:fcc5b206e9329a1674dd9e8efbe45c9be28dd0dcabbba3c6bb67a2f22fcfcf2a
Copying blob sha256:e7021e0589e07471d99c4265b7c8e64da328e49f116b5f260353b2e0a2abd373
Copying blob sha256:9a0a629e11c896fb5c92c3ede51fa50843042f978963a2a9e31288d1a7d5489
Copying config sha256:c8fb8cb622514af059a359153cbf76b7db3270816d4b7ef5a48b2a0080f1dbb0
Writing manifest to image destination
Storing signatures
Successfully pushed image-registry.openshift-image-registry.svc:5000/your-project/your-project@sha256:b3c84caa0cadaa4bd2a3a502e7b2fa12b
Push successful

```

Figure 1.14: The Logs tab on the Build Details page

Managing Builds and Deployments with the Web Console

The OpenShift web console provides features to manage both build and deployment configurations as well as all the individual builds and deployments triggered by each configuration. Much of this configuration is done on details pages that are specific to the Build Config or Deployment Config you wish to update.

The details page for a Build Config provides the **Details**, **YAML**, **Builds**, **Environment**, and **Events** tabs, as well as the **Actions** button which has a **Start Build** action. This action performs the same function as the **oc start-build** command.

If the application source-code repository is not configured to use OpenShift webhooks, you need to use either the web console or the CLI to trigger new builds after pushing updates to the application source code.

The details page for a Deployment Config provides significantly more functionality, including actions to customize various aspects of a deployment configuration, such as the desired pod count or pod storage.

The details page for a deployment also provides the **Action** button with different actions from the Build Config details page. A **Start Rollout** action is available, which performs the same function as the `oc rollout latest` command. You need to use the CLI to perform more specific `oc rollout` operations.

Editing OpenShift Resources

Most resource details pages from the OpenShift web console provide an **Actions** button that displays a menu. This menu may provide some of the following choices:

- **Edit resource:** Edit an existing resource using the raw YAML syntax, in a browser-based text editor with syntax highlighting. This action is equivalent to using the `oc edit -o yaml` command.
- **Delete resource:** Delete an existing resource. This action is equivalent to using the `oc delete` command.
- **Edit Labels:** Opens a modal dialog to edit resource labels.
- **Edit Annotations:** Opens a modal dialog to edit annotations' keys and values.

Not all resource management operations can be performed using the web console. Cluster administrator operations, in particular, usually require the CLI.



References

Further information about the web console organization, navigation and use is available in the *Web Console* guide for Red Hat OpenShift Container Platform 4.5 at https://access.redhat.com/documentation/en-us/openshift_container_platform/4.5/html-single/web_console/index

► Guided Exercise

Managing an Application with the Web Console

In this exercise, you will use the OpenShift web console to deploy an Apache HTTP Server container image.

Outcomes

You should be able to use the OpenShift web console to:

- Create a new project and add a new application that deploys a container image.
- Perform common troubleshooting tasks, such as viewing logs, inspecting resource definitions, and deleting resources.

Before You Begin

To perform this exercise, ensure you have access to:

- A running OpenShift cluster.
- The container image for the sample application (**redhattraining/php-hello-dockerfile**).

Run the following command on the **workstation** VM to validate the prerequisites:

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

- 1. Open a web browser and navigate to **https://console-openshift-console.apps.cluster.domain.example.com** to access the OpenShift web console. Log in and create a new project named **youruser-deploy-image**.
- 1.1. Find your OpenShift cluster's wildcard domain. It is the **RHT_OCP4_WILDCARD_DOMAIN** variable in the **/usr/local/etc/ocp4.config** classroom configuration file.

```
[student@workstation ~]$ grep RHT_OCP4_WILDCARD_DOMAIN /usr/local/etc/ocp4.config
RHT_OCP4_WILDCARD_DOMAIN=apps.cluster.domain.example.com
```

Another way to find the host name of your OpenShift web console is inspecting the routes on the **openshift-console** project. You must be logged into OpenShift before.

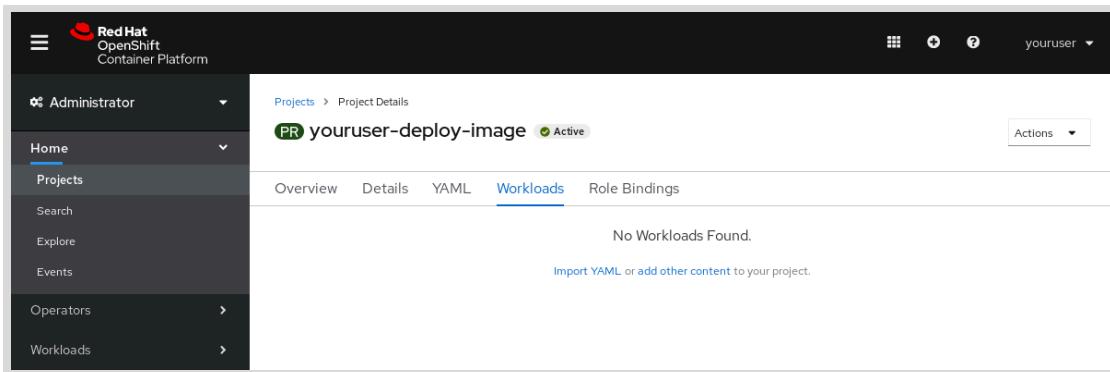
```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
[student@workstation ~]$ oc get route -n openshift-console
NAME      HOST/PORT          PATH ...
console   console-openshift-console.apps.cluster.domain.example.com ...
downloads downloads-openshift-console.apps.cluster.domain.example.com ...
```

**Note**

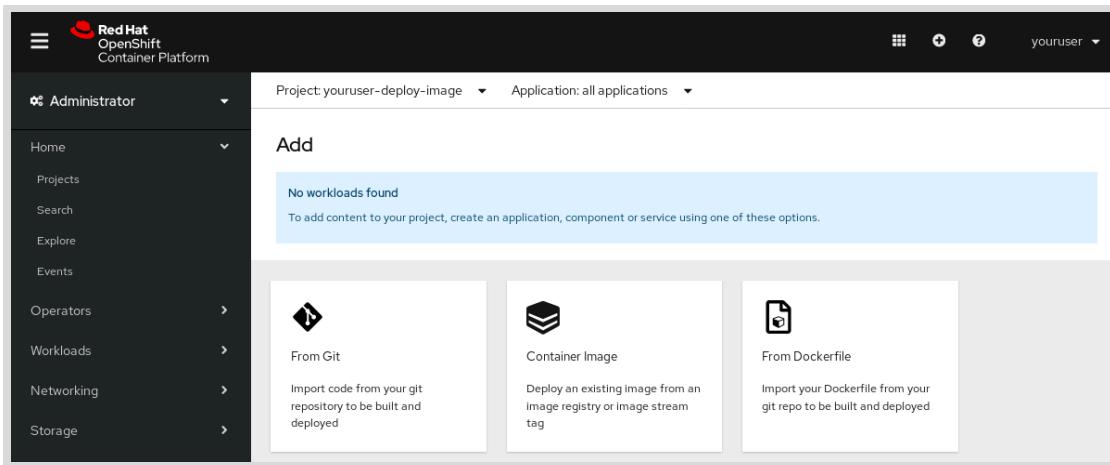
The Red Hat OpenShift Container Platform default settings do not allow regular users to access the **openshift-console** project but the classroom environment was configured to grant these permissions.

- 1.2. Open a web browser and navigate to `https://console-openshift-console.apps.cluster.domain.example.com` to access the OpenShift web console. Replace **apps.cluster.domain.example.com** with the value you got from the previous step. You should see the login page for the web console.
- 1.3. Log in as your developer user. Your user name (*youruser*) is the **RHT_OCP4_DEV_USER** variable in the `/usr/local/etc/ocp4.config` classroom configuration file. Your password is the value of the **RHT_OCP4_DEV_PASSWORD** variable in the same file.

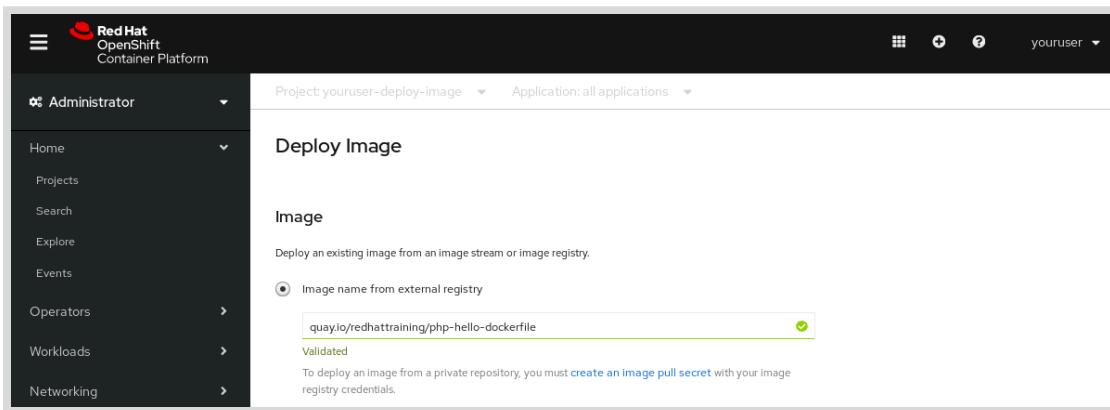
- 1.4. Navigate to the Administrator perspective, then select **Home** → **Projects** from the left side menu. First time users are placed in that page initially, Click **Create Project**. In the **Create Project** dialog box, enter **youruser-deploy-image** in the **Name** field. Replace **youruser** with the value of your **RHT_OCP4_DEV_USER** variable. You do not need to complete the display name and description fields.
Click **Create** to create the new project.
- 2. Create a new application from a prebuilt container image that contains a "Hello, World" application written in PHP and create a route to expose the application publicly.
- 2.1. On the **Project Details** page, click on the **Workloads** tab and then click on the **add other content** link.



- 2.2. On the **Add** page, click on the **Container Image** button.



- 2.3. On the **Deploy Image** page, enter **quay.io/redhattraining/php-hello-dockerfile** in the **Image Name** field. The OpenShift web console connects to the **quay.io** public registry and retrieves information about the container image.



- 2.4. Scroll down to see information about the image and replace **php-hello-dockerfile** with **hello** in the **Name** and **Application Name** fields.

The screenshot shows the 'General' configuration page for creating a new application. The 'Application Name' is set to 'hello'. In the 'Resources' section, the 'Deployment Config' option is selected. Other options like 'Deployment' and 'Builds' are also listed.

Select **Deployment Config** in the **Resources** section.

Uncheck the **Create a route to the application** option in the **Advanced Options** section at the bottom of the page.

- 2.5. Click **Create** to create the new application. The web console creates all the OpenShift resources required to deploy the container image and switches to the project's **Topology** page.

The screenshot shows the 'Topology' page for the 'hello' deployment. A large circular icon with a refresh symbol is centered, indicating the deployment status. Below the icon, there are two buttons: 'DC hello' and 'A hello'.

- 2.6. Click the icon above the deployment name to expand the deployment details and to view the number of running pods. Wait until the overview page displays a successful deployment with one pod:

The screenshot shows the Red Hat OpenShift Container Platform web interface. The left sidebar is titled 'Administrator' and includes links for Home, Projects, Search, Explore, Events, Operators, Workloads, Networking, and Storage. The main content area displays the 'hello' application under the 'youruser-deploy-image' project. The application card shows a circular icon with a blue and red logo, the name 'DC hello', and a status indicator 'A hello'. Below the card, there is a 'Health Checks' section with a note about missing health checks and a link to 'Add Health Checks'. The 'Resources' tab is selected, showing one pod named 'hello-1-rtgdgd' which is 'Running'. There is also a 'Monitoring' tab and a 'View logs' button.

2.7. In the navigation bar, click **Networking** → **Routes**.

On the **Routes** page, click the **Create Route** button.

The screenshot shows the 'Routes' page within the Red Hat OpenShift Container Platform. The left sidebar is identical to the previous screenshot. The main content area is titled 'Routes' and shows a message 'No Routes Found'. A blue 'Create Route' button is located in the top right corner of the content area.

Enter **hello-route** in the **Name** field. Enter **hello-youruser.apps.cluster.domain.example.com** in the **Hostname** field.

Replace **youruser** with the value of your **RHT_OCP4_DEV_USER** variable; that is, the user name you used to log in to OpenShift. Replace **apps.cluster.domain.example.com** with the value of your **RHT_OCP4_WILDCARD_DOMAIN** variable.

The screenshot shows the 'Create Route' form. The left sidebar has 'Networking' selected. The main form has a title 'Create Route' and a 'Edit YAML' link. It contains two fields: 'Name*' with the value 'hello-route' and 'Hostname' with the value 'hello-youruser.apps.cluster.domain.example.com'. Below the 'Hostname' field is a note: 'Public hostname for the route. If not specified, a hostname is generated.'

Scroll down and select the **hello** service from the **Service** list. From the **Target Port** list, select **8080** → **8080 (TCP)**. Do not change any other field. Scroll down and click **Create**.

- 2.8. The route details page changes to display the URL to access the application, using the new route.

Click **http://hello-youruser.apps.cluster.domain.example.com** to open a new browser tab that displays the default page returned by the PHP application. It is a simple "Hello, World" message with the PHP version.

- 3. Explore the web console troubleshooting features.

- 3.1. View the logs of the application pod.

In the navigation bar, click **Workloads** → **Pods**.

Name	Namespace	Status	Ready	Owner	Memory	CPU
hello-1-deploy	psolarvi-deploy-image	Completed	0/1	RC hello-1	-	-
hello-1-rtdgd	psolarvi-deploy-image	Running	1/1	RC hello-1	77.2 MB	0.001 cores

Click the application pod name, such as **hello-1-bmrc9** to display the **Pod Details** page. Click the **Logs** tab to see the pod logs. The warning message about the server fully qualified domain name is expected and can be safely ignored.

```
[04-Aug-2020 08:10:59] NOTICE: [pool www] 'user' directive is ignored when FPM is not running as root
[04-Aug-2020 08:10:59] NOTICE: [pool www] 'group' directive is ignored when FPM is not running as root
AH00558: httpd: Could not reliably determine the server's fully qualified domain name, using 10.131.1.68. Set
```

3.2. Start a shell session inside a running container.

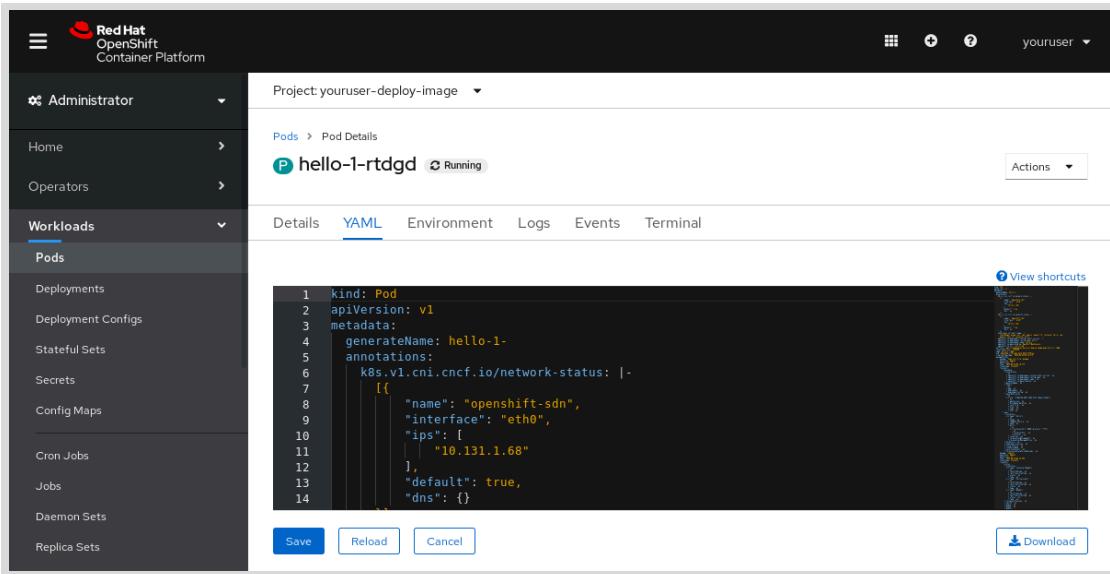
Still on the **Pod Details** page, click the **Terminal** tab to open a remote shell to the single container in the application pod. If the terminal window is too small, click **Expand** to hide the web console navigation panels. The terminal can only execute commands that exist inside the application's container image. The **ps** command is not available, but you can see the Apache HTTP Server access logs.

```
sh-4.4$ ps ax
sh: ps: command not found
sh-4.4$ sh-4.4$ cat /var/log/httpd/access_log
10.131.0.1 - - [04/Aug/2020:08:39:57 +0000] "GET / HTTP/1.1" 200 36 "-" "Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101 Firefox/68.0"
10.131.0.1 - - [04/Aug/2020:08:39:57 +0000] "GET /favicon.ico HTTP/1.1" 404 209 "-" "Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101 Firefox/68.0"
sh-4.4$ sh-4.4$ cat /var/log/php-fpm/error.log
[04-Aug-2020 08:10:59] NOTICE: [pool www] 'user' directive is ignored when FPM is not running as root
[04-Aug-2020 08:10:59] NOTICE: [pool www] 'group' directive is ignored when FPM is not running as root
[04-Aug-2020 08:10:59] NOTICE: fpm is running, pid 9
[04-Aug-2020 08:10:59] NOTICE: ready to handle connections
[04-Aug-2020 08:10:59] NOTICE: systemd monitor interval set to 10000ms
sh-4.4$
```

If necessary, click **Collapse** to show the web console navigation panels again.

3.3. View a resource definition.

Still on the **Pod Details** page, click the **YAML** tab.



The tab includes a nice web-based rich editor for YAML syntax. Do not make any changes and click **Cancel** to exit the editor.

► 4. Delete resources from the project.

- 4.1. On the **Pod Details** page, click **Actions** → **Delete Pod**. In the confirmation dialog box, click **Delete** to delete the running pod. The web console displays the **Pods** page. Wait until the **Pods** page shows that a new pod that was created by the deployment configuration to replace the deleted pod.
 - 4.2. On the navigation bar, click **Workloads** → **Deployment Configs** to view the **Deployment Configs** page. Click **hello** to view the deployment configuration details page.
In the upper-right corner, click **Actions** → **Delete Deployment Config**. In the confirmation dialog box, leave the checkbox checked and click **Delete** to delete the deployment configuration.
 - 4.3. On the navigation bar, click **Networking** → **Services** to see that there is still a service in the project.
Click **hello** to access the **Service Details** page. In the upper-right corner, click **Actions** → **Delete Service**. In the confirmation dialog box, click **Delete** to delete the service.
 - 4.4. On the navigation bar, click **Networking** → **Routes** and notice that there is still a route in the project.
Click **hello-route** to access the **Route Details** page. In the upper-right corner, click **Actions** → **Delete Route**. In the confirmation dialog box, click **Delete** to delete the route.

► 5. Delete the project.

In the upper-left corner, click **Home** → **Projects** to view the **Projects** page. Click the menu icon to the right of the **youruser-deploy-image** project, and then click **Delete Project**. Enter **youruser-deploy-image** in the confirmation dialog box, and then click **Delete** to delete the project.

Wait until the **youruser-deploy-image** project disappears from the **Projects** page. Recall that you do not need to remove application resources one by one, as you did in the previous step. Deleting a project deletes all resources inside the project.

Finish

On **workstation**, run the **lab deploy-image finish** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

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

This concludes the guided exercise.

Managing Applications with the CLI

Objectives

After completing this section, you should be able to:

- Deploy an application from source code and manage its resources using the command-line interface.
- Describe the OpenShift roles required to administer a project and a cluster.
- Describe how Source-to-Image determines the builder image for an application.

OpenShift Cluster and Project Administration Privileges

Many of the tasks involved in administering an OpenShift cluster require special administrative privileges. You may have access to an OpenShift cluster where you are the cluster administrator, but typically you do not have this level of access.

OpenShift clusters more commonly serve many different users, possibly from multiple organizations. These multinode clusters provide users with different access levels: cluster administrator, project administrator, and developer.

The default configuration for an OpenShift cluster allows any user to create new projects. A user is automatically a project administrator for any project they create. A cluster administrator can change the OpenShift cluster permissions so that users are not allowed to create projects. In this scenario, only a cluster administrator can create new projects. The cluster administrator then assigns project administrator and developer privileges to other users.

The following list summarizes the tasks that users with each access level can perform:

Cluster administrator

Manage projects, add nodes, create persistent volumes, assign project quotas, and perform other cluster-wide administration tasks.

Project administrator

Manage resources inside a project, assign resource limits, and grant other users permission to view and manage resources inside the project.

Developer

Manage a subset of a project's resources. The subset includes resources required to build and deploy applications, such as build and deployment configurations, persistent volume claims, services, secrets, and routes. A developer cannot grant to other users any permission over these resources, and cannot manage most project-level resources such as resource limits.

You perform most of the hands-on activities in this course as a **developer** user, who receives project administrator rights for the projects they create. When an activity requires cluster administrator privileges, the activity either describes how to log in as a user with cluster administrator privileges, and then perform the required administrative tasks, or it provides the administrator resources preconfigured.

**Note**

The *OpenShift Enterprise Administration I* (DO280) course covers how to perform administration tasks and how to assign cluster and project administrator permissions to users.

Troubleshooting Builds, Deployments, and Pods Using the CLI

OpenShift provides three primary mechanisms to obtain troubleshooting information about a project and its resources:

Status information

Commands such as **oc status** and **oc get** provide summary information about resources in a project. Use these commands to obtain critical information such as whether or not a build failed or if a pod is ready and running.

Resource description

The **oc describe** command displays detailed information about a resource, including its current state, configuration, and recent events. The **-o** option with the **oc get** command displays complete, low-level, configuration, and status information about a resource. Use these commands to inspect a resource and to determine if OpenShift was able to detect any specific error conditions related to the resource.

Resource logs

Runnable resources, such as pods and builds, store logs that can be viewed using the **oc logs** command. These logs are generated by the application running inside a pod, or by the build process. Use these commands to retrieve any application-specific error messages and to obtain detailed information about build errors.

When these mechanisms do not provide sufficient information, you can use the **oc cp** and **oc rsh** commands for direct interaction with a containerized application.

Comparing Two Commands You Can Use to Describe OpenShift Resources

The **oc describe** command can follow relationships between resources. For example, describing a build configuration shows information about the most recent builds. The **oc get -o** command shows information only about the requested resource. For example, running the **oc get -o** command against a build configuration does not show information about the recent builds.

Use the **oc edit** command to make changes to an OpenShift resource. The **oc edit** command combines retrieving the resource description, using the **oc get -o** command, opening the output file using a text editor, and then applying changes using the **oc apply** command.

Improving Containerized Application Logs

The usability of the application logs stored by OpenShift depends on the design of the application container image. A containerized application is expected to send all its logging output to the standard output. If the application sends its logging output to a log file, as is usual for non-containerized applications, these logs are kept inside the container ephemeral storage and are lost when the application pod is terminated.

OpenShift also provides an optional logging subsystem, based on the *EFK* stack (Elasticsearch, Fluentd, and Kibana). The logging subsystem provides long-term storage and search capabilities

for both OpenShift cluster nodes and application logs. An application might be designed to take full advantage of the OpenShift logging subsystem, or to send its log output to the standard output and let the EFK stack collect and process its logs.

Installing and configuring the OpenShift logging subsystem is outside the scope of this course.

Reading Build Logs

Build logs for a specific build can be retrieved in two ways: you can refer to the build configuration or the build resource, or you can refer to the build pod.

The following example uses a build configuration named **myapp**:

```
[user@host ~]$ oc logs bc/myapp
```

The logs from a build configuration are the logs from the latest build, whether it was successful or not.

This example uses the second build from the same build configuration:

```
[user@host ~]$ oc logs build/myapp-2
```

This example uses the build pod created to perform the same build:

```
[user@host ~]$ oc logs myapp-build-2
```

Obtaining Direct Access to a Containerized Application

If an application stores its logs in the ephemeral container storage, use the **oc cp** and **oc rsync** commands to retrieve the log files. These commands can be used to retrieve any file inside a running container file system, such as configuration files for the containerized application.

You must use the remote file path on the container file system with both the **oc cp** and **oc rsync** commands. You can store the files in the ephemeral container storage or a persistent volume mounted by the container.

For example, to retrieve the Apache HTTP Server error logs stored inside an application pod called **frontend**, use the following command:

```
[user@host ~]$ oc cp frontend-1-zvjh: /var/log/httpd/error_log \> /tmp/frontend-server.log
```

The **oc cp** command copies entire folders by default. If the source argument is a single file, the destination argument also needs to be a single file. Unlike the UNIX **cp** command, the **oc cp** command cannot copy a source file to a destination folder.

The **oc cp** command requires that the underlying application container image provides the **tar** command in order to function. If the **tar** is not installed inside the application container, the **oc cp** will fail.

The same command is used to copy files to a container file system. Use this capability to perform quick tests inside a running container. Do not use this capability to fix an issue permanently. The recommended way to fix an issue in a container is to apply the fix to the container image and the application resources and then deploy a new application pod.

The **oc rsync** command synchronizes local folders with remote folders from a running container. It uses the local **rsync** command to reduce bandwidth usage but does not require the **rsync** or **ssh** commands to be available in the container image.

If retrieving files is not sufficient to troubleshoot a running container, the **oc rsh** command creates a remote shell to execute commands inside the container. It uses the OpenShift master API to create a secure tunnel to the remote pod but does not use the **ssh** or the **rsh** UNIX commands.

The following example demonstrates running the **ps ax** command inside a pod called **frontend**:

```
[user@host ~]$ oc rsh frontend-1-zvjhbs ps ax
```



Note

Many container images do not include common UNIX troubleshooting commands such as **ps** and **ping**. The **oc rsh** command can only run commands provided by the remote container.

Add the **-t** option to the **oc rsh** command to start an interactive shell session inside the container:

```
[user@host ~]$ oc rsh -t frontend-1-zvjhbs
```



Note

The shell prompt displayed by the **oc rsh** command depends on the shell provided by the container image.

Build and Deployment Environment Variables

Many container images expect users to define environment variables to provide configuration information. For example, the MySQL database image from the Red Hat Container Catalog requires the **MYSQL_DATABASE** variable to provide the database name.

Add the **-e** option to the **oc new-app** command to provide values for environment variables. These values are stored in the deployment configuration and added to all pods created by a deployment.

Source-to-Image (S2I) builder images can also accept configuration parameters from environment variables. For example, Node.js applications often require a *npm* repository, the primary package manager for Node.js, to download the Node.js dependencies required by the application. For this reason, the Node.js S2I builder from the Container Catalog accepts either the **npm_config_registry** or the **NPM_PROXY** variable to provide a URL where the S2I builder can locate the npm module repository required to retrieve the necessary Node.js dependencies.



Note

The **npm** command, executed by the Node.js S2I builder image, requires that you provide a value for the **npm_config_registry** environment variable. The **assemble** script from the Node.js S2I builder image, which calls the **npm** command, requires that you provide a value for the **NPM_PROXY** environment variable.

S2I builder image variables are useful to avoid storing configuration information with the application sources in the Git repository. Different environments could require different configurations, for example:

- A development environment would use an npm repository server where developers can install new modules.
- A QA environment would use a different npm repository server, where a security team could vet modules before they are promoted to higher environments.

You define environment variables for an application pod using the **-e** option of the **oc new-app** command. For a builder pod, you define environment variables using the **--build-env** option of the **oc new-app** command.

Notice that a deployment configuration stores the environment variables for application pods, while a build configuration stores the environment variables for builder pods. Refer to the documentation for each builder image to find information about its build variables and their default values.



References

Further information is available in the *Understanding Containers, Images, and Imagestreams* chapter of the *Images* guide for Red Hat OpenShift Container Platform 4.5 at
https://access.redhat.com/documentation/en-us/openshift_container_platform/4.5/html/images/understanding-images

► Guided Exercise

Managing an Application with the CLI

In this exercise, you will deploy a containerized application comprised of multiple pods from a template. You will also troubleshoot and fix a deployment error.

Outcomes

You should be able to:

- Create an application from a custom template. The template deploys an application pod from PHP source code and a database pod from a MySQL server container image.
- Identify the root cause of a deployment error using the OpenShift CLI.
- Fix the error using the OpenShift CLI.

Before You Begin

To perform this exercise, ensure you have access to:

- A running OpenShift cluster.
- The S2I builder image and the database image required by the custom template (PHP 7.2 and MySQL 5.7).
- The sample application in the Git repository (**quotes**).

Run the following command on the **workstation** VM to validate the prerequisites and download the files required to complete this exercise:

```
[student@workstation ~]$ lab build-template start
```

► 1. Review the Quotes application source code.

1. Enter your local clone of the **D0288-apps** Git repository and check out the **master** branch of the course's repository to ensure you start this exercise from a known good state:

```
[student@workstation ~]$ cd ~/D0288-apps
[student@workstation D0288-apps]$ git checkout master
...output omitted...
```

2. The application is comprised of two PHP pages:

```
[student@workstation D0288-apps]$ ls ~/D0288-apps/quotes
get.php  index.php
```

The welcome page (**index.php**) displays a short description of the application. It links to another page (**get.php**) that gets a random quote from a MySQL database.

3. Review the PHP code that accesses the database:

```
[student@workstation D0288-apps]$ less ~/D0288-apps/quotes/get.php
<?php
$link = mysqli_connect($_ENV["DATABASE_SERVICE_NAME"], $_ENV["DATABASE_USER"],
$_ENV["DATABASE_PASSWORD"], $_ENV["DATABASE_NAME"]);
if (!$link) {
    http_response_code(500);
    error_log("Error: unable to connect to database\n");
    die();
}
...output omitted...
```

Press **q** to exit.

The sample application only uses standard PHP functions and uses no framework. It uses environment variables to retrieve database connection parameters and returns standard HTTP status codes if there are errors.

- ▶ 2. Inspect the custom template at **~/D0288/labs/build-template/php-mysql-ephemeral.json**. To save space, the complete contents of the custom template are not listed. You do not need to make any changes to the template; it is ready to use. The following listings review the most important parts of the template:

- 2.1. The template starts by defining a secret and a route:

```
{
  "kind": "Template",
  "apiVersion": "v1",
  "metadata": {
    "name": "php-mysql-ephemeral", ①
  ...output omitted...
  "objects": [
    {
      "apiVersion": "v1",
      "kind": "Secret", ②
  ...output omitted...
    },
    "stringData": {
      "database-password": "${DATABASE_PASSWORD}",
      "database-user": "${DATABASE_USER}"
  ...output omitted...
    },
    "spec": {
      "host": "${APPLICATION_DOMAIN}",
      "to": {
        "kind": "Service",
        "name": "${NAME}"
  ...output omitted...
```

- ① The template is a copy of the standard **cakephp-mysql-example** template, with resources and parameters specific to that framework deleted. The custom template is suitable for any simple PHP application that uses a MySQL database.

- ❷ A secret stores database login credentials and populates environment variables in both the application and the database pods. OpenShift **secrets** are explained later in this book.
 - ❸ A route provides external access to the application.
- 2.2. The template defines resources for a PHP application. The following listing focuses on the build configuration, and omits the service and image stream resources:

```
...output omitted...
{
    "apiVersion": "v1",
    "kind": "BuildConfig", ❶
...output omitted...
    "source": {
        "contextDir": "${CONTEXT_DIR}",
        "git": {
            "ref": "${SOURCE_REPOSITORY_REF}",
            "uri": "${SOURCE_REPOSITORY_URL}"
        },
        "type": "Git"
    },
    "strategy": {
        "sourceStrategy": {
            "from": {
                "kind": "ImageStreamTag", ❷
                "name": "php:7.2",
            }
        }
    }
...output omitted...
```

- ❶ A build configuration uses the S2I process to build and deploy a PHP application from source code. The build configuration and associated resources are the same as those that would be created by the **oc new-app** command on the Git repository.
- ❷ A standard OpenShift image stream provides the PHP runtime builder image.

- 2.3. The template defines resources for a MySQL database. The following listing focuses on the deployment configuration, and omits the service and image stream resources:

```
...output omitted...
{
    "apiVersion": "v1",
    "kind": "DeploymentConfig", ❶
...output omitted...
    "containers": [
...output omitted...
        "name": "mysql",
        "ports": [
            {
                "containerPort": 3306
            }
        ]
    ...output omitted...
    "volumes": [
        {
            "emptyDir": {}, ❷
            "name": "data"
        }
    ...output omitted...
```

```

        "triggers": [
...output omitted...
            "from": {
                "kind": "ImageStreamTag", ③
                "name": "mysql:5.7",
...output omitted...

```

- ➊ A deployment configuration deploys a MySQL database container. The deployment configuration and associated resources are the same as those that would be created by the `oc new-app --as-deployment-config` command on the database image.
 - ➋ The database is not backed by persistent storage. All data would be lost if the database pod is restarted.
 - ➌ A standard OpenShift image stream provides the MySQL database image.
- 2.4. Finally, the template defines a few parameters. Some of these parameters are listed below. You will use more of them.

```

...output omitted...
"parameters": [
{
    "name": "NAME",
    "displayName": "Name",
    "description": "The name assigned to all of the app objects defined in this template.",
...output omitted...
{
    "name": "SOURCE_REPOSITORY_URL", ①
    "displayName": "Git Repository URL",
    "description": "The URL of the repository with your application source code.",
...output omitted...
{
    "name": "DATABASE_USER", ②
    "displayName": "Database User",
...output omitted...

```

- ➊ Parameters provide application-specific configurations, such as the Git repository URL.
 - ➋ The template's parameters also include database configuration and connection credentials.
- ▶ 3. Install the custom template.

3.1. Load your classroom environment configuration.

Run the following command to load the environment variables created in the first guided exercise:

```
[student@workstation DO288-apps]$ source /usr/local/etc/ocp4.config
```

3.2. Log in to OpenShift using your developer user name:

```
[student@workstation D0288-apps]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful.
...output omitted...
```

- 3.3. Search for a template that uses PHP and MySQL. OpenShift provides a template based on the CakePHP framework.

This template is not adequate for the Quotes application because it adds resources and dependencies required by the framework. A simpler template is provided for this exercise; the template you reviewed in the previous step.

```
[student@workstation D0288-apps]$ oc get templates -n openshift | grep php \
> | grep mysql
cakephp-mysql-example      An example CakePHP application ...
cakephp-mysql-persistent    An example CakePHP application ...
```

- 3.4. Create a new project to host the custom template:

```
[student@workstation D0288-apps]$ oc new-project ${RHT_OCP4_DEV_USER}-common
Now using project "youruser-common" on server
"https://api.cluster.domain.example.com:6443".
...output omitted...
[student@workstation D0288-apps]$ oc create -f \
> ~/D0288/labs/build-template/php-mysql-ephemeral.json
template.template.openshift.io/php-mysql-ephemeral created
```

Red Hat recommends that you create reusable OpenShift resources, such as image streams and templates, in a shared project.

► 4. Deploy the application using the custom template.

- 4.1. Create a new project to host the application:

```
[student@workstation D0288-apps]$ oc new-project \
> ${RHT_OCP4_DEV_USER}-build-template
Now using project "youruser-build-template" on server
"https://api.cluster.domain.example.com:6443".
...output omitted...
```

- 4.2. Review the template parameters to decide which ones might be required to deploy the Quotes application. Read the description of the template's parameters:

```
[student@workstation D0288-apps]$ oc describe template php-mysql-ephemeral \
> -n ${RHT_OCP4_DEV_USER}-common
Name:  php-mysql-ephemeral
...output omitted...
Parameters:
  Name:          NAME
  Display Name:  Name
  Description:   The name assigned to all of the app objects defined in this
                 template.
```

```
Required:      true
Value:        php-app
...output omitted...
```

- 4.3. Review the **create-app.sh** script. It provides the **oc new-app** command, which uses the custom template and provides all the parameters required to deploy the Quotes application, so you do not have to type a long command:

```
[student@workstation D0288-apps]$ cat ~/D0288/labs/build-template/create-app.sh
...output omitted...
oc new-app --as-deployment-config --template ${RHT_OCP4_DEV_USER}-common/php-
mysql-ephemeral \
-p NAME=quotesapi \
-p APPLICATION_DOMAIN=quote-${RHT_OCP4_DEV_USER}.${RHT_OCP4_WILDCARD_DOMAIN} \
-p SOURCE_REPOSITORY_URL=https://github.com/${RHT_OCP4_GITHUB_USER}/D0288-apps \
-p CONTEXT_DIR=quotes \
-p DATABASE_SERVICE_NAME=quotesdb \
-p DATABASE_USER=user1 \
-p DATABASE_PASSWORD=mypa55 \
--name quotes
```

- 4.4. Run the **create-app.sh** script:

```
[student@workstation D0288-apps]$ ~/D0288/labs/build-template/create-app.sh
--> Deploying template "youruser-common/php-mysql-ephemeral" for "youruser-common/
php-mysql-ephemeral" to project youruser-build-template
...output omitted...
--> Creating resources ...
secret "quotesapi" created
service "quotesapi" created
route.route.openshift.io "quotesapi" created
imagestream.image.openshift.io "quotesapi" created
buildconfig.build.openshift.io "quotesapi" created
deploymentconfig.apps.openshift.io "quotesapi" created
service "quotesdb" created
deploymentconfig.apps.openshift.io "quotesdb" created
--> Success
...output omitted...
```

- 4.5. Follow the application build logs:

```
[student@workstation D0288-apps]$ oc logs -f bc/quotesapi
Cloning "https://github.com/youruser/D0288-apps" ...
...output omitted...
Push successful
```

- 4.6. Wait for both the application and the database pods to be ready and running. Note that the exact names of your pods may differ from those shown in the following output:

```
[student@workstation D0288-apps]$ oc get pod
NAME          READY   STATUS    RESTARTS   AGE
quotesapi-1-build   0/1     Completed   0          89s
quotesapi-1-deploy  0/1     Completed   0          35s
quotesapi-1-r6f31  1/1     Running    0          28s
quotesdb-1-deploy  0/1     Completed   0          88s
quotesdb-1-hh2g9   1/1     Running    0          80s
```

Make a note of the application and database pod names (**quotesapi-1-r6f31** and **quotesdb-1-hh2g9** in the sample output). You need them for the next steps.

- 4.7. Use the route that was created by the template to test the application's **/get.php** endpoint. It returns an HTTP error code:

```
[student@workstation D0288-apps]$ oc get route
NAME      HOST/PORT
quotesapi quote-youruser.apps.cluster.domain.example.com ...
[student@workstation D0288-apps]$ curl -si \
> http://quote-$RHT_OCP4_DEV_USER.$RHT_OCP4_WILDCARD_DOMAIN/get.php
HTTP/1.1 500 Internal Server Error
...output omitted...
```

► 5. Troubleshoot connectivity between the application and the database pod.

The application is not working as expected, but the build and deployment processes were successful. The application error might be due to an application bug, a missing prerequisite, or an incorrect configuration.

As a first troubleshooting step, verify that the application pod is connecting to the correct database pod.

- 5.1. Verify that the database service found the correct database pod. Use the database pod name you got from Step 4.6:

```
[student@workstation D0288-apps]$ oc describe svc quotesdb | grep Endpoints
Endpoints: 10.129.0.142:3306
[student@workstation ~]$ oc describe pod quotesdb-1-hh2g9 | grep IP
IP: 10.129.0.142
```

5.2. Verify the database pod login credentials:

```
[student@workstation D0288-apps]$ oc describe pod quotesdb-1-hh2g9 \
> | grep -A 4 Environment
Environment:
  MYSQL_USER:      <set to the key 'database-user' in secret 'quotesapi'>
  MYSQL_PASSWORD:  <set to the key 'database-password' in secret 'quotesapi'>
  MYSQL_DATABASE:  phpapp
Mounts:
```

- 5.3. Verify the database connection parameters in the application pod. Use the application pod name you got from Step 4.6:

```
[student@workstation D0288-apps]$ oc describe pod quotesapi-1-r6f31 \
> | grep -A 5 Environment
Environment:
  DATABASE_SERVICE_NAME:     quotesdb
  DATABASE_NAME:            phpapp
  DATABASE_USER:            <set to the key 'database-user' in secret
'quotesapi'>
  DATABASE_PASSWORD:         <set to the key 'database-password' in secret
'quotesapi'>
Mounts:
```

Notice that the environment variables that provide the database login credentials match between the two pods:

- **DATABASE_NAME** is equal to the value of **MYSQL_DATABASE**.
 - **DATABASE_USER** is set to the value of the **database-user** key in the **quotesapi** secret.
 - **DATABASE_PASSWORD** is set to the value of the **database-user** key in the **quotesapi** secret.
 - **DATABASE_SERVICE_NAME** is equal to the database service name, which is **quotesdb**.
- 5.4. Verify that the application pod can reach the database pod. The PHP S2I builder image does not provide common networking utilities, such as the **ping** command, but in this case the **curl** command provides useful output:

```
[student@workstation D0288-apps]$ oc rsh quotesapi-1-r6f31 bash -c \
> 'echo > /dev/tcp/$DATABASE_SERVICE_NAME/3306 && echo OK || echo FAIL'
OK
```

The output from the previous command proves that network connectivity is not the issue.

► 6. Review the application logs to find the root cause of the error and fix it.

- 6.1. Review the application logs.

```
[student@workstation D0288-apps]$ oc logs quotesapi-1-r6f31
AH00558: httpd: Could not reliably determine the server's fully qualified domain
name, using 10.129.0.143. Set the 'ServerName' directive globally to suppress
this message
...output omitted...
[Mon May 27 14:54:56.187516 2019] [php7:notice] [pid 52] [client
10.128.2.3:43952] SQL error: Table 'phpapp.quote' doesn't exist
10.128.2.3 - - [27/May/2019:14:54:51 +0000] "GET /get.php HTTP/1.1" 500 - "-"
"curl/7.29.0"
...output omitted...
```

The message about the server name can be safely ignored. The log entry that precedes the HTTP error code shows an SQL error. The SQL error indicates that the application cannot query the **quote** table in the **phpapp** database.

The previous step indicated that the database name is correct. The logical conclusion is that the table was not created. An SQL script to populate the database is provided as part of this exercise's files, in the `~/D0288/labs/build-template` folder.

- 6.2. Copy the SQL script to the database pod:

```
[student@workstation D0288-apps]$ oc cp ~/D0288/labs/build-template/quote.sql \
> quotesdb-1-hh2g9:/tmp/quote.sql
```

- 6.3. Run the SQL script inside the database pod:

```
[student@workstation D0288-apps]$ oc rsh -t quotesdb-1-hh2g9
sh-4.2$ mysql -u$MYSQL_USER -p$MYSQL_PASSWORD $MYSQL_DATABASE < /tmp/quote.sql
...output omitted...
sh-4.2$ exit
[student@workstation D0288-apps]$
```

- 6.4. Access the application to verify that it now works. You will get a random quote. Remember to use the route host name from Step 4.6:

```
[student@workstation D0288-apps]$ curl -si \
> http://quote-$RHT_OCP4_DEV_USER.$RHT_OCP4_WILDCARD_DOMAIN/get.php
HTTP/1.1 200 OK
...output omitted...
Always remember that you are absolutely unique. Just like everyone else.
```

You will probably see a different random message, but receiving a quote proves the application now works.

- 7. Clean up. Change to your home folder and delete the projects created during this exercise.

```
[student@workstation D0288-apps]$ cd ~
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-build-template
project.project.openshift.io "youruser-build-template" deleted
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-common
project.project.openshift.io "youruser-common" deleted
```

Finish

On **workstation**, run the **lab build-template finish** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab build-template finish
```

This concludes the guided exercise.

► Lab

Deploying and Managing Applications on an OpenShift Cluster

Performance Checklist

In this lab, you will deploy an application to an OpenShift cluster from source code. The application build configuration file has an error that you will troubleshoot and fix.



Note

The **grade** command at the end of each chapter lab requires that you use the exact project names and other identifiers, as stated in the specification of the lab.

Outcomes

You should be able to:

- Create an application using the source build strategy.
- Review the build logs to find information about a build error.
- Update the application build tool configuration to fix the build error.
- Rebuild the application and verify that it deploys successfully.

Before You Begin

To perform this exercise, ensure you have access to:

- A running OpenShift cluster.
- The S2I builder image for Node.js 12 applications.
- The application in the Git repository (**nodejs-helloworld**).

Run the following command on **workstation** to validate the prerequisites. The command also downloads helper files and solution files for the review lab:

```
[student@workstation ~]$ lab source-build start
```

Requirements

The provided application is written in JavaScript, using the Node.js runtime. It is a “hello, world” application based on the Express framework. Build and deploy the application to an OpenShift cluster according to the following requirements:

- Application code is deployed from a new branch, named **source-build**.
- The project name for OpenShift is **youruser-source-build**.

- The application name for OpenShift is **greet**.
 - The application should be accessible from the default route:
greet-youruser-source-build.apps.cluster.domain.example.com
 - The Git repository that contains the application directory sources is:
https://github.com/yourgithubuser/D0288-apps/nodejs-helloworld.
 - Npm modules required to build the application are available from:
http://nexus-common.apps.cluster.domain.example.com/repository/nodejs
- Use the **npm_config_registry** build environment variable to pass this information to the S2I builder image for Node.js.
- You can use the **python -m json.tool filename.json** command to identify syntax errors in JSON files.

Steps

1. Navigate to your local clone of the **D0288-apps** Git repository and create a new branch named **source-build** from the master branch. Deploy the application in the **nodejs-helloworld** folder to the **youruser-source-build** project on the OpenShift cluster.
2. Display the build logs to identify the build error and inspect the application sources to determine the root cause.
Recall that you can use the **python -m json.tool filename** command to validate a JSON file.
3. Fix the error in the build tool configuration file and push the changes to the Git repository.
4. Start a new build of the application and verify that the application deploys successfully.
5. Verify that the application logs show no errors, expose the application for external access, and verify that the application returns a “hello, world” message.

Evaluation

As the **student** user on the **workstation** machine, use the **lab** command to grade your work. Correct any reported failures and rerun the command until successful.

```
[student@workstation ~]$ lab source-build grade
```

Finish

On **workstation**, run the **lab source-build finish** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab source-build finish
```

This concludes the lab.

► Solution

Deploying and Managing Applications on an OpenShift Cluster

Performance Checklist

In this lab, you will deploy an application to an OpenShift cluster from source code. The application build configuration file has an error that you will troubleshoot and fix.



Note

The **grade** command at the end of each chapter lab requires that you use the exact project names and other identifiers, as stated in the specification of the lab.

Outcomes

You should be able to:

- Create an application using the source build strategy.
- Review the build logs to find information about a build error.
- Update the application build tool configuration to fix the build error.
- Rebuild the application and verify that it deploys successfully.

Before You Begin

To perform this exercise, ensure you have access to:

- A running OpenShift cluster.
- The S2I builder image for Node.js 12 applications.
- The application in the Git repository (**nodejs-helloworld**).

Run the following command on **workstation** to validate the prerequisites. The command also downloads helper files and solution files for the review lab:

```
[student@workstation ~]$ lab source-build start
```

Requirements

The provided application is written in JavaScript, using the Node.js runtime. It is a “hello, world” application based on the Express framework. Build and deploy the application to an OpenShift cluster according to the following requirements:

- Application code is deployed from a new branch, named **source-build**.
- The project name for OpenShift is **youruser-source-build**.
- The application name for OpenShift is **greet**.

- The application should be accessible from the default route:

greet-youruser-source-build.apps.cluster.domain.example.com

- The Git repository that contains the application directory sources is:

https://github.com/yourgithubuser/D0288-apps/nodejs-helloworld.

- Npm modules required to build the application are available from:

http://nexus-common.apps.cluster.domain.example.com/repository/nodejs

Use the **npm_config_registry** build environment variable to pass this information to the S2I builder image for Node.js.

- You can use the **python -m json.tool filename.json** command to identify syntax errors in JSON files.

Steps

1. Navigate to your local clone of the **D0288-apps** Git repository and create a new branch named **source-build** from the master branch. Deploy the application in the **nodejs-helloworld** folder to the **youruser-source-build** project on the OpenShift cluster.

- 1.1. Prepare the lab environment.

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

- 1.2. Log in to OpenShift and create the project:

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful
...output omitted...
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-source-build
...output omitted...
```

- 1.3. Enter your local clone of the **D0288-apps** Git repository and check out the **master** branch of the course's repository to ensure you start this exercise from a known good state:

```
[student@workstation ~]$ cd D0288-apps
[student@workstation D0288-apps]$ git checkout master
...output omitted...
```

- 1.4. Create a new branch where you can save any changes you make during this exercise:

```
[student@workstation D0288-apps]$ git checkout -b source-build
Switched to a new branch 'source-build'
[student@workstation D0288-apps]$ git push -u origin source-build
...output omitted...
* [new branch]      source-build -> source-build
Branch source-build set up to track remote branch source-build from origin.
```

- 1.5. Create a new application from sources in the Git repository. Name the application **greet**. Use the **--build-env** option with the **oc new-app --as-deployment-config** command to define the build environment variable with the npm modules URL. Either copy or execute the command from the **oc-new-app.sh** script in the **/home/student/D0288/labs/source-build** folder:

```
[student@workstation D0288-apps]$ oc new-app --as-deployment-config --name greet \
> --build-env npm_config_registry=\
> http://${RHT_OCP4_NEXUS_SERVER}/repository/nodejs \
> nodejs:12-https://github.com/${RHT_OCP4_GITHUB_USER}/D0288-apps#source-build \
> --context-dir nodejs-helloworld
...output omitted...
--> Creating resources ...
imagestream.image.openshift.io "greet" created
buildconfig.build.openshift.io "greet" created
deploymentconfig.apps.openshift.io "greet" created
service "greet" created
--> Success
...output omitted...
```

**Note**

There is no space before or after the equals sign (=) after **npm_config_registry**. The complete **key=value** pair for the build environment variable is too long for the page width.

2. Display the build logs to identify the build error and inspect the application sources to determine the root cause.

Recall that you can use the **python -m json.tool filename** command to validate a JSON file.

- 2.1. Follow the build logs:

```
[student@workstation D0288-apps]$ oc logs -f bc/greet
```

You should see a JSON parse error message:

```
...output omitted...
--> Installing all dependencies
npm ERR! code EJSONPARSE
npm ERR! file /opt/app-root/src/package.json
npm ERR! JSON.parse Failed to parse json
npm ERR! JSON.parse Unexpected string in JSON at position 241 while parsing '{'
npm ERR! JSON.parse     "name": "nodejs-helloworld",
npm ERR! JSON.parse     "vers"
npm ERR! JSON.parse Failed to parse package.json data.
npm ERR! JSON.parse package.json must be actual JSON, not just JavaScript.
...output omitted...
```

The Node.js builder image does not provide a specific error location inside the **package.json** build tool configuration file.

- 2.2. Use the **python -m json.tool** command to validate the JSON file:

```
[student@workstation D0288-apps]$ python -m json.tool \
> nodejs-helloworld/package.json
```

You should see the following error message:

```
Expecting : delimiter: line 12 column 15 (char 241)
```

- 2.3. Open the **~/D0288-apps/nodejs-helloworld/package.json** build tool configuration file with a text editor and identify the syntax error. The following partial listing shows that a colon (:) is missing after the **express** key:

```
"dependencies": {
    "express" "4.14.x"
}
```

3. Fix the error in the build tool configuration file and push the changes to the Git repository.

- 3.1. Edit the **~/D0288-apps/nodejs-helloworld/package.json** source file to add a colon (:) after the **express** key. The final file contents should be as follows:

```
"dependencies": {
    "express": "4.14.x"
}
```

- 3.2. Commit and push the fixes to the Git repository:

```
[student@workstation D0288-apps]$ cd nodejs-helloworld
[student@workstation nodejs-helloworld]$ git commit -a -m 'Fixed JSON syntax'
...output omitted...
[student@workstation nodejs-helloworld]$ git push
...output omitted...
[student@workstation nodejs-helloworld]$ cd ~
[student@workstation ~]$
```

4. Start a new build of the application and verify that the application deploys successfully.

- 4.1. Start a new build of the greet application and follow its logs. Wait for the build to finish without errors:

```
[student@workstation ~]$ oc start-build --follow bc/greet
build "greet-2" started
...output omitted...
Push successful
```

- 4.2. Verify that a new deployment starts:

```
[student@workstation ~]$ oc status
...output omitted...
svc/greet - 172.30.160.185:8080
dc/greet deploys istag/greet:latest <
bc/greet source builds https://github.com/yourgithubuser/D0288-apps#source-
build on openshift/nodejs:10
  deployment #1 deployed 23 seconds ago - 1 pod
...output omitted...
```

4.3. Wait until the application pod is ready and running:

```
[student@workstation ~]$ oc get pod
NAME      READY   STATUS    RESTARTS   AGE
greet-1-build  0/1     Error     0          4m
greet-1-deploy  0/1     Completed  0          14m
greet-2-build  0/1     Completed  0          2m
greet-1-gf59d  1/1     Running   0          59s
```

5. Verify that the application logs show no errors, expose the application for external access, and verify that the application returns a “hello, world” message.

5.1. Access the application logs. You should see no error messages from the application.

```
[student@workstation ~]$ oc logs greet-1-gf59d
...output omitted...
Example app listening on port 8080!
```

5.2. Expose the application to external access:

```
[student@workstation ~]$ oc expose svc/greet
route.route.openshift.io/greet exposed
```

5.3. Get the host name generated by OpenShift for the new route:

```
[student@workstation ~]$ oc get route
NAME      HOST/PORT
greet     greet-youruser-source-build.apps.cluster.domain.example.com ...
```

5.4. Send an HTTP request to the application using the **curl** command and the host name from the previous step. It returns a “hello, world” message:

```
[student@workstation ~]$ curl \
> http://greet-${RHT_OCP4_DEV_USER}-source-build.${RHT_OCP4_WILDCARD_DOMAIN}
Hello, World!
```

Evaluation

As the **student** user on the **workstation** machine, use the **lab** command to grade your work. Correct any reported failures and rerun the command until successful.

```
[student@workstation ~]$ lab source-build grade
```

Finish

On **workstation**, run the **lab source-build finish** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab source-build finish
```

This concludes the lab.

Summary

In this chapter, you learned that:

- RHOCP provides a PaaS tool running on Red Hat CoreOS and Kubernetes.
- OpenShift supports building container images from application source code or directly from Dockerfiles, using the S2I process.
- Build and deployment configuration resources automate the build and deployment processes, and can automatically react to changes in application source code or updates made to container images.
- The **oc new-app** command can detect the source programming language used by an application in a Git repository automatically. It also provides a number of disambiguation options.
- Useful **oc** subcommands for troubleshooting builds and deployments are **get**, **describe**, **edit**, **logs**, **cp**, and **rsh**.
- Developers might not have cluster administrator privileges to their development environment, or they could have only project administration or edit privileges over a subset of all the projects in the OpenShift cluster.

Chapter 2

Designing Containerized Applications for OpenShift

Goal

Select an application containerization method for an application and package it to run on an OpenShift cluster.

Objectives

- Select an appropriate application containerization method.
- Build a container image with advanced Dockerfile directives.
- Select a method for injecting configuration data into an application and create the necessary resources to do so.

Sections

- Selecting a Containerization Approach (and Quiz)
- Building Container Images with Advanced Dockerfile Directives (and Guided Exercise)
- Injecting Configuration Data into an Application (and Guided Exercise)

Lab

Designing Containerized Applications for OpenShift

Selecting a Containerization Approach

Objectives

After completing this section, you should be able to select an appropriate application containerization method.

Selecting a Build Method

The primary deployment unit in OpenShift is a *container image*, also referred to as just an *image*. A container image consists of the application and all its dependencies, such as shared libraries, runtime environments, interpreters, and so on. There are several ways to create container images, depending on the type of application that you are planning to deploy and run on an OpenShift cluster:

Container images

Container images built outside of OpenShift can be deployed directly on an OpenShift cluster. This approach is useful in cases where you already have an application packaged as a container image. You can also use this approach when you have a certified and supported container image provided by a third-party vendor. You can deploy images built by a third-party vendor to an OpenShift cluster.

Dockerfiles

In some instances, the Dockerfile used to build the application container image is provided to you. In such scenarios, you have more options to consider:

- You can customize the Dockerfile and build new images to suit the needs of your application. Use this option when the changes are small and if you do not want to add too many layers to the image.
- You can create a Dockerfile using the provided container image as a parent and customize the base image to suit the needs of your application. Use this option when you want to create a new child image with more customizations, and that inherits the layers from the parent image.

Source-to-Image (S2I) builder images

An S2I builder image contains base operating system libraries, compilers and interpreters, runtimes, frameworks, and Source-to-Image tooling. When an application is built using this approach, OpenShift combines the application source code and the builder image to create a ready-to-run container image that OpenShift can then deploy to the cluster. This approach has several advantages for developers and is the quickest way to build new applications for deployment to an OpenShift cluster.

Depending on the needs of your application, you have several options to use S2I builder images:

- Red Hat provides many supported S2I builder images for various types of applications. Red Hat recommends that you use one of these standard S2I builder images whenever possible.
- S2I builder images are like regular container images, but with extra metadata, scripts, and tooling included in the image. You can use Dockerfiles to create child images based on the parent builder images provided by Red Hat.

- If none of the standard Red Hat-provided S2I builder images suit your application needs, you can build your own customized S2I builder image.

S2I Builder Images

An S2I builder image is a specialized form of container image that produces application container images as output. Builder images include base operating system libraries, language runtimes, frameworks, and libraries that an application depends on, and Source-to-Image tools and utilities.

For example, if you have an application written in PHP that you want to deploy to OpenShift, you can use a PHP builder image to produce an application container image. You provide the location of the Git repository where you keep the application source code, and OpenShift combines the source code and the base builder image to produce a container image that OpenShift can deploy to the cluster. The resulting application container image includes a version of Red Hat Enterprise Linux, a PHP runtime, and the application. Builder images are a convenient mechanism to go from code to runnable container quickly and easily without the need to create Dockerfiles.

Using Container Images

Although Source-to-Image builds are the preferred way to build and deploy applications to OpenShift, there may be scenarios where you need to deploy applications that a third party provides to you prebuilt. For example, certain vendors provide fully certified and supported container images that are ready to run. In such cases, OpenShift supports deployments of prebuilt container images.

The **oc new-app** command provides several flexible ways to deploy container images to an OpenShift cluster. The most straightforward approach is to make the prebuilt image available via a public (such as docker.io or quay.io) or private (hosted within your organization) registry, and then provide the location of this image to the **oc new-app** command. OpenShift then pulls the image and deploys it to an OpenShift cluster like any other image built within OpenShift.



Note

A sample tutorial that demonstrates prebuilt container image deployments to an OpenShift cluster is available at OpenShift Binary Deployments [<https://blog.openshift.com/binary-deploymentsOpenshift-3/>]

The *Red Hat Container Catalog* is a trusted source of secure, certified, and up-to-date container images. It contains both plain container images as well as S2I builder images. Mission-critical applications require trusted containers. Red Hat builds the Container Catalog container images from RPM resources that have been vetted by Red Hat's internal security team and hardened against security flaws.

The Red Hat Container Catalog portal at <https://registry.redhat.io> provides information about a number of container images that Red Hat builds on versions of Red Hat Enterprise Linux (RHEL) and related systems. It provides a number of ready-to-use container images to start developing applications for OpenShift.

Red Hat uses a *Container Health Index* for security risk assessment of container images that Red Hat provides through the Red Hat Container Catalog. For more details on the grading system used by Red Hat in the Container Health Index, see <https://access.redhat.com/articles/2803031>.

For more details on the Red Hat Container Catalog, see Frequently Asked Questions (FAQ) [<https://access.redhat.com/containers/#/faq>]

Selecting Container Images for Applications

Selecting a container image for your application depends on a number of factors. If you want to build a custom container, start with a base operating system image (such as **rhel7**). To build and run applications requiring specific development tools and runtime libraries, Red Hat provides container images that feature tools such as Node.js (**rhscl/nodejs-8-rhel7**), Ruby on Rails (**rhscl/ror-50-rhel7**), and Python (**rhscl/python-36-rhel7**).

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 Red Hat Enterprise Linux (RHEL) release schedule. Red Hat maintains many container images offered through the Red Hat Container Catalog as part of the RHSCl.

Red Hat also provides Red Hat OpenShift Application Runtimes (RHOAR), which is Red Hat's development platform for cloud-native and microservices applications. RHOAR provides a Red Hat optimized and supported approach for developing microservices applications that target OpenShift as the deployment platform.

RHOAR supports multiple runtimes, languages, frameworks, and architectures. It offers the choice and flexibility to pick the right frameworks and runtimes for the right job. Applications developed with RHOAR can run on any infrastructure where Red Hat OpenShift Container Platform can run, offering freedom from vendor lock-in.

RHOAR provides:

- Access to Red Hat-built and supported binaries for selected microservices development frameworks and runtimes.
- Access to Red Hat-built and supported binaries for integration modules that replace or enhance a framework's implementation of a microservices pattern to use OpenShift features.
- Developer support for writing applications using selected microservice development frameworks, runtimes, integration modules, and integration with selected external services, such as database servers.
- Production support for deploying applications using selected microservice development frameworks, runtimes, integration modules, and integrations on a supported OpenShift cluster.

Creating S2I Builder Images

If you want your applications to use a custom S2I builder image with your own custom set of runtimes, scripts, frameworks, and libraries, you can build your own S2I builder image. Several options exist for creating S2I builder images:

- Create your own S2I builder image from scratch. In scenarios where your application cannot use the S2I builder images provided by the Container Catalog as-is, you can build a custom S2I builder image that customizes the build process to suit your application's needs.

OpenShift provides the **s2i** command-line tool that helps you bootstrap the build environment for creating custom S2I builder images. It is available in the *source-to-image* package from the RHSCl Yum repositories (**rhel-server-rhscl-7-rpms**).

- Fork an existing S2I builder image. Rather than starting from scratch, you can use the Dockerfiles for the existing builder images in the Container Catalog, which are available at <https://github.com/sclorg/?q=s2i>, and customize them to suit your needs.

- Extend an existing S2I builder image. You can also extend an existing builder image by creating a child image and then adding or replacing content from the existing builder image.

A good tutorial that walks through building a custom S2I builder image is available at <https://blog.openshift.com/create-s2i-builder-image/>.



References

Red Hat Container Catalog

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

Dockerfiles for images that are part of the Red Hat Software Collections library are available at

<https://github.com/sclorg?q=-container>

Further information about container images that Red Hat supports for use with OpenShift is in the *Creating Images* chapter of the *Images* guide for Red Hat OpenShift Container Platform 4.5; at

https://access.redhat.com/documentation/en-us/openshift_container_platform/4.5/html/images/creating-images

► Quiz

Selecting a Containerization Approach

Choose the correct answers to the following questions:

When you have completed the quiz, click **CHECK**. If you want to try again, click **RESET**. Click **SHOW SOLUTION** to see all of the correct answers.

- ▶ 1. You have been asked to deploy a commercial, third-party, .NET-based application to an OpenShift cluster, which is packaged by the vendor as a container image. Which of the following options would you use to deploy the application?
 - a. A Source-to-Image build.
 - b. A custom Source-to-Image builder.
 - c. Stage the container image in a private container registry, and then deploy the container image to an OpenShift cluster using the OpenShift **oc** command-line tool.
 - d. None of these. You cannot deploy .NET-based applications on an OpenShift cluster.

- ▶ 2. You are tasked with deploying a RHEL 7-based, custom, in-house-developed C++ application to an OpenShift cluster. You will be given the full source code of the application. Which two of the following options, based on best practices, can be used to build a container image to deploy to an OpenShift cluster? (Choose two.)
 - a. Create a Dockerfile using the **rhel7** container image from the Red Hat Container Catalog as the base for the application and provide it to OpenShift using a Git repository. OpenShift can create a container image from the provided Dockerfile.
 - b. Download a CentOS 7/C++ based container image from a public registry such as docker.io and create a Dockerfile that compiles and packages the application. Because CentOS 7-based binaries are binary-compatible with RHEL 7, the application will work correctly when deployed to an OpenShift cluster.
 - c. Create a Dockerfile using the RHEL 7 container image from the Red Hat Container Catalog as the base and then build a container image based on the Dockerfile. Deploy the container image to an OpenShift cluster using the OpenShift **oc** command-line tool.
 - d. Create a Dockerfile using any Linux-based container image from a public registry as a base, and then build a container image based on the Dockerfile. Deploy the container image to an OpenShift cluster using the OpenShift **oc** command-line tool.

- **3. You have been asked to migrate and deploy two applications to an OpenShift cluster, based on Ruby on Rails and Node.js, respectively. These applications were previously running in an environment using virtual machines. You have been provided with the location of the Git repositories where the application source code is located for both applications. Which of the following options is recommended to deploy the applications to an OpenShift cluster, keeping in mind that you have been asked to further enhance these applications with more features and to continue development in an OpenShift environment?**
- a. Use the Ruby on Rails and Node.js container images from docker.io as a base. Create custom Dockerfiles for each of these two applications and build container images from them. Deploy the images to an OpenShift cluster using the standard binary deployment process.
 - b. Create custom S2I-based builder images, because there are no builder images available for Ruby on Rails and Node.js in OpenShift.
 - c. Use the Ruby on Rails and Node.js S2I builder images from the Red Hat Container Catalog, and deploy the applications to an OpenShift cluster using a standard S2I build process.
 - d. Use the plain Ruby and Node.js-based container images from the Red Hat Container Catalog and create custom Dockerfiles for each of them. Build and deploy the resulting container images to an OpenShift cluster.

► 4. You have been asked to deploy a web application written in the Go programming language to an OpenShift cluster. Your organization's security team has mandated that all applications be run through a static source code analysis system and a suite of automated unit and integration tests before deploying to production environments. The security team has also provided a custom Dockerfile that ensures that all applications are deployed on a RHEL 7-based operating system, based on the standard RHEL 7 image from the Red Hat Container Catalog. Their environment includes a curated list of packages, users, and customized configuration for the core services in the operating system. Furthermore, the application architect has insisted that there be clear separation between source-code level changes and infrastructure changes (operating system, Go compiler, and Go tools). Changes and patches to the operating system or the Go runtime layers should automatically trigger a rebuild and redeployment of the application.

Which of the following options would you use to achieve this objective?

- a. Create a custom Dockerfile that builds an application container image consisting of the RHEL 7 operating system base, the Go runtime, and the analysis tool. Deploy the resulting image to an OpenShift cluster using the binary deployment process.
- b. Create separate Dockerfiles for the RHEL 7 operating system base, the Go runtime, and the analysis tool. OpenShift can automatically merge the Dockerfiles to produce a single runnable application container image.
- c. Create a custom S2I builder image for this application that embeds the static analysis tool, the Go compiler and runtime, and the RHEL 7 operating system image based on the Dockerfile.
- d. Create separate container images for the RHEL 7 operating system base, the Go runtime, and the analysis tool. After these images have been staged in a private or public container image registry, OpenShift can automatically concatenate the layers from each individual image to create the final runnable application container image.
- e. None of these. This requirement cannot be met and the application cannot be deployed on an OpenShift cluster.

► Solution

Selecting a Containerization Approach

Choose the correct answers to the following questions:

When you have completed the quiz, click **CHECK**. If you want to try again, click **RESET**. Click **SHOW SOLUTION** to see all of the correct answers.

- ▶ 1. You have been asked to deploy a commercial, third-party, .NET-based application to an OpenShift cluster, which is packaged by the vendor as a container image. Which of the following options would you use to deploy the application?
 - a. A Source-to-Image build.
 - b. A custom Source-to-Image builder.
 - c. Stage the container image in a private container registry, and then deploy the container image to an OpenShift cluster using the OpenShift **oc** command-line tool.
 - d. None of these. You cannot deploy .NET-based applications on an OpenShift cluster.

- ▶ 2. You are tasked with deploying a RHEL 7-based, custom, in-house-developed C++ application to an OpenShift cluster. You will be given the full source code of the application. Which two of the following options, based on best practices, can be used to build a container image to deploy to an OpenShift cluster? (Choose two.)
 - a. Create a Dockerfile using the **rhel7** container image from the Red Hat Container Catalog as the base for the application and provide it to OpenShift using a Git repository. OpenShift can create a container image from the provided Dockerfile.
 - b. Download a CentOS 7/C++ based container image from a public registry such as docker.io and create a Dockerfile that compiles and packages the application. Because CentOS 7-based binaries are binary-compatible with RHEL 7, the application will work correctly when deployed to an OpenShift cluster.
 - c. Create a Dockerfile using the RHEL 7 container image from the Red Hat Container Catalog as the base and then build a container image based on the Dockerfile. Deploy the container image to an OpenShift cluster using the OpenShift **oc** command-line tool.
 - d. Create a Dockerfile using any Linux-based container image from a public registry as a base, and then build a container image based on the Dockerfile. Deploy the container image to an OpenShift cluster using the OpenShift **oc** command-line tool.

- **3. You have been asked to migrate and deploy two applications to an OpenShift cluster, based on Ruby on Rails and Node.js, respectively. These applications were previously running in an environment using virtual machines. You have been provided with the location of the Git repositories where the application source code is located for both applications. Which of the following options is recommended to deploy the applications to an OpenShift cluster, keeping in mind that you have been asked to further enhance these applications with more features and to continue development in an OpenShift environment?**
- a. Use the Ruby on Rails and Node.js container images from docker.io as a base. Create custom Dockerfiles for each of these two applications and build container images from them. Deploy the images to an OpenShift cluster using the standard binary deployment process.
 - b. Create custom S2I-based builder images, because there are no builder images available for Ruby on Rails and Node.js in OpenShift.
 - c. Use the Ruby on Rails and Node.js S2I builder images from the Red Hat Container Catalog, and deploy the applications to an OpenShift cluster using a standard S2I build process.
 - d. Use the plain Ruby and Node.js-based container images from the Red Hat Container Catalog and create custom Dockerfiles for each of them. Build and deploy the resulting container images to an OpenShift cluster.

► **4. You have been asked to deploy a web application written in the Go programming language to an OpenShift cluster. Your organization's security team has mandated that all applications be run through a static source code analysis system and a suite of automated unit and integration tests before deploying to production environments. The security team has also provided a custom Dockerfile that ensures that all applications are deployed on a RHEL 7-based operating system, based on the standard RHEL 7 image from the Red Hat Container Catalog. Their environment includes a curated list of packages, users, and customized configuration for the core services in the operating system.**
Furthermore, the application architect has insisted that there be clear separation between source-code level changes and infrastructure changes (operating system, Go compiler, and Go tools). Changes and patches to the operating system or the Go runtime layers should automatically trigger a rebuild and redeployment of the application.

Which of the following options would you use to achieve this objective?

- a. Create a custom Dockerfile that builds an application container image consisting of the RHEL 7 operating system base, the Go runtime, and the analysis tool. Deploy the resulting image to an OpenShift cluster using the binary deployment process.
- b. Create separate Dockerfiles for the RHEL 7 operating system base, the Go runtime, and the analysis tool. OpenShift can automatically merge the Dockerfiles to produce a single runnable application container image.
- c. Create a custom S2I builder image for this application that embeds the static analysis tool, the Go compiler and runtime, and the RHEL 7 operating system image based on the Dockerfile.
- d. Create separate container images for the RHEL 7 operating system base, the Go runtime, and the analysis tool. After these images have been staged in a private or public container image registry, OpenShift can automatically concatenate the layers from each individual image to create the final runnable application container image.
- e. None of these. This requirement cannot be met and the application cannot be deployed on an OpenShift cluster.

Building Container Images with Advanced Dockerfile Instructions

Objectives

After completing this section, you should be able to:

- Build containerized applications with the Red Hat Universal Base Image.
- Build a container image with advanced Dockerfile instructions.

Introducing the Red Hat Universal Base Image

The Red Hat Universal Base Image (*UBI*) aims to be a high-quality, flexible base container image for building containerized applications. The goal of the Red Hat Universal Base Image is to allow users to build and deploy containerized applications using a highly supportable, enterprise-grade container base image that is lightweight and performant. You can run containers built using the Universal Base Image on Red Hat platforms as well as non-Red Hat platforms.

Red Hat derives the Red Hat Universal Base Image from Red Hat Enterprise Linux (RHEL). It does differ from the existing RHEL 7 base images, most notably the fact that it can be redistributed under the terms of the Red Hat Universal Base Image End User License Agreement (EULA) that allows Red Hat's partners, customers, and community members to standardize on well-curated enterprise software and tools to deliver value through the addition of third-party content.

The support plan is for the Universal Base Image to follow the same life cycle and support dates as the underlying RHEL content. When run on a subscribed RHEL or OpenShift node, it follows the same support policy as the underlying RHEL content. Red Hat maintains a Universal Base Image for RHEL 7, which maps to RHEL 7 content, and another UBI for RHEL 8, which maps to RHEL 8 content.

Red Hat recommends using the Universal Base Image as the base container image for new applications. Red Hat commits to continue to support earlier RHEL base images for the duration of the support life cycle of the RHEL release.

The Red Hat Universal Base Image consists of:

- A set of three base images (**ubi**, **ubi-minimal**, and **ubi-init**). These mirror what is provided for building containers with RHEL 7 base images.
- A set of language runtime images (**java**, **php**, **python**, **ruby**, **nodejs**). These runtime images enable developers to start developing applications with the confidence that a Red Hat built and supported container image provides.
- A set of associated Yum repositories and channels that include RPM packages and updates. These allow you to add application dependencies and rebuild container images as needed.

Types of Universal Base Images

The Red Hat Universal Base Image provides three main base images:

ubi

A standard base image built on enterprise-grade packages from RHEL. Good for most application use cases.

ubi-minimal

A minimal base image built using **microdnf**, a scaled down version of the **dnf** utility. This provides the smallest container image.

ubi-init

This image allows you to easily run multiple services, such as web servers, application servers, and databases, all in a single container. It allows you to use the knowledge built into systemd unit files without having to determine how to start the service.

Advantages of the Red Hat Universal Base Image

Using the Red Hat Universal Base Image as the base image for your containerized applications has several advantages:

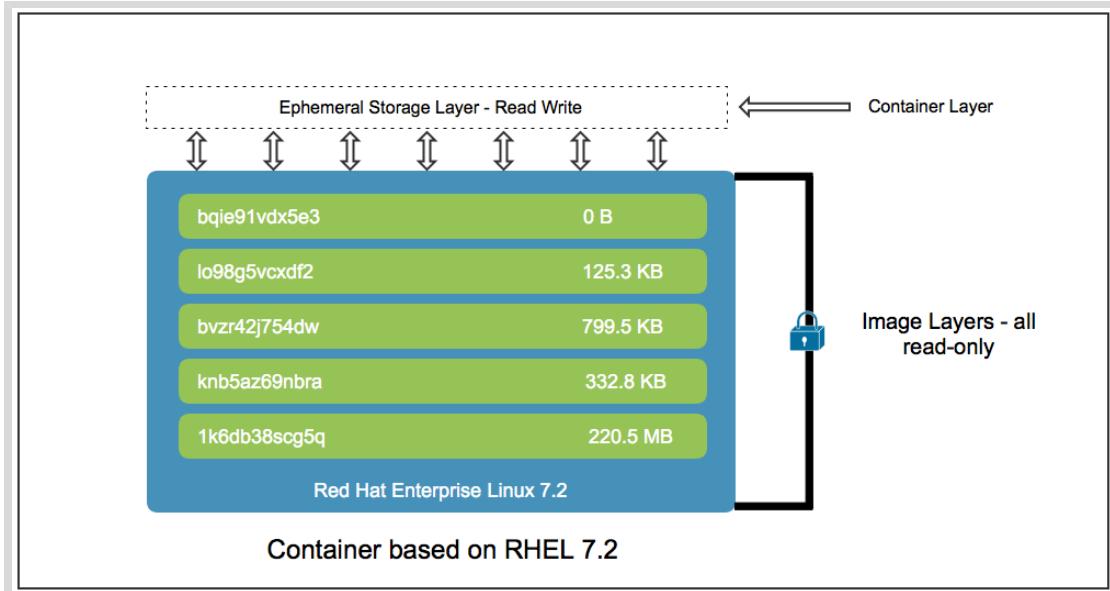
- **Minimal size:** The Universal Base Image is a relatively minimal (approximately 90-200 MB) base container image with fast startup times.
- **Security:** Provenance is a huge concern when using container base images. You must use a trusted image, from a trusted source. Language runtimes, web servers, and core libraries such as OpenSSL have an impact on security when moved into production. The Universal Base Image receives timely security updates from Red Hat security teams.
- **Performance:** The base images are tested, tuned, and certified by a Red Hat internal performance engineering team. These are proven container images used extensively in some of the world's most compute-intensive, I/O intensive, and fault sensitive workloads.
- **ISV, vendor certification, and partner support:** The Universal Base Image inherits the broad ecosystem of RHEL partners, ISVs, and third-party vendors supporting thousands of applications. The Universal Base Image makes it easy for these partners to build, deploy, and certify their applications and allows them to deploy the resulting containerized application on both Red Hat platforms such as RHEL and OpenShift, as well as non-Red Hat container platforms.
- **Build once, deploy onto many different hosts:** The Red Hat Universal Base Image can be built and deployed anywhere: on OpenShift/RHEL or any other container host (Fedora, Debian, Ubuntu, and more).

Advanced Dockerfile Instructions

A *Dockerfile* automates the building of container images. A Dockerfile is a text file that contains a set of instructions about how to build the container image. The instructions are executed one by one in sequential order. This section reviews some of the basic Dockerfile instructions and then discusses some more advanced instructions, including practical ways to use them when building container images for deployment in OpenShift.

The RUN Instruction

The **RUN** instruction executes commands in a new layer on top of the current image and then commits the results. The container build process then uses the committed result in the next step in the Dockerfile. The container build process uses **/bin/sh** to execute commands.

**Figure 2.1: Layers in a container image**

Each instruction in a Dockerfile results in a new layer for the final container image. Therefore, having too many instructions in a Dockerfile creates too many layers, resulting in images that are unnecessarily large. 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
RUN yum clean all -y
```

The previous example is not a good practice when creating container images, because it creates four layers for a single purpose. When creating Dockerfiles, Red Hat recommends that you always minimize the number of layers. You can achieve the same objective using the **&&** command separator to execute multiple commands within a single **RUN** instruction:

```
RUN yum --disablerepo=* --enablerepo="rhel-7-server-rpms" && \
    yum update && \
    yum install -y httpd && \
    yum clean all -y
```

The updated example creates only one layer and the readability is not compromised.



Note

In Docker 1.10 and later, only the **RUN**, **COPY**, and **ADD** instructions create layers. Other instructions create temporary intermediate images and do not directly increase the size of the image.

The **LABEL** Instruction

The **LABEL** instruction defines image metadata. Labels are key-value pairs attached to an image. The **LABEL** instruction typically adds descriptive metadata to the image, such as versions, a short description, and other details that provide information to users of the image.

When building images for OpenShift, prefix the label name with **io.openshift** to distinguish between OpenShift and Kubernetes related metadata. The OpenShift tooling can parse specific labels and perform specific actions based on the presence of these labels. The table below lists some of the most commonly used tags:

OpenShift Supported Labels

Label Name	Description
io.openshift.tags	This label contains a list of comma-separated tags. Tags categorize container images into broad areas of functionality. Tags help UI and generation tools to suggest relevant container images during the application creation process.
io.k8s.description	This label provides consumers of the container image more detailed information about the service or functionality that the image provides.
io.openshift.expose-services	This label contains a list of service ports that match the EXPOSE instructions in the Dockerfile and provides more descriptive information about what actual service is provided to consumers. The format is PORT[/PROTO]:NAME where [/PROTO] is optional and it defaults to tcp if not specified.

For a full list of all the OpenShift-specific labels, their descriptions, and example usage, refer to the references at the end of this section.

The WORKDIR Instruction

The **WORKDIR** instruction sets the working directory for any following **RUN**, **CMD**, **ENTRYPOINT**, **COPY**, or **ADD** instructions in a Dockerfile.

Red Hat recommends using absolute paths in **WORKDIR** instructions. Use **WORKDIR** instead of multiple **RUN** instructions where you change directories and then run some commands. This approach ensures better maintainability in the long run and is easier to troubleshoot.

The ENV Instruction

The **ENV** instruction defines 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.

It is good practice to use **ENV** instructions to define file and folder paths instead of hard-coding them in the Dockerfile instructions. It is useful to store information such as software version numbers, and also to append directories to the **PATH** environment variable.

The 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 for all labels and environment variables, and separate each key-value pair with an equal sign (=):

```
LABEL version="2.0" \
      description="This is an example container image" \
      creationDate="01-09-2017"
```

```
ENV MYSQL_DATABASE_USER="my_database_user" \
    MYSQL_DATABASE="my_database"
```

The USER Instruction

Red Hat recommends that you run the image as a non-root user for security reasons. To reduce the number of layers, avoid using the **USER** instruction too many times in a Dockerfile. The security implications that are specific to running containers as a non-root user are discussed later in this section.



Warning

OpenShift, by default, does not honor the **USER** instruction set by the container image. For security reasons, OpenShift uses a random userid other than the root userid (0) to run containers.

The VOLUME Instruction

The **VOLUME** instruction creates a mount point inside the container and indicates to image consumers that externally mounted volumes from the host machine or other containers can bind to this mount point.

Red Hat recommends using **VOLUME** instructions for persistent data. OpenShift can mount network-attached storage to the node running the container, and if the container moves to a new node then the storage is reattached to that node. By using the volume for all persistent storage needs, you preserve the content even if the container is restarted or moved.

Furthermore, explicitly defining volumes in your Dockerfile makes it easy for image consumers to understand what volumes they can define when running your image.

Building Images with the ONBUILD Instruction

The **ONBUILD** instruction registers *triggers* in the container image. A Dockerfile uses **ONBUILD** to declare instructions that are executed only when building a child image.

The **ONBUILD** instruction is useful to support easy customization of a container image for common use cases, such as preloading data or providing custom configuration to an application. The parent image provides commands that are common to all downstream child images. The child image only provides the data and configuration files. The Dockerfile for the child image could be as simple as having just the **FROM** instruction that references the parent image.



Note

The **ONBUILD** instruction is not included in the OCI specification, and therefore is not supported by default when you build containers with Podman or Buildah.

Use the **--format docker** option in order to enable support for the **ONBUILD** instruction.

For example, assume you are building a Node.js parent image that you want all developers in your organization to use as a base when they build applications with the following requirements:

- Enforce certain standards, such as copying the JavaScript sources to the application folder, so that they are interpreted by the Node.js engine.

- Execute the **npm install** command, which fetches all dependencies described in the **package.json** file.

You cannot embed these requirements as instructions in the parent Dockerfile because you do not have the application source code, and each application may have different dependencies listed in their **package.json** file.

Declare **ONBUILD** instructions in the parent Dockerfile. The parent Dockerfile is shown below:

```
FROM registry.access.redhat.com/rhscl/nodejs-6-rhel7
EXPOSE 3000
# Mandate that all Node.js apps use /usr/src/app as the main folder (APP_ROOT).
RUN mkdir -p /opt/app-root/
WORKDIR /opt/app-root

# Copy the package.json to APP_ROOT
ONBUILD COPY package.json /opt/app-root

# Install the dependencies
ONBUILD RUN npm install

# Copy the app source code to APP_ROOT
ONBUILD COPY src /opt/app-root

# Start node server on port 3000
CMD [ "npm", "start" ]
```

Assuming you built the parent container image and called it **mynodejs-base**, a child Dockerfile for an application that uses the parent image appears as follows:

```
FROM mynodejs-base
RUN echo "Started Node.js server..."
```

When the build process for the child image starts, it triggers the execution of the three **ONBUILD** instructions defined in the parent image, before invoking the **RUN** instruction in the child Dockerfile.

OpenShift Considerations for the USER Instruction

By default, OpenShift runs containers using an arbitrarily assigned userid. This approach mitigates the risk of processes running in the container getting escalated privileges on the host machine due to security vulnerabilities in the container engine.

Adapting Dockerfiles for OpenShift

When you write or change a Dockerfile that builds an image to run on an OpenShift cluster, you need to address the following:

- Directories and files that are read from or written to by processes in the container should be owned by the **root** group and have group read or group write permission.
- Files that are executable should have group execute permissions.
- The processes running in the container must not listen on privileged ports (that is, ports below 1024), because they are not running as privileged users.

Adding the following RUN instruction to your Dockerfile sets the directory and file permissions to allow users in the **root** group to access them in the container:

```
RUN chgrp -R 0 directory && \
    chmod -R g=u directory
```

The user account that runs the container is always a member of the **root** group, hence the container can read and write to this directory. The **root** group does not have any special permissions (unlike the **root** user) which minimizes security risks with this configuration.

The **g=u** argument from the **chmod** command makes the group permissions equal to the owner user permissions, which by default are read and write. You can use the **g+rwx** argument with the same results.

Running Containers as root Using Security Context Constraints (SCC)

In certain cases, you may not have access to Dockerfiles for certain images. You may need to run the image as the **root** user. In this scenario, you need to configure OpenShift to allow containers to run as **root**.

OpenShift provides *Security Context Constraints* (SCCs), which control the actions that a pod can perform and which resources it can access. OpenShift ships with a number of built-in SCCs. All containers created by OpenShift use the SCC named **restricted** by default, which ignores the userid set by the container image, and assigns a random userid to containers.

To allow containers to use a fixed userid, such as 0 (the **root** user), you need to use the *anyuid* SCC. To do so, you first need to create a *service account*. A service account is the OpenShift identity for a pod. All pods from a project run under a default service account, unless the pod, or its deployment configuration, is configured otherwise.

If you have an application that requires a capability not granted by the restricted SCC, then create a new, specific service account, add it to the appropriate SCC, and change the deployment configuration that creates the application pods to use the new service account.

The following steps detail how to allow containers to run as the **root** user in an OpenShift project:

- Create a new service account:

```
[user@host ~]$ oc create serviceaccount myserviceaccount
```

- Modify the deployment configuration for the application to use the new service account. Use the **oc patch** command to do this:

```
[user@host ~]$ oc patch dc/demo-app --patch \
> '{"spec": {"template": {"spec": {"serviceAccountName": "myserviceaccount"}}}}'
```



Note

For details on how to use the **oc patch** command, see OpenShift admins guide to **oc patch** [<https://access.redhat.com/articles/3319751>]. Run the **oc patch -h** command to display usage.

- Add the **myserviceaccount** service account to the **anyuid** SCC to run using a fixed userid in the container:

```
[user@host ~]$ oc adm policy add-scc-to-user anyuid -z myserviceaccount
```



References

Best practices for writing Dockerfiles

https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices

Further information about creating images is available in the *Creating Images* chapter of the *Images* guide of the OpenShift Container Platform product documentation at

https://access.redhat.com/documentation/en-us/openshift_container_platform/4.5/html/images/creating-images

► Guided Exercise

Building Container Images with Advanced Dockerfile Instructions

In this exercise, you will use Red Hat OpenShift to build and deploy an Apache HTTP Server container from a Dockerfile.

Outcomes

You should be able to:

- Create an Apache HTTP Server container image using a Dockerfile and deploy it to an OpenShift cluster.
- Create a child container image by extending the parent Apache HTTP Server image.
- Change the Dockerfile for the child container image so that it runs on an OpenShift cluster with a random user id.

Before You Begin

To perform this exercise, you need access to:

- A running OpenShift cluster.
- The parent image for the Apache HTTP Server ([quay.io/redhattraining/httpd-parent](https://quay.io/repository/redhattraining/httpd-parent)).
- The Dockerfile for the child container image in the Git repository ([container-build](#)).

Run the following command on the **workstation** VM to validate the prerequisites and to download the solution files:

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

► 1. Review the Apache HTTP Server parent Dockerfile.

A prebuilt Apache HTTP Server parent container image is provided in the Quay.io public registry at [quay.io/redhattraining/httpd-parent](https://quay.io/repository/redhattraining/httpd-parent). Briefly review the Dockerfile for this parent image located at [~/D0288/labs/container-build/httpd-parent/Dockerfile](#):

```
FROM registry.access.redhat.com/ubi8/ubi:8.0 ①

MAINTAINER Red Hat Training <training@redhat.com>

# DocumentRoot for Apache
ENV DOCROOT=/var/www/html ②

RUN yum install -y --no-docs --disableplugin=subscription-manager httpd && \
    yum clean all --disableplugin=subscription-manager -y && \ ③
```

```

echo "Hello from the httpd-parent container!" > ${DOCROOT}/index.html

# Allows child images to inject their own content into DocumentRoot
ONBUILD COPY src/ ${DOCROOT}/ ④

EXPOSE 80

# This stuff is needed to ensure a clean start
RUN rm -rf /run/httpd && mkdir /run/httpd

# Run as the root user
USER root ⑤

# Launch httpd
CMD /usr/sbin/httpd -DFOREGROUND

```

- ①** The base image is the Universal Base Image (UBI) for Red Hat Enterprise Linux 8.0 from the Red Hat Container Catalog.
- ②** Environment variables for this container image.
- ③** The **RUN** instruction contains several commands that install the Apache HTTP Server and create a default home page for the web server.
- ④** The **ONBUILD** instruction allows child images to provide their own customized web server content when building an image that extends from this parent image.
- ⑤** The **USER** instruction runs the Apache HTTP Server process as the **root** user.



Note

Notice how the **RUN** lines combine several commands into a single instruction wherever possible to reduce the number of layers in the image. This results in smaller images, which are faster to deploy.

▶ 2. Review the Apache HTTP Server child Dockerfile.

Use the parent Apache HTTP Server container image (**redhattraining/httpd-parent**) as a base to extend and customize the image to suit your application. The Dockerfile for the child container image is stored in the classroom Git repository server. To view the Dockerfile, perform the following steps:

2.1. Load your classroom environment configuration.

Run the following command to load the environment variables created in the first guided exercise:

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

2.2. Enter your local clone of the **D0288-apps** Git repository and checkout the **master** branch of the course's repository to ensure you start this exercise from a known good state:

```
[student@workstation ~]$ cd D0288-apps
[student@workstation D0288-apps]$ git checkout master
...output omitted...
```

2.3. Create a new branch where you can save any changes you make during this exercise:

```
[student@workstation D0288-apps]$ git checkout -b container-build
Switched to a new branch 'container-build'
[student@workstation D0288-apps]$ git push -u origin container-build
...output omitted...
* [new branch]      container-build -> container-build
Branch container-build set up to track remote branch container-build from origin.
```

- 2.4. Inspect the `~/D0288-apps/container-build/Dockerfile` file. The Dockerfile has a single instruction, `FROM`, which uses the `redhattraining/httpd-parent` image:

```
FROM quay.io/redhattraining/http-parent
```

- 2.5. The child container provides its own `index.html` file in the `~/D0288-apps/container-build/src` folder, which overwrites the parent's `index.html` file. The contents of the child container image's `index.html` file is shown below:

```
<!DOCTYPE html>
<html>
<body>
  Hello from the Apache child container!
</body>
</html>
```

- 3. Build and deploy a container to an OpenShift cluster using the Apache HTTP Server child Dockerfile.

- 3.1. Log in to OpenShift using your developer username:

```
[student@workstation D0288-apps]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful.
...output omitted...
```

- 3.2. Create a new project for the application. Prefix the project name with your developer username.

```
[student@workstation D0288-apps]$ oc new-project \
> ${RHT_OCP4_DEV_USER}-container-build
Now using project "youruser-container-build" on server "https://
api.cluster.domain.example.com:6443".
```

- 3.3. Build and deploy the Apache HTTP Server child image:

```
[student@workstation D0288-apps]$ oc new-app --as-deployment-config --name hola \
> https://github.com/${RHT_OCP4_GITHUB_USER}/D0288-apps#container-build \
> --context-dir container-build
--> Found Docker image 1d6f8d3 (11 minutes old) from quay.io for "quay.io/
redhattraining/httpd-parent"
```

Red Hat Universal Base Image 8

The Universal Base Image is designed and engineered to be the base layer for all of your containerized applications, middleware and utilities. This base image is freely redistributable, but Red Hat only supports Red Hat technologies through subscriptions for Red Hat products. This image is maintained by Red Hat and updated regularly.

```
Tags: base rhel8
```

```
...output omitted...
--> Success
Build scheduled, use 'oc logs -f bc/hola' to track its progress.
Application is not exposed. You can expose services to the outside world by
executing one or more of the commands below:
'oc expose svc/hola'
Run 'oc status' to view your app.
...output omitted...
```

▶ 4. View the build logs.



Important

The build process may take some time to start. If you see the following output, run the command again.

```
Error from server (BadRequest): unable to wait for build hola-1 to run: timed
out waiting for the condition
```

```
[student@workstation D0288-apps]$ oc logs -f bc/hola
Cloning "https://github.com/youruser/D0288-apps" ... ①
Commit: 823cd7a7476b5664cb267e5d0ac611de35df9f07 (Initial commit)
Author: Your Name <youremail@example.com>
Date: Sun Jun 9 20:45:53 2019 -0400
Replaced Dockerfile FROM image quay.io/redhattraining/httpd-parent
Caching blobs under "/var/cache/blobs".

Pulling image quay.io/redhattraining/httpd-parent@sha256:3e454fdac5 ②
...output omitted...
Getting image source signatures
Copying blob sha256:d02c3bd
...output omitted...
Writing manifest to image destination
Storing signatures
STEP 1: FROM quay.io/redhattraining/httpd-parent@sha256:2833...86ff
```

```
STEP 2: COPY src/ ${DOCROOT}/③
...output omitted...
Successfully pushed //image-registry.openshift-image-registry.svc:5000/... ④
Push successful
```

- ① OpenShift clones the Git repository from the URL provided by the `oc new-app` command.
 - ② The Dockerfile at the root of the Git repository is automatically identified and a Docker build process is kicked off.
 - ③ The `ONBUILD` instruction in the parent Dockerfile triggers the copying of the child's `index.html` file, which overwrites the parent index file.
 - ④ Finally, the built image is pushed to the OpenShift internal registry.
- ▶ 5. Verify that the application pod fails to start. The pod will be in the `Error` state, but if you wait too long, the pod will move to the `CrashLoopBackOff` state.

```
[student@workstation D0288-apps]$ oc get pods
NAME        READY   STATUS    RESTARTS   AGE
hola-1-build 0/1     Completed  0          22s
hola-1-deploy 1/1     Running   0          14s
hola-1-p75f5  0/1     Error     0          12s
```

- ▶ 6. Inspect the container's logs to see why the pod failed to start:

```
[student@workstation D0288-apps]$ oc logs hola-1-p75f5
AH00558: httpd: Could not reliably determine the server's fully qualified domain
name...
(13)Permission denied: AH00072: make_sock: could not bind to address [::]:80 ①
(13)Permission denied: AH00072: make_sock: could not bind to address 0.0.0.0:80 ②
no listening sockets available, shutting down
AH00015: Unable to open logs ③
```

- ① ② Because OpenShift runs containers using a random userid, ports below 1024 are privileged and can only be run as `root`.
- ③ The random userid used by OpenShift to run the container does not have permissions to read and write log files in `/var/log/httpd` (the default log file location for the Apache HTTP Server on RHEL 7).



Warning

The failed application pod is deleted after a short while. Make sure you inspect the application pod log files before the pod is deleted.

- ▶ 7. Delete all resources from the OpenShift project. The next step changes the Dockerfile to follow Red Hat recommendations for OpenShift.

Before updating the child Apache HTTP Server Dockerfile, delete all the resources in the project that have been created so far:

```
[student@workstation D0288-apps]$ oc delete all -l app=hola
pod "hola-2-gbx6f" deleted
replicationcontroller "hola-1" deleted
service "hola" deleted
```

```
deploymentconfig.apps.openshift.io "hola" deleted
buildconfig.build.openshift.io "hola" deleted
build.build.openshift.io "hola-1" deleted
imagestream.image.openshift.io "httpd-parent" deleted
imagestream.image.openshift.io "hola" deleted
```

- 8. Change the Dockerfile for the child container to run on an OpenShift cluster by updating the Apache HTTP Server process to run as a random, unprivileged user.
- 8.1. Edit the **~/D0288-apps/container-build/Dockerfile** file and perform the following steps. You can also copy these instructions from the provided **~/D0288-solutions/container-build/Dockerfile** file.
 - 8.2. Override the **EXPOSE** instruction from the parent image and change the port to 8080.

EXPOSE 8080

- 8.3. Include the **io.openshift.expose-service** label to indicate the changed port that the web server runs on:

LABEL io.openshift.expose-services="8080:http"

Update the list of labels to include the **io.k8s.description**, **io.k8s.display-name**, and **io.openshift.tags** labels that OpenShift consumes to provide helpful metadata about the container image:

```
LABEL io.k8s.description="A basic Apache HTTP Server child image, uses ONBUILD" \
      io.k8s.display-name="Apache HTTP Server" \
      io.openshift.expose-services="8080:http" \
      io.openshift.tags="apache, httpd"
```

- 8.4. You need to run the web server on an unprivileged port (that is, greater than 1024). Use a **RUN** instruction to change the port number in the Apache HTTP Server configuration file from the default port 80 to 8080:

RUN sed -i "s/Listen 80/Listen 8080/g" /etc/httpd/conf/httpd.conf

- 8.5. Change the group ID and permissions of the folders where the web server process reads and writes files:

```
RUN chgrp -R 0 /var/log/httpd /var/run/httpd && \
      chmod -R g=u /var/log/httpd /var/run/httpd
```

- 8.6. Add a **USER** instruction for an unprivileged user. The Red Hat convention is to use userid 1001:

USER 1001

- 8.7. Save the Dockerfile and commit the changes to the Git repository from the **~/D0288-apps/container-build** folder:

```
[student@workstation D0288-apps]$ cd container-build
[student@workstation container-build]$ git commit -a -m \
> "Changed Dockerfile to enable running as a random uid on OpenShift"
...output omitted...
[student@workstation container-build]$ git push
...output omitted...
[student@workstation container-build]$ cd ~
```

▶ 9. Rebuild and redeploy the Apache HTTP Server child container image.

- 9.1. Re-create the application using the new Dockerfile:

```
[student@workstation ~]$ oc new-app --as-deployment-config --name hola \
> https://github.com/${RHT_OCP4_GITHUB_USER}/D0288-apps#container-build \
> --context-dir container-build
```

- 9.2. Wait until the build finishes and the pod is ready and running. View the status of the application pod:

```
[student@workstation ~]$ oc get pods
NAME        READY   STATUS    RESTARTS   AGE
hola-1-75gkw 1/1     Running   0          5s
hola-1-build  0/1     Completed  0          15s
```

The application pod will now start successfully and be in a **Running** state.

▶ 10. Create an OpenShift route to expose the application to external access:

```
[student@workstation ~]$ oc expose svc/hola
route.route.openshift.io/hola exposed
```

▶ 11. Obtain the route URL using the **oc get route** command:

```
[student@workstation ~]$ oc get route
NAME      HOST/PORT                               PATH  SERVICES
PORT      TERMINATION    WILDCARD
hola      hola-youruser-container-build.cluster.domain.example.com   hola
8080-tcp           None
```

▶ 12. Test the application using the route URL you obtained in the previous step:

```
[student@workstation ~]$ curl \
> http://hola-${RHT_OCP4_DEV_USER}-container-build.${RHT_OCP4_WILDCARD_DOMAIN}
...output omitted...
Hello from the Apache child container!
...output omitted...
```

▶ 13. Clean up. Delete the project:

```
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-container-build
```

Finish

On **workstation**, run the **lab container-build finish** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

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

This concludes the guided exercise.

Injecting Configuration Data into an Application

Objectives

After completing this section, you should be able to select a method for injecting configuration data into an application and create the necessary resources to do so.

Externalizing Application Configuration in OpenShift

Typically, developers configure their applications through a combination of environment variables, command-line arguments, and configuration files. When deploying applications to OpenShift, configuration management presents a challenge due to the immutable nature of containers. Unlike traditional, non-containerized deployments, it is not recommended to couple the application with the configuration when running containerized applications.

The recommended approach for containerized applications is to decouple the static application binaries from the dynamic configuration data and to externalize the configuration. This separation ensures the portability of applications across many environments.

For example, suppose you want to promote an application that is deployed to an OpenShift cluster from a development environment to a production environment, with intermediate stages such as testing and user acceptance. You should use the same application container image in all stages and have the configuration details specific to each environment outside the container image.

Using Secret and Configuration Map Resources

OpenShift provides the secret and configuration map resource types to externalize and manage configuration for applications.

Secret resources are used to store sensitive information, such as passwords, keys, and tokens. As a developer, it is important to create secrets to avoid compromising credentials and other sensitive information in your application. There are different secret types which can be used to enforce usernames and keys in the secret object: **service-account-token**, **basic-auth**, **ssh-auth**, **tls** and **opaque**. The default type is **opaque**. The **opaque** type does not perform any validation, and allows unstructured key:value pairs that can contain arbitrary values.

Configuration map resources are similar to secret resources, but they store nonsensitive data. A configuration map resource can be used to store fine-grained information, such as individual properties, or coarse-grained information, such as entire configuration files and JSON data.

You can create configuration map and secret resources using the OpenShift CLI or the web console. You can then reference them in your pod specification and OpenShift automatically injects the resource data into the container as environment variables, or as files mounted through volumes inside the application container.

You can also change the deployment configuration for a running application to reference configuration map and secret resources. OpenShift then automatically redeploys the application and makes the data available to the container.

Data is stored inside a secret resource using base64 encoding. When data from a secret is injected into a container, the data is decoded and either mounted as a file, or injected as environment variables inside the container.

Features of Secrets and Configuration Maps

Notice the following with respect to secrets and configuration maps:

- They can be referenced independently of their definition.
- For security reasons, mounted volumes for these resources are backed by temporary file storage facilities (tmpfs) and never stored on a node.
- They are scoped to a namespace.

Creating and Managing Secrets and Configuration Maps

Secrets and configuration maps must be created before creating the pods that depend on them. You can use the CLI or the Web Console to create these resources.

Using the Command Line

Use the **oc create** command to create secrets and configuration map resources.

To create a new configuration map that stores string literals:

```
[user@host ~]$ oc create configmap config_map_name \
> --from-literal key1=value1 \
> --from-literal key2=value2
```

To create a new secret that stores string literals:

```
[user@host ~]$ oc create secret generic secret_name \
> --from-literal username=user1 \
> --from-literal password=mypa55w0rd
```

To create a new configuration map that stores the contents of a file or a directory containing a set of files:

```
[user@host ~]$ oc create configmap config_map_name \
> --from-file /home/demo/conf.txt
```

When you create a configuration map from a file, the key name will be the name of the file by default and the value will be the contents of the file.

When you create a configuration map resource based on a directory, each file whose name is a valid key in the directory is stored in the configuration map. Subdirectories, symbolic links, device files, and pipes are ignored.

Run the **oc create configmap --help** command for more information.

**Note**

You can also abbreviate the **configmap** resource type argument as **cm** in the **oc** command-line interface. For example:

```
[user@host ~]$ oc create cm myconf --from-literal key1=value1
[user@host ~]$ oc get cm myconf
```

To create a new secret that stores the contents of a file or a directory containing a set of files:

```
[user@host ~]$ oc create secret generic secret_name \
> --from-file /home/demo/mysecret.txt
```

When you create a secret from either a file or a directory, the key names are set the same way as for configuration maps.

For more details, including storing TLS certificates and keys in secrets, run the **oc create secret --help** and the **oc secret** commands.

Using the OpenShift Web Console

You can also use the OpenShift web console to create configuration maps and secrets. To create and manage secrets from the web console, log in to the OpenShift web console and navigate to the **Workloads → Secrets** page.

Name	Namespace	Type	Size	Created	⋮
builder-dockercfg-tbf7z	NS your-project	kubernetes.io/dockercfg	1	Aug 3, 12:05 am	⋮
builder-token-cs2r4	NS your-project	kubernetes.io/service-account-token	4	Aug 3, 12:05 am	⋮
builder-token-hxj2h	NS your-project	kubernetes.io/service-account-token	4	Aug 3, 12:05 am	⋮
default-dockercfg-97qm	NS your-project	kubernetes.io/dockercfg	1	Aug 3, 12:05 am	⋮
default-token-c892t	NS your-project	kubernetes.io/service-account-token	4	Aug 3, 12:05 am	⋮

Figure 2.2: Managing secrets from the web console

To create and manage configuration maps from the web console, navigate to the **Workloads → Config Maps** page.

Name	Namespace	Size	Created
CM your-project-1-ca	NS your-project	1	Aug 3, 3:18 pm
CM your-project-1-global-ca	NS your-project	1	Aug 3, 3:18 pm
CM your-project-1-sys-config	NS your-project	0	Aug 3, 3:18 pm

Figure 2.3: Managing configuration maps from the web console

You can edit the value assigned to each key in a configuration map, and also the encoded value assigned to each key in a secret, using the YAML editor provided by the web console. However, in the case of a secret, you need to encode your data in base64 format before inserting it into the secret resource definition.

Configuration Map and Secret Resource Definitions

Because configuration maps and secrets are regular OpenShift resources, you can use either the **oc create** command or the web console to import these resource definition files in YAML or JSON format.

A sample configuration map resource definition in YAML format is shown below:

```
apiVersion: v1
data:
  key1: ① value1 ②
  key2: ③ value2 ④
kind: ConfigMap ⑤
metadata:
  name: myconf ⑥
```

- ① The name of the first key. By default, an environment variable or a file with the same name as the key is injected into the container depending on whether the configuration map resource is injected as an environment variable or a file.
- ② The value stored for the first key of configuration map.
- ③ The name of the second key.
- ④ The value stored for the second key of the configuration map.
- ⑤ The OpenShift resource type; in this case, a configuration map.
- ⑥ A unique name for this configuration map inside a project.

A sample secret resource in YAML format is shown below:

```

apiVersion: v1
data:
  username: ❶ cm9vdAo= ❷
  password: ❸ c2VjcmV0Cg== ❹
kind: Secret ❺
metadata:
  name: mysecret ❻
  type: Opaque

```

- ❶ The name of the first key. This provides the default name for either an environment variable or a file in a pod, in the same way as key names from a configuration map.
- ❷ The value stored for the first key, in base64-encoded format.
- ❸ The name of the second key.
- ❹ The value stored for the second key, in base64-encoded format.
- ❺ The OpenShift resource type; in this case, a secret.
- ❻ A unique name for this secret resource inside a project.

Alternative Syntax for Secret Resource Definitions

A template cannot define secrets using the standard syntax, because all key values are encoded. OpenShift provides an alternative syntax for this scenario, where the **stringData** attribute replaces the **data** attribute, and the key values are not encoded.

Using the alternative syntax, the previous example becomes:

```

apiVersion: v1
stringData:
  username: user1
  password: pass1
kind: Secret
metadata:
  name: mysecret
  type: Opaque

```

The alternative syntax is never saved in the OpenShift master etcd database. OpenShift converts secret resources defined using the alternative syntax into the standard representation for storage. If you run **oc get** with a secret that was created using the alternative syntax, you get a resource using the standard syntax.

Commands to Manipulate Configuration Maps

To view the details of a configuration map in JSON format, or to export a configuration map resource definition to a JSON file for offline creation:

```
[user@host ~]$ oc get configmap/myconf -o json
```

To delete a configuration map:

```
[user@host ~]$ oc delete configmap/myconf
```

To edit a configuration map, use the **oc edit** command. This command opens a Vim-like buffer by default, with the configuration map resource definition in YAML format:

```
[user@host ~]$ oc edit configmap/myconf
```

Use the **oc patch** command to edit a configuration map resource. This approach is noninteractive and is useful when you need to script the changes to a resource:

```
[user@host ~]$ oc patch configmap/myconf --patch '{"data":{"key1":"newValue1"}}'
```

Commands to Manipulate Secrets

The commands to manipulate secret resources are similar to those used for configuration map resources.

To view or export the details of a secret:

```
[user@host ~]$ oc get secret/mysecret -o json
```

To delete a secret:

```
[user@host ~]$ oc delete secret/mysecret
```

To edit a secret, first encode your data in base64 format, for example:

```
[user@host ~]$ echo 'newpassword' | base64  
bmV3cGFzc3dvcmQK
```

Use the encoded value to update the secret resource using the **oc edit** command:

```
[user@host ~]$ oc edit secret/mysecret
```

You can also edit a secret resource using the **oc patch** command:

```
[user@host ~]$ oc patch secret/mysecret --patch \  
> '{"data":{"password":"bmV3cGFzc3dvcmQK"}}'
```

Configuration maps and secrets can also be changed and deleted using the OpenShift web console.

Injecting Data from Secrets and Configuration Maps into Applications

Configuration maps and secrets can be mounted as data volumes, or exposed as environment variables, inside an application container.

To inject all values stored in a configuration map into environment variables for pods created from a deployment configuration, use the **oc set env** command:

```
[user@host ~]$ oc set env dc/mydcname \  
> --from configmap/myconf
```

To mount all keys from a configuration map as files from a volume inside pods created from a deployment configuration, use the **oc set volume** command:

```
[user@host ~]$ oc set volume dc/mydcname --add \
> -t configmap -m /path/to/mount/volume \
> --name myvol --configmap-name myconf
```

To inject data inside a secret into pods created from a deployment configuration, use the **oc set env** command:

```
[user@host ~]$ oc set env dc/mydcname \
> --from secret/mysecret
```

To mount data from a secret resource as a volume inside pods created from a deployment configuration, use the **oc set volume** command:

```
[user@host ~]$ oc set volume dc/mydcname --add \
> -t secret -m /path/to/mount/volume \
> --name myvol --secret-name mysecret
```

Changing Configuration Maps and Secrets

Each time you change a deployment configuration, using commands such as **oc set env** and **oc set volume**, a new deployment is triggered by default.

If you make multiple changes to the same deployment configuration, it might be advisable to disable configuration change triggers using the **oc set triggers** command:

```
[user@host ~]$ oc set triggers dc/mydcname --from-config --remove
```

After making all of the changes to your config maps or secrets, use the **oc rollout latest** command to trigger a new deployment of the application:

```
[user@host ~]$ oc rollout latest mydcname
```

Use the **oc set triggers** command to re-enable the triggers:

```
[user@host ~]$ oc set triggers dc/mydcname --from-config
```

Application Configuration Options

Use configuration maps to store configuration data in plain text and if the information is not sensitive. Use secrets if the information you are storing is sensitive.

If your application only has a few simple configuration variables that can be read from environment variables or passed on the command line, use environment variables to inject data from configuration maps and secrets. Environment variables are the preferred approach over mounting volumes inside the container.

On the other hand, if your application has a large number of configuration variables, or if you are migrating a legacy application that makes extensive use of configuration files, use the volume

mount approach instead of creating an environment variable for each of the configuration variables. For example, if your application expects one or more configuration files from a specific location on your file system, you should create secrets or configuration maps from the configuration files and mount them inside the container ephemeral file system at the location that the application expects.

To accomplish that goal, with secrets pointing to **/home/student/configuration.properties** file, use the following command:

```
[user@host ~]$ oc create secret generic security \
> --from-file /home/student/configuration.properties
```

To inject the secret into the application, configure a volume that refers to the secrets created in the previous command. The volume must point to an actual directory inside the application where the secrets file is stored.

In the following example, the **configuration.properties** file is stored in the **/opt/app-root/secure** directory. To bind the file to the application, configure the deployment configuration from the application (**dc/application**):

```
[user@host ~]$ oc set volume dc/application --add \
> -t secret -m /opt/app-root/secure \
> --name myappsec-vol --secret-name security
```

To create a configuration map, use the following command:

```
[user@host ~]$ oc create configmap properties \
> --from-file /home/student/configuration.properties
```

To bind the application to the configuration map, update the deployment configuration from that application to use the configuration map:

```
[user@host ~]$ oc set env dc/application \
> --from configmap/properties
```



References

Further information about secrets is available in the *Providing Sensitive Data to Pods* chapter of the *Nodes* guide for Red Hat OpenShift Container Platform 4.5; at https://access.redhat.com/documentation/en-us/openshift_container_platform/4.5/html/nodes/working-with-pods#nodes-pods-secrets

► Guided Exercise

Injecting Configuration Data into an Application

In this exercise, you will use configuration maps and secrets to externalize the configuration for a containerized application.

Outcomes

You should be able to:

- Deploy a simple Node.js-based application that prints configuration details from environment variables and files.
- Inject configuration data into the container using configuration maps and secrets.
- Change the data in the configuration map and verify that the application picks up the changed values.

Before You Begin

To perform this exercise, ensure you have access to:

- A running OpenShift cluster.
- The S2I builder image for Node.js 12.
- The sample application in the Git repository ([app-config](#)).

Run the following command on the **workstation** VM to validate the exercise prerequisites and to download the lab and solution files:

```
[student@workstation ~]$ lab app-config start
```

► 1. Review the application source code.

- 1.1. Enter your local clone of the **DO288-apps** Git repository and check out the **master** branch of the course's repository to ensure you start this exercise from a known good state:

```
[student@workstation ~]$ cd DO288-apps
[student@workstation DO288-apps]$ git checkout master
...output omitted...
```

- 1.2. Create a new branch where you can save any changes you make during this exercise:

```
[student@workstation D0288-apps]$ git checkout -b app-config
Switched to a new branch 'app-config'
[student@workstation D0288-apps]$ git push -u origin app-config
...output omitted...
* [new branch]      app-config -> app-config
Branch app-config set up to track remote branch app-config from origin.
```

1.3. Inspect the `/home/student/D0288-apps/app-config/app.js` file.

The application reads the value of the `APP_MSG` environment variable and prints the contents of the `/opt/app-root/secure/myapp.sec` file:

```
// read in the APP_MSG env var
var msg = process.env.APP_MSG;
...output omitted...
// Read in the secret file
fs.readFile('/opt/app-root/secure/myapp.sec', 'utf8', function (secerr, secdata) {
...output omitted...
```

▶ 2. Build and deploy the application.

2.1. Load your classroom environment configuration.

Run the following command to load the environment variables created in the first guided exercise:

```
[student@workstation D0288-apps]$ source /usr/local/etc/ocp4.config
```

2.2. Log in to OpenShift using your developer user account:

```
[student@workstation D0288-apps]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful
...output omitted...
```

2.3. Create a new project for the application. Prefix the project name with your developer username:

```
[student@workstation D0288-apps]$ oc new-project ${RHT_OCP4_DEV_USER}-app-config
```

2.4. Create a new application called myapp from sources in Git. Use the branch you created in a previous step.

You can copy or execute the command from the `oc-new-app.sh` script in the `/home/student/D0288/labs/app-config` folder:

```
[student@workstation D0288-apps]$ oc new-app --as-deployment-config --name myapp \
> --build-env npm_config_registry=\
> http://${RHT_OCP4_NEXUS_SERVER}/repository/nodejs \
> nodejs:12-https://github.com/${RHT_OCP4_GITHUB_USER}/D0288-apps#app-config \
> --context-dir app-config
...output omitted...
--> Creating resources ...output omitted...
```

```
imagestream.image.openshift.io "myapp" created
buildconfig.build.openshift.io "myapp" created
deploymentconfig.apps.openshift.io "myapp" created
service "myapp" created
--> Success
...output omitted...
```

Notice there is no space before or after the equal sign (=) after **npm_config_registry**.

- 2.5. View the build logs. Wait until the build finishes and the application container image is pushed to the OpenShift internal registry:

```
[student@workstation D0288-apps]$ oc logs -f bc/myapp
Cloning "https://github.com/youruser/D0288-apps#app-config" ...
...output omitted...
---> Installing application source ...
---> Building your Node application from source
...output omitted...
Pushing image image-registry.openshift-image-registry.svc:5000/youruser-app-
config/myapp:latest ...
...output omitted...
Push successful
```

▶ 3. Test the application.

- 3.1. Wait until the application deploys. View the status of the application pod. The application pod should be in a **Running** state:

```
[student@workstation D0288-apps]$ oc get pods
NAME        READY   STATUS    RESTARTS   AGE
myapp-1-2w1nc  1/1     Running   0          81s
myapp-1-deploy  0/1     Completed  0          89s
myapp-1-build   0/1     Completed  0          3m
```

- 3.2. Use a route to expose the application to external access:

```
[student@workstation D0288-apps]$ oc expose svc myapp
route.route.openshift.io/myapp exposed
```

- 3.3. Identify the route URL where the application API is exposed:

```
[student@workstation D0288-apps]$ oc get route
NAME      HOST/PORT           ...
myapp     myapp-youruser-app-config.apps.cluster.domain.example.com ...
```

- 3.4. Invoke the route URL identified from the previous step using the **curl** command:

```
[student@workstation D0288-apps]$ curl \
> http://myapp-${RHT_OCP4_DEV_USER}-app-config.${RHT_OCP4_WILDCARD_DOMAIN}
Value in the APP_MSG env var is => undefined
Error: ENOENT: no such file or directory, open '/opt/app-root/secure/myapp.sec'
```

The **undefined** value for the environment variable, and the **ENOENT: no such file or directory** error is shown because there is no such environment variable or file that exists in the container.

▶ 4. Create the configuration map and secret resources.

- 4.1. Create a configuration map resource to hold configuration variables that store plain text data.

Create a new configuration map resource called **myappconf**. Store a key called **APP_MSG** with the value **Test Message** in this configuration map:

```
[student@workstation D0288-apps]$ oc create configmap myappconf \
> --from-literal APP_MSG="Test Message"
configmap/myappconf created
```

- 4.2. Verify that the configuration map contains the configuration data:

```
[student@workstation D0288-apps]$ oc describe cm/myappconf
Name: myappconf
...output omitted...
Data
=====
APP_MSG:
-----
Test Message
...output omitted...
```

- 4.3. Review the contents of the **/home/student/D0288-apps/app-config/myapp.sec** file:

```
username=user1
password=pass1
salt=xyz123
```

- 4.4. Create a new secret to store the contents of the **myapp.sec** file.

```
[student@workstation D0288-apps]$ oc create secret generic myappfilesec \
> --from-file /home/student/D0288-apps/app-config/myapp.sec
secret/myappfilesec created
```

- 4.5. Verify the contents of the secret. Note that the contents are stored in base64-encoded format:

```
[student@workstation D0288-apps]$ oc get secret/myappfilesec -o json
{
  "apiVersion": "v1",
  "data": {
    "myapp.sec": "dXNlcj5hbWU9dXNlcjEKcGFzc3dvcmQ9cGFzcxEKc2...
  },
  "kind": "Secret",
  "metadata": {
    ...output omitted...
```

```

    "name": "myappfilesec",
    ...output omitted...
},
"type": "Opaque"
}

```

- ▶ 5. Inject the configuration map and the secret into the application container.
- 5.1. Use the **oc set env** command to add the configuration map to the deployment configuration:

```
[student@workstation ~]$ oc set env dc/myapp \
> --from configmap/myappconf
deploymentconfig.apps.openshift.io/myapp updated
```

- 5.2. Use the **oc set volume** command to add the secret to the deployment configuration:

You can copy or execute the command from the **inject-secret-file.sh** script in the **/home/student/D0288/labs/app-config** folder:

```
[student@workstation D0288-apps]$ oc set volume dc/myapp --add \
> -t secret -m /opt/app-root/secure \
> --name myappsec-vol --secret-name myappfilesec
deploymentconfig.apps.openshift.io/myapp volume updated
```

- ▶ 6. Verify that the application is redeployed and uses the data from the configuration map and the secret.
- 6.1. Verify that the application is redeployed because of the changes made to the deployment configuration in previous steps:

```
[student@workstation D0288-apps]$ oc status
In project youruser-app-config on server ...output omitted...

http://myapp-youruser-app-config.apps.cluster.domain.example.com to pod port 8080-
tcp (svc/myapp)
dc/myapp deploys istag/myapp:latest <
  bc/myapp source builds https://github.com/youruser/D0288-apps#app-config on
openshift/nodejs:12
  deployment #3 deployed 24 seconds ago - 1 pod
  deployment #2 deployed 4 minutes ago
  deployment #1 deployed 18 minutes ago
```



Note

You should see the application redeployed twice, due to the two **oc set env** commands that change the deployment configuration.

You can also safely ignore errors messages similar to the following:

```
deployment #2 failed 59 seconds ago: newer deployment was found running
```

- 6.2. Wait until the application pod is ready and in a **Running** state. Get the name of the application pod using the **oc get pods** command,

```
[student@workstation D0288-apps]$ oc get pods
NAME        READY   STATUS    RESTARTS   AGE
myapp-1-build  0/1     Completed  0          13m
myapp-1-deploy 0/1     Completed  0          12m
myapp-3-deploy 0/1     Completed  0          3m
myapp-3-wzdbh  1/1     Running   0          2m
```

- 6.3. Use the **oc rsh** command to inspect the environment variables in the container:

```
[student@workstation D0288-apps]$ oc rsh myapp-3-wzdbh env | grep APP_MSG
APP_MSG=Test Message
```

- 6.4. Verify that the configuration map and secret were injected into the container. Retest the application using the route URL:

```
[student@workstation D0288-apps]$ curl \
> http://myapp-$${RHT_OCP4_DEV_USER}-app-config.$${RHT_OCP4_WILDCARD_DOMAIN}
Value in the APP_MSG env var is => Test Message
The secret is => username=user1
password=pass1
salt=xyz123
```

OpenShift injects the configuration map as an environment variable and mounts the secret as a file into the container. The application reads the environment variable and file and displays their data.

► 7. Change the information stored in the configuration map and retest the application.

- 7.1. Use the **oc edit configmap** command to change the value of the **APP_MSG** key:

```
[student@workstation D0288-apps]$ oc edit cm/myappconf
```

The above command opens a Vim-like buffer with the configuration map attributes in YAML format. Edit the value associated with the **APP_MSG** key under the data section and change the value as follows:

```
...output omitted...
apiVersion: v1
data:
  APP_MSG: Changed Test Message
kind: ConfigMap
...output omitted...
```

Save and close the file.

- 7.2. Verify that the value in the **APP_MSG** key is updated:

```
[student@workstation D0288-apps]$ oc describe cm/myappconf
Name:  myappconf
...output omitted...
```

```
Data
=====
APP_MSG:
-----
Changed Test Message
...output omitted...
```

- 7.3. Use the **oc rollout latest** command to trigger a new deployment. This ensures that the application picks up the changed values in the configuration map:

```
[student@workstation D0288-apps]$ oc rollout latest dc/myapp
deploymentconfig.apps.openshift.io/myapp rolled out
```

- 7.4. Wait for the application pod to redeploy and appear in a **Running** state:

```
[student@workstation D0288-apps]$ oc get pods
NAME        READY   STATUS    RESTARTS   AGE
...output omitted...
myapp-4-2feqr  1/1     Running      0          6s
```

- 7.5. Test the application and verify that the changed values in the configuration map display:

```
[student@workstation D0288-apps]$ curl \
> http://myapp-${RHT_OCP4_DEV_USER}-app-config.${RHT_OCP4_WILDCARD_DOMAIN}
Value in the APP_MSG env var is => Changed Test Message
The secret is => username=user1
password=pass1
salt=xyz123
```

- 8. Clean up. Delete the **youruser-app-config** project in OpenShift.

```
[student@workstation D0288-apps]$ oc delete project \
> ${RHT_OCP4_DEV_USER}-app-config
project.project.openshift.io "youruser-app-config" deleted
```

Finish

On **workstation**, run the **lab app-config finish** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises. The finish action releases this project and its resources.

```
[student@workstation D0288-apps]$ lab app-config finish
```

This concludes the guided exercise.

► Lab

Designing Containerized Applications for OpenShift

Performance Checklist

In this lab, you will fix the Dockerfile for an application based on the Thorntail framework to run on an OpenShift cluster. You will also use a configuration map to configure the application.



Note

The **grade** command at the end of each chapter lab requires that you use the exact project names and other identifiers as given in the lab specification.

Outcomes

You should be able to fix the Dockerfile for an application based on the Thorntail framework to run as a random user, and deploy the application to an OpenShift cluster. You should also be able to use a configuration map to store a simple text string that is used to configure the application.

Before You Begin

To perform this exercise, ensure you have access to:

- A running OpenShift cluster.
- The runnable fat JAR of the application.
- The application Git repository.

Run the following command on **workstation** to validate the prerequisites. The command also downloads helper files and solution files for the lab:

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

Requirements

The application is written in Java, using the Thorntail framework. The prebuilt, runnable JAR file (fat JAR) containing the application and the Thorntail runtime is provided. The application provides a simple REST API that responds to requests based on a configuration that is injected into the container as an environment variable. Build and deploy the application to an OpenShift cluster according to the following requirements:

- The application name for OpenShift is **elvis**. The configuration for the application should be stored in a configuration map called **appconfig**.
- Deploy the application to a project named **youruser-design-container**.
- The REST API for the application should be accessible at the URL:

`elvis-youruser-design-container.apps.cluster.domain.example.com/api/hello.`

- The Git repository and folder that contains the application sources is:

`https://github.com/youruser/D0288-apps/hello-java.`

- The prebuilt application JAR file is available at:

`https://github.com/RedHatTraining/D0288-apps/releases/download/OCP-4.1-1/hello-java.jar`

Steps

1. Navigate to your local clone of the **D0288-apps** Git repository and create a new branch named **design-container** from the **master** branch. Briefly review the Dockerfile for the application in the `/home/student/D0288-apps/hello-java/` directory.
2. Deploy the application in the **hello-java** folder of the **D0288-apps** Git repository, using the **design-container** branch of the application. Deploy the application to the **youruser-design-container** project in OpenShift without making any changes.
Do not forget to source the variables in the `/usr/local/etc/ocp4.config` file before logging in to the OpenShift cluster.
3. Follow the build logs and verify that the container image is built and pushed to the OpenShift internal registry. View the deployment status of the application pod. The pod will be in a **CrashLoopBackOff** or **Error** state. View the application logs to see why the application is not starting correctly.
4. Edit the Dockerfile for the application to ensure successful deployment to an OpenShift cluster. The container should run using a random user id rather than the currently configured **wildfly** user.
5. Commit the changes you made to the Dockerfile and push the changes to the classroom Git repository.
6. Start a new build of the application. Follow the build log for the new build. Verify that the application pod starts successfully.
7. Expose the service to external access and test the application. Access the application's API using the `/api/hello` context path.
8. Create a new configuration map called **appconfig**. Store a key called **APP_MSG** with the value **Elvis lives** in this configuration map. Add that key as an environment variable to the application's deployment configuration.
9. Verify that a new deployment is triggered and wait for a new application pod to be ready and running. Verify that the **APP_MSG** key is injected into the container as an environment variable.
10. Test the application by invoking its REST API URL (`http://elvis-youruser-design-container.apps.cluster.domain.example.com/api/hello`) and verify that the **APP_MSG** key value appears in the response.

Evaluation

As the **student** user on the **workstation** machine, use the **lab** command to grade your work. Correct any reported failures and rerun the command until successful.

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

Finish

On **workstation**, run the **lab design-container finish** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises. The finish action releases this project and its resources.

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

This concludes the lab.

► Solution

Designing Containerized Applications for OpenShift

Performance Checklist

In this lab, you will fix the Dockerfile for an application based on the Thorntail framework to run on an OpenShift cluster. You will also use a configuration map to configure the application.



Note

The **grade** command at the end of each chapter lab requires that you use the exact project names and other identifiers as given in the lab specification.

Outcomes

You should be able to fix the Dockerfile for an application based on the Thorntail framework to run as a random user, and deploy the application to an OpenShift cluster. You should also be able to use a configuration map to store a simple text string that is used to configure the application.

Before You Begin

To perform this exercise, ensure you have access to:

- A running OpenShift cluster.
- The runnable fat JAR of the application.
- The application Git repository.

Run the following command on **workstation** to validate the prerequisites. The command also downloads helper files and solution files for the lab:

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

Requirements

The application is written in Java, using the Thorntail framework. The prebuilt, runnable JAR file (fat JAR) containing the application and the Thorntail runtime is provided. The application provides a simple REST API that responds to requests based on a configuration that is injected into the container as an environment variable. Build and deploy the application to an OpenShift cluster according to the following requirements:

- The application name for OpenShift is **elvis**. The configuration for the application should be stored in a configuration map called **appconfig**.
- Deploy the application to a project named **youruser-design-container**.

- The REST API for the application should be accessible at the URL:

`elvis-youruser-design-container.apps.cluster.domain.example.com/api/hello.`

- The Git repository and folder that contains the application sources is:

`https://github.com/youruser/D0288-apps/hello-java.`

- The prebuilt application JAR file is available at:

`https://github.com/RedHatTraining/D0288-apps/releases/download/OCP-4.1-1/hello-java.jar`

Steps

1. Navigate to your local clone of the **D0288-apps** Git repository and create a new branch named **design-container** from the **master** branch. Briefly review the Dockerfile for the application in the **/home/student/D0288-apps/hello-java/** directory.

- 1.1. Check out the **master** branch of the Git repository.

```
[student@workstation ~]$ cd D0288-apps
[student@workstation D0288-apps]$ git checkout master
...output omitted...
```

- 1.2. Create a new branch where you can save any changes you make during this exercise:

```
[student@workstation D0288-apps]$ git checkout -b design-container
Switched to a new branch 'design-container'
[student@workstation D0288-apps]$ git push -u origin design-container
...output omitted...
* [new branch]      design-container -> design-container
Branch design-container set up to track remote branch design-container from
origin.
```

- 1.3. Inspect the **/home/student/D0288-apps/hello-java/Dockerfile** file. Do not make any changes to it for now.
2. Deploy the application in the **hello-java** folder of the **D0288-apps** Git repository, using the **design-container** branch of the application. Deploy the application to the **youruser-design-container** project in OpenShift without making any changes.

Do not forget to source the variables in the **/usr/local/etc/ocp4.config** file before logging in to the OpenShift cluster.

- 2.1. Load your classroom environment configuration.

Run the following command to load the configuration variables created in the first guided exercise:

```
[student@workstation D0288-apps]$ source /usr/local/etc/ocp4.config
```

- 2.2. Log in to OpenShift using your developer user account:

```
[student@workstation D0288-apps]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful
...output omitted...
```

- 2.3. Create a new project for the application. Prefix the project's name with your developer username:

```
[student@workstation D0288-apps]$ oc new-project \
> ${RHT_OCP4_DEV_USER}-design-container
```

- 2.4. Create a new application called elvis from the Dockerfile stored in the Git repository.

Copy or execute the command from the `oc-new-app.sh` script in the `/home/student/D0288/labs/design-container` directory.

```
[student@workstation D0288-apps]$ oc new-app --as-deployment-config --name elvis \
> https://github.com/${RHT_OCP4_GITHUB_USER}/D0288-apps#design-container \
> --context-dir hello-java
...output omitted...
--> Creating resources ...
imagestream.image.openshift.io "ubi" created
imagestream.image.openshift.io "elvis" created
buildconfig.build.openshift.io "elvis" created
deploymentconfig.apps.openshift.io "elvis" created
service "elvis" created
--> Success
...output omitted...
```

3. Follow the build logs and verify that the container image is built and pushed to the OpenShift internal registry. View the deployment status of the application pod. The pod will be in a **CrashLoopBackOff** or **Error** state. View the application logs to see why the application is not starting correctly.

- 3.1. Follow the build logs:

```
[student@workstation D0288-apps]$ oc logs -f bc/elvis
...output omitted...
STEP 1: FROM registry.access.redhat.com/ubi8/ubi@sha256...
...output omitted...
Pushing image image-registry.openshift-image-registry.svc:5000/youruser-design-
container/elvis:latest ...
...output omitted...
Push successful
```

- 3.2. Wait until the application pod is deployed. The application does not reach a **Ready** status. It will, after some time, stay in either a **CrashLoopBackOff** or **Error** status.

```
[student@workstation D0288-apps]$ oc get pods
NAME          READY   STATUS      RESTARTS   AGE
...output omitted...
elvis-1-tgv5s  0/1     CrashLoopBackOff   1          13s
```

- 3.3. View the logs for the application pod and investigate why the application pod failed to start.

```
[student@workstation D0288-apps]$ oc logs elvis-1-tgv5s  
/bin/sh: /opt/app-root/bin/run-app.sh: Permission denied
```

The application fails to start due to a "Permission denied" error in the file system, because OpenShift does not run the pod using the user specified in the Dockerfile.

4. Edit the Dockerfile for the application to ensure successful deployment to an OpenShift cluster. The container should run using a random user id rather than the currently configured **wildfly** user.
- 4.1. Edit the Dockerfile at **/home/student/D0288-apps/hello-java/Dockerfile** with a text editor. Make the changes outlined in the steps below. You can also copy the instructions and commands from the solution file provided at **/home/student/D0288/solutions/design-container/Dockerfile**.

Remove the **useradd** command in the first **RUN** instruction (line 12).

```
useradd wildfly && \
```

- 4.2. Locate the **chown** and **chmod** commands on lines 19 and 20:

```
RUN chown -R wildfly:wildfly /opt/app-root && \  
chmod -R 700 /opt/app-root
```

Replace them with the following:

```
RUN chgrp -R 0 /opt/app-root && \  
chmod -R g=u /opt/app-root
```

- 4.3. Replace the **wildfly** user in the **USER** instruction on line 24 with the generic userid of 1001 to avoid inheriting the user from the parent RHEL image. This generic userid is ignored by OpenShift and follows Red Hat recommendations and conventions for building images:

```
USER 1001
```

5. Commit the changes you made to the Dockerfile and push the changes to the classroom Git repository.

```
[student@workstation D0288-apps]$ cd hello-java  
[student@workstation hello-java]$ git commit -a -m \  
> "Fixed Dockerfile to run with random user id on OpenShift"  
[student@workstation hello-java]$ git push  
[student@workstation hello-java]$ cd ~  
[student@workstation ~]$
```

6. Start a new build of the application. Follow the build log for the new build. Verify that the application pod starts successfully.

- 6.1. Start a new build for the application:

```
[student@workstation ~]$ oc start-build elvis
build.build.openshift.io/elvis-2 started
```

- 6.2. Follow the build log and verify that a new container image is created and pushed to the OpenShift internal registry:

```
[student@workstation ~]$ oc logs -f bc/elvis
...output omitted...
Pushing image image-registry.openshift-image-registry.svc:5000/youruser-design-
container/elvis:latest ...
...output omitted...
Push successful
```

- 6.3. Wait for the application pod to be ready and running.

```
[student@workstation ~]$ oc get pods
NAME          READY   STATUS    RESTARTS   AGE
...output omitted...
elvis-2-9gvqr  1/1     Running   0          9s
```

- 6.4. View the logs for the application pod and verify that there are no errors during startup:

```
[student@workstation ~]$ oc logs elvis-2-9gvqr
Starting hello-java app...
JVM options => -Xmx512m
...output omitted...
2017-10-12 13:49:20,647 INFO  [org.jboss.as] (MSC service thread 1-3) WFLYSRV0049:
Thorntail 2.4.0.Final (WildFly Core 7.0.0.Final) starting
...output omitted...
2017-10-12 13:49:23,237 INFO  [org.wildfly.swarm] (main) THORN99999: Thorntail is
Ready
```

7. Expose the service to external access and test the application. Access the application's API using the **/api/hello** context path.

- 7.1. Expose the application for external access.

```
[student@workstation ~]$ oc expose svc/elvis
route.route.openshift.io/elvis exposed
```

- 7.2. Identify the host name where the application API is exposed:

```
[student@workstation ~]$ oc get route
NAME      HOST/PORT
elvis    elvis-youruser-design-container.apps.cluster.domain.example.com
```

- 7.3. Test the application by invoking the API URL using the host name identified from the previous step (<http://elvis-youruser-design-container.apps.cluster.domain.example.com/api/hello>), and verify that the application pod name appears in the response:

```
[student@workstation ~]$ curl \
> http://elvis-${RHT_OCP4_DEV_USER}-design-container.${RHT_OCP4_WILDCARD_DOMAIN}\
> /api/hello
Hello world from host elvis-2-9gvqr
```

8. Create a new configuration map called **appconfig**. Store a key called **APP_MSG** with the value **Elvis lives** in this configuration map. Add that key as an environment variable to the application's deployment configuration.

8.1. Create the configuration map:

```
[student@workstation ~]$ oc create cm appconfig \
> --from-literal APP_MSG="Elvis lives"
configmap/appconfig created
```

8.2. View the details of the configuration map:

```
[student@workstation ~]$ oc describe cm/appconfig
Name: appconfig
...output omitted...

Data
=====
APP_MSG:
-----
Elvis lives
```

8.3. Use the **oc set env** command to add the configuration map to the deployment configuration:

```
[student@workstation ~]$ oc set env dc/elvis --from cm/appconfig
deploymentconfig.apps.openshift.io/elvis updated
```

9. Verify that a new deployment is triggered and wait for a new application pod to be ready and running. Verify that the **APP_MSG** key is injected into the container as an environment variable.

9.1. Verify that a new deployment was triggered:

```
[student@workstation ~]$ oc status
...output omitted...
dc/elvis deploys istag/elvis:latest <-
  bc/elvis docker builds https://github.com/youruser/D0288-apps#design-container
  on istag/ubi:8.0
    deployment #3 deployed 3 minutes ago - 1 pod
    deployment #2 deployed 38 minutes ago
...output omitted...
```

9.2. Wait for the application pod to redeploy. Verify that the new application pod is ready and running:

```
[student@workstation ~]$ oc get pods
NAME        READY   STATUS    RESTARTS   AGE
...output omitted...
elvis-3-ks1np  1/1     Running   0          3m
```

9.3. Verify that the **APP_MSG** key is injected into the container as an environment variable:

```
[student@workstation ~]$ oc rsh elvis-3-ks1np env | grep APP_MSG
APP_MSG=Elvis lives
```

10. Test the application by invoking its REST API URL (`http://elvis-youruser-design-container.apps.cluster.domain.example.com/api/hello`) and verify that the **APP_MSG** key value appears in the response.

```
[student@workstation ~]$ curl \
> http://elvis-${RHT_OCP4_DEV_USER}-design-container.${RHT_OCP4_WILDCARD_DOMAIN}\
> /api/hello
Hello world from host [elvis-3-ks1np]. Message received = Elvis lives
```

Evaluation

As the **student** user on the **workstation** machine, use the **lab** command to grade your work. Correct any reported failures and rerun the command until successful.

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

Finish

On **workstation**, run the **lab design-container finish** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises. The finish action releases this project and its resources.

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

This concludes the lab.

Summary

In this chapter, you learned:

- RHOCP container deployment options include:
 - Deploy pre-built container images directly on an OpenShift cluster.
 - Create a Dockerfile using a base image and customize to suit your needs.
 - Use a Source-to-Image (S2I) build where RHOCP combines source code with a builder image.
- Common changes to Dockerfiles required to run a container on RHOCP:
 - Root group permissions on files that are read or written by processes in the container.
 - Files that are executable must have group execute permissions.
 - Processes running in the container must not listen on privileged ports (ports below 1024).
- Use Secrets to store sensitive information and access from your pods.
- Use configuration map resources to store nonsensitive environment specific data.

Chapter 3

Publishing Enterprise Container Images

Goal

Interact with an enterprise registry and publish container images to it.

Objectives

- Manage container images in registries using Linux container tools.
- Access the OpenShift internal registry using Linux container tools.
- Create image streams for container images in external registries.

Sections

- Managing Images in an Enterprise Registry (and Guided Exercise)
- Allowing Access to the OpenShift Registry (and Guided Exercise)
- Creating Image Streams (and Guided Exercise)

Lab

Publishing Enterprise Container Images

Managing Images in an Enterprise Registry

Objectives

After completing this section, you should be able to manage container images in registries using Linux container tools.

Reviewing Container Registries

A *container image registry*, *container registry*, or *registry server* stores the images that you deploy as containers and provides mechanisms to pull, push, update, search, and remove container images. It uses a standard REST API defined by the *Open Container Initiative (OCI)*, which is based on the Docker Registry HTTP API v2. From the perspective of an organization that runs an OpenShift cluster, there are many kinds of container registries:

Public registries

Registries that allow anyone to consume container images directly from the internet without any authentication. Docker Hub, Quay.io, and the Red Hat Registry are examples of public container registries.

Private registries

Registries that are available only to selected consumers and usually require authentication. The Red Hat terms-based registry is an example of a private container registry.

External registries

Registries that your organization does not control. They are usually managed by a cloud provider or a software vendor. Quay.io is an example of an external container registry.

Enterprise registries

Registry servers that your organization manages. They are usually available only to the organization's employees and contractors.

OpenShift internal registries

A registry server managed internally by an OpenShift cluster to store container images. Create those images using OpenShift's build configurations and the S2I process or a Dockerfile, or import them from other registries.

These kinds of registries are not mutually exclusive: a registry can be, at the same time, both public and private registry. Usually a public registry is also an external registry, because your organization can access it over the internet, without authentication, and your organization does not control it. The same registry could also be a private registry, if your organization has a plan with the registry provider that allows you to host private images and your organization also has control over who else can access those private images.

Quay.io works as both a public and a private registry for some users. The same developer can use some public container images from Quay.io, and also some container images from a vendor, that requires authentication.

Red Hat-Managed Registries

Red Hat manages a set of public and private container registries to serve different kinds of container images to distinct audiences. Container images that are supported with production-

Chapter 3 | Publishing Enterprise Container Images

level SLAs by either Red Hat or its partners are accessible through the *Red Hat Container Catalog*. Community and unsupported container images are accessible through *Quay.io*.

The Red Hat Container Catalog at <https://access.redhat.com/containers> is a web user interface allowing you to browse and search those registries and to get detailed information on images based on Red Hat Enterprise Linux.

Images provided by Red Hat benefit from the long experience Red Hat has in managing security vulnerabilities and defects in Red Hat Enterprise Linux and other products. The Red Hat security team hardens and controls these high-quality images, and then signs these images to prevent tampering. Red Hat also rebuilds these images every time new vulnerabilities are discovered and executes a quality assurance process.

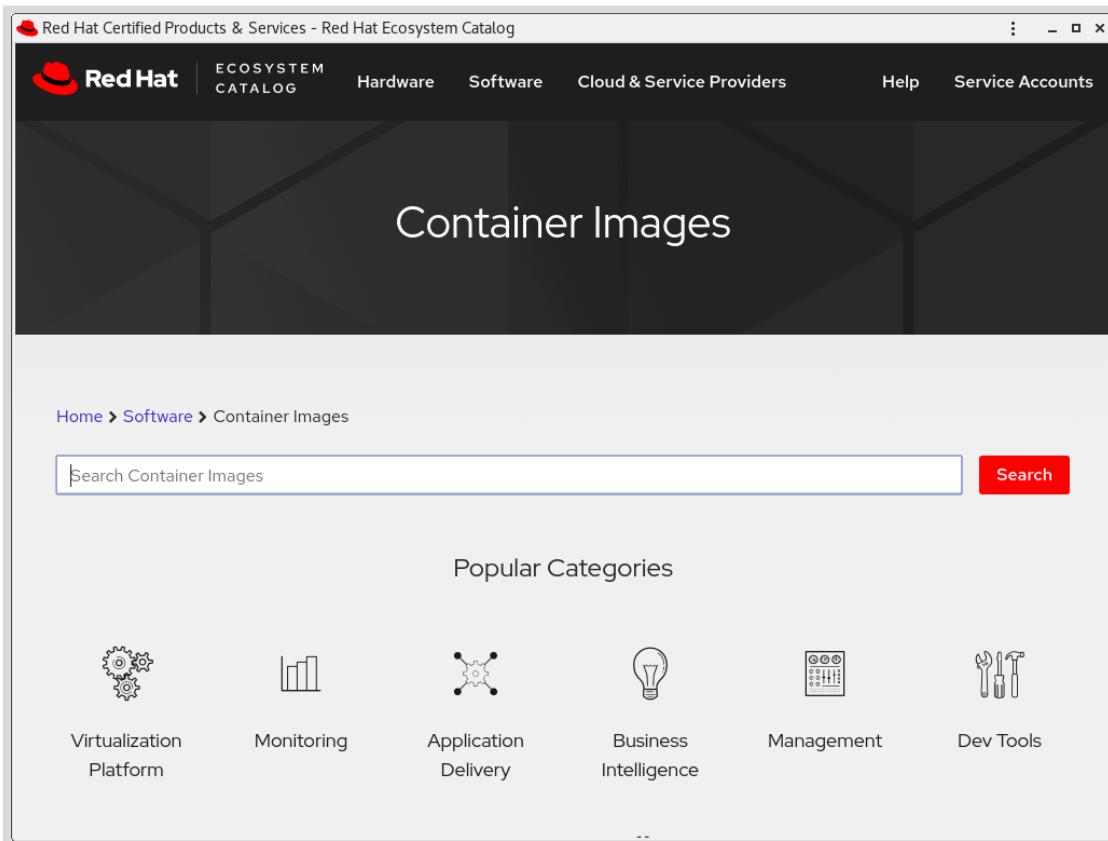


Figure 3.1: Red Hat Container Catalog

Mission-critical applications should rely on these trusted and supported images as much as possible rather than using images from some other public registries. Those other images may not be adequately tested, maintained, or updated promptly when new security issues are detected.

The Red Hat Container Catalog presents a unified view of three underlying container registries:

Red Hat Container Registry at registry.access.redhat.com

It is a public registry that hosts images for Red Hat products and requires no authentication. Note that, while this container registry is public, most of the container images that Red Hat provides require that the user has an active Red Hat product subscription and that they comply with the product's End-User Agreement (EUA). Only a subset of the images available from Red Hat's public registry are freely redistributable. These are images based on the Red Hat Enterprise Linux Universal Base Images (UBI).

Red Hat terms-based registry at registry.redhat.io

It is a private registry that hosts images for Red Hat products, and requires authentication. To pull images from it, you need to authenticate with your Red Hat Customer Portal credentials. For shared environments, such as OpenShift or CI/CD pipelines, you can create a service account, or authentication token, to avoid exposing your personal credentials.

Red Hat partner registry at registry.connect.redhat.com

It is a private registry that hosts images for third-party products from certified partners. It also needs your Red Hat Customer Portal credentials for authentication. They may be subject to subscription or licenses at the partner's discretion.

The Quay.io Registry

Red Hat also manages the Quay.io container registry where anyone can register for a free account and publish their own container images.

Red Hat offers no assurances about any container image hosted at Quay.io. They may range from completely unmaintained, one-time experiments; passing through good, stable, and properly maintained container images from open source communities with no Service-Level Agreement (SLA); to fully supported products by vendors that may offer free, unauthenticated access to their containers images for product trials.

Most users employ Quay.io as a public registry, but organizations can also purchase plans that allow using Quay.io as a private registry.

Deploying Enterprise Container Registries

Accessing external, public, or private registries over the internet is very convenient, but many organizations do not allow developers to pull and run external container images. These organizations restrict developers to a set of container images that pass security, quality, and conformance criteria.

Relying on external registries to pull and push images for your production hosts is not without risks. For example, when the registry is down due to a failure or planned maintenance by the provider, you cannot deploy new containers. In the event of a failure, OpenShift autoscaling also fails, because OpenShift cannot pull the images it needs to start additional pods. Depending on your bandwidth, pushing and pulling images to or from the internet may also be a slow process.

In some organizations, container images are for internal use only and cannot be made public. Setting up an enterprise registry solves this problem. After establishing an enterprise registry, configure container hosts inside the organization to only pull images from this registry rather than the default external registries.

By running a registry server in your organization, you can mitigate risks and implement additional features. For example, you can create different environments and control who can push or pull images to them. You can define an approval workflow to move validated images from development to production. You can implement vulnerability scanning and send a notification when the scanner detects a flaw in an image in production.

Registry Server Software

Among the available registry server software are *Red Hat Quay Enterprise*, the open source Docker-Distribution server, and products such as JFrog and Nexus. OpenShift can deploy containers from any of these registry servers.

Red Hat Quay Enterprise is a container image registry with advanced features such as image security scanning, role-based access, organization and team management, image build automation, auditing, geo-replication, and high availability.

Red Hat Quay Enterprise provides a web interface and a REST API. It can be deployed as a container on premises, in selected cloud providers, and also on Red Hat OpenShift Container Platform. It is also the server software behind Quay.io.

If you deploy Quay Enterprise on OpenShift, it does not replace the cluster's internal registry. A Quay Enterprise instance running on an OpenShift cluster is, for all practical purposes, like any other OpenShift application, and it is usually available to other OpenShift clusters and any other container hosts in your organization.

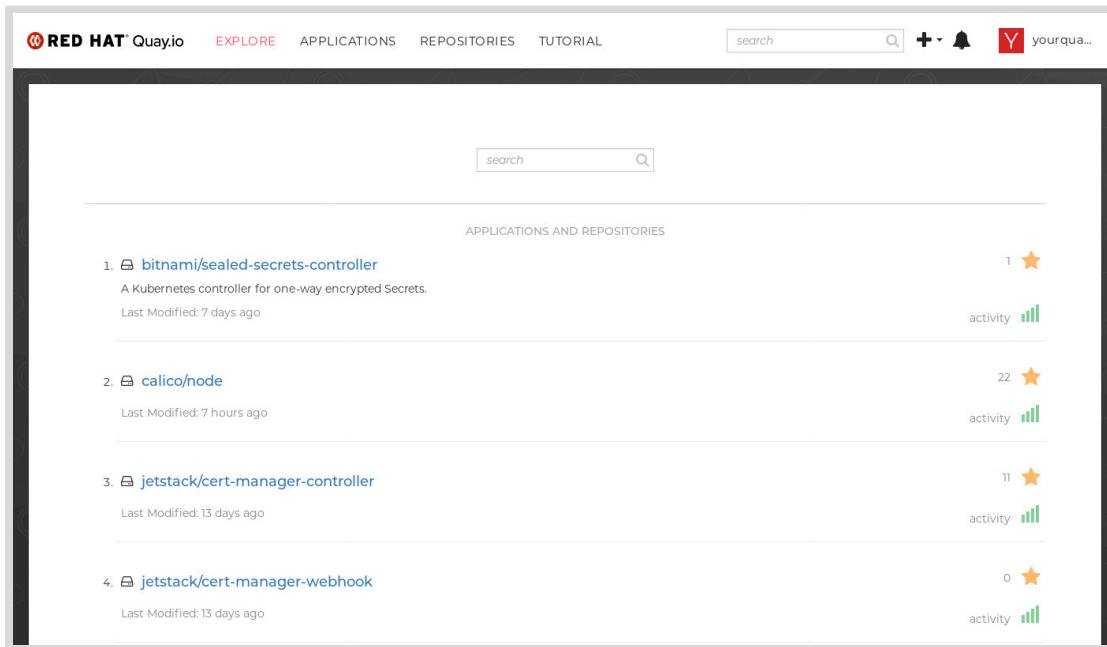


Figure 3.2: Quay.io web page

Accessing Container Registries

Accessing a public registry from OpenShift usually requires no configuration because a public registry is supposed to present a TLS certificate provided by a trusted Certificate Authority (CA).

Accessing a private registry from OpenShift usually requires additional configuration to manage authentication credentials and tokens. An enterprise registry may also require configuration of internal CAs.

To access container registries, you use the *OCI Distribution API*, which is based on the older Docker Registry API. To use the API, Red Hat recommends that you use the Red Hat Enterprise Linux (RHEL) container tools: Podman, Buildah, and Skopeo. The RHEL container tools can customize, deploy, and debug container images based on OCI standards.

The Open Container Initiative (OCI) organization defines open standards for the container runtime, container image format, and related REST APIs. The OCI container image format is a file-system folder with discrete files that store the container image manifest, metadata, and layers.

Managing Containers with Podman

Podman is a tool designed for managing pods and containers without requiring a container daemon, thus reducing the attack surface and improving performance. Pod and container processes are created as child processes of the **podman** command.

Podman can restart stopped containers, and also push, commit, configure, build, and create container images. The **podman** command usually follows the **docker** command syntax and provides additional capabilities, such as managing pods. Podman does not support **docker** commands that are unrelated to the container engine, such as the swarm mode.

Authenticating with Registries

To access a private registry, you usually need to authenticate. Podman provides the **login** subcommand that generates an access token and stores it for subsequent reuse.

```
[user@host ~]$ podman login quay.io
Username: developer1
Password: MyS3cret!
Login Succeeded!
```

After successful authentication, Podman stores an access token in the **/run/user/UID/containers/auth.json** file. The **/run/user/UID** path prefix is not fixed and comes from the **XDG_RUNTIME_DIR** environment variable.

You can simultaneously log in to multiple registries with Podman. Each new login either adds or updates an access token in the same file. Each access token is indexed by the registry server FQDN.

To log out of a registry, use the **logout** subcommand:

```
[user@host ~]$ podman logout quay.io
Remove login credentials for registry.redhat.io
```

To log out of all registries, discarding all access tokens that were stored for reuse, use the **--all** option:

```
[user@host ~]$ podman logout --all
Remove login credentials for all registries
```

Skopeo and Buildah can also use the authentication tokens stored by Podman, but they cannot present an interactive password prompt.

Podman requires TLS and verification of the remote certificate by default. If your registry server is not configured to use TLS, or is configured to use a self-signed TLS certificate or a TLS certificate signed by an unknown CA, you can add the **--tls-verify=false** option to the **login** and **pull** subcommands.

Managing Container Registries with Skopeo

Red Hat supports the **skopeo** command to manage images in a container image registry. Skopeo does not use a container engine so it is more efficient than using the **tag**, **pull**, and **push** subcommands from Podman.

Skopeo also provides additional capabilities not found in Podman, such as signing and deleting container images from a registry server.

The **skopeo** command takes a subcommand, options, and arguments:

```
[user@host ~]$ skopeo subcommand [options] location...
```

Main Subcommands

- **copy** to copy images from one location to another.
- **delete** to delete images from a registry.
- **inspect** to view metadata about an image.

Main Options

--creds *username:password*

To provide login credentials or an authentication token to the registry.

--[src-|dest-]tls-verify=false

Disables TLS certificate verification.

For authentication to private registries, Skopeo can also use the same **auth.json** file created by the **podman login** command. Alternatively, you can pass your credentials on the command line, as shown below.

```
[user@host ~]$ skopeo inspect --creds developer1:MyS3cret! \
> docker://registry.redhat.io/rhscl/postgresql-96-rhel7
```



Warning

Although you can provide credentials to command line tools, this creates an entry in your command history along with other security concerns. Use techniques to avoid passing plain text credentials to commands:

```
[user@host ~]$ read -p "PASSWORD: " -s password
PASSWORD:
[user@host ~]$ skopeo inspect --creds developer1:$password \
> docker://registry.redhat.io/rhscl/postgresql-96-rhel7
```

Skopeo uses URIs to represent container image locations and URI schemas to represent container image formats and registry APIs. The following list shows the most common URI schemas:

oci

denotes container images stored in a local, OCI-formatted folder.

docker

denotes remote container images stored in a registry server.

containers-storage

denotes container images stored in the local container engine cache.

Pushing and Tagging Images in a Registry Server

The **copy** subcommand from Skopeo can copy container images directly between registries, without saving the image layers in the local container storage. It can also copy container images from the local container engine to a registry server and tag these images in a single operation.

To copy a container image named **myimage** from the local container engine to an insecure, public registry at `registry.example.com` under the **myorg** organization or user account:

```
[user@host ~]$ skopeo copy --dest-tls-verify=false \
> containers-storage:myimage \
> docker://registry.example.com/myorg/myimage
```

To copy a container image from the `/home/user/myimage` OCI-formatted folder to the insecure, public registry at `registry.example.com` under the **myorg** organization or user account:

```
[user@host ~]$ skopeo copy --dest-tls-verify=false \
> oci:/home/user/myimage \
> docker://registry.example.com/myorg/myimage
```

When copying container images between private registries, you can either authenticate to both registries using Podman before invoking the **copy** subcommand, or use the **--src-creds** and **--dest-creds** options to specify the authentication credentials, as shown below:

```
[user@host ~]$ skopeo copy --src-creds=testuser:testpassword \
> --dest-creds=testuser1:testpassword \
> docker://srcregistry.domain.com/org1/private \
> docker://dstregistry.domain2.com/org2/private
```

Arguments to the **skopeo** command are always complete image names. The following example is an invalid command because it provides only the registry server name as the destination argument:

```
[user@host ~]$ skopeo copy oci:myimage \
> docker://registry.example.com/
```

The Skopeo **copy** command can also tag images in remote repositories. The following example tags an existing image with tag **1.0** as **latest**:

```
[user@host ~]$ skopeo copy docker://registry.example.com/myorg/myimage:1.0 \
> docker://registry.example.com/myorg/myimage:latest
```

For efficiency, Skopeo does not read or send image layers that already exist at the destination. It first reads the source image manifest, then determines which layers already exist at the destination, and then only copies the missing ones. If you copy multiple images built from the same parent, Skopeo does not copy the parent layers multiple times.

Deleting Images from a Registry

To delete the **myorg/myimage** container image from the registry at `registry.example.com`, run the following command:

```
[user@host ~]$ skopeo delete docker://registry.example.com/myorg/myimage
```

The **delete** subcommand can optionally take the **--creds** and **--tls-verify=false** options.

Authenticating OpenShift to Private Registries

OpenShift also requires credentials to access container images in private registries. These credentials are stored as secrets.

You can provide your private registry credentials directly to the **oc create secret** command:

```
[user@host ~]$ oc create secret docker-registry registrycreds \
> --docker-server registry.example.com \
> --docker-username youruser \
> --docker-password yourpassword
```

Another way of creating the secret is to use the authentication token from the **podman login** command:

```
[user@host ~]$ oc create secret generic registrycreds \
> --from-file .dockerconfigjson=${XDG_RUNTIME_DIR}/containers/auth.json \
> --type kubernetes.io/dockerconfigjson
```

You then link the secret to the **default** service account from your project:

```
[user@host ~]$ oc secrets link default registrycreds --for pull
```

To use the secret to access an S2I builder image, link the secret to the **builder** service account from your project:

```
[user@host ~]$ oc secrets link builder registrycreds
```



References

Open Container Initiative (OCI)

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

A Practical Introduction to Container Terminology

<https://developers.redhat.com/blog/2018/02/22/container-terminology-practical-introduction>

Red Hat Container Registry Authentication

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

Further information about the RHEL container tools is available in the *Building, running, and managing containers* guide for Red Hat Enterprise Linux 8; at https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html-single/building_running_and_managing_containers/index

► Guided Exercise

Using an Enterprise Registry

In this exercise, you will interact with a container image registry server.

Outcomes

You should be able to:

- Push images to an external, authenticated container registry.
- Deploy a containerized application to OpenShift using an external, authenticated container registry as input.

Before You Begin

To perform this exercise, ensure that you have access to:

- A running OpenShift cluster.
- The **podman** and **skopeo** commands.
- OCI-compliant files for the sample **ubi-sleep** container image.

Run the following command on the **workstation** VM to validate the prerequisites and to download the solution files:

```
[student@workstation ~]$ lab external-registry start
```

- 1. Log in to an external registry and push an image to it from an OCI-compliant folder on disk.

- 1.1. Inspect the **ubi-sleep** container OCI image layers on the local disk. OCI images are stored as a file-system folder containing multiple files:

```
[student@workstation ~]$ ls ~/DO288/labs/external-registry/ubi-sleep
blobs  index.json  oci-layout
```

- 1.2. Load your classroom environment configuration.

Run the following command to load the environment variables created in the first guided exercise:

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

- 1.3. Log in to your personal Quay.io account using Podman. Podman prompts you to enter your Quay.io password.

```
[student@workstation ~]$ podman login -u ${RHT_OCP4_QUAY_USER} quay.io
Password:
Login Succeeded!
```

- 1.4. Copy the OCI image to the external registry at Quay.io using Skopeo and tag it as **1.0**.

You can also execute or cut and paste the following **skopeo copy** command from the **push-image.sh** script in the **/home/student/D0288/labs/external-registry** folder.

```
[student@workstation ~]$ skopeo copy \
> oci:/home/student/D0288/labs/external-registry/ubi-sleep \
> docker://quay.io/${RHT_OCP4_QUAY_USER}/ubi-sleep:1.0
...output omitted...
Writing manifest to image destination
Storing signatures
```

- 1.5. Verify that the image exists in the external registry using Podman.



Note

If you cannot find the image in your **podman search** results, move to the next step. Quay.io may truncate search results that would return too many matches.

```
[student@workstation ~]$ podman search quay.io/ubi-sleep
INDEX      NAME
...          ...
...output omitted...
quay.io    quay.io/yourquayuser/ubi-sleep
...          ...
...output omitted...
```

- 1.6. Inspect the image in the external registry using Skopeo:

```
[student@workstation ~]$ skopeo inspect \
> docker://quay.io/${RHT_OCP4_QUAY_USER}/ubi-sleep:1.0
{
  "Name": "quay.io/yourquayuser/ubi-sleep",
  "Tag": "1.0",
  ...output omitted...
```

- 2. Verify that Podman can run images from the external registry.

- 2.1. Start a test container from the image in the external registry. You need to log in to Quay.io again because now you need to run Podman as the **root** user.

```
[student@workstation ~]$ sudo podman login -u ${RHT_OCP4_QUAY_USER} quay.io
Password:
Login Succeeded!
[student@workstation ~]$ sudo podman run -d --name sleep \
> quay.io/${RHT_OCP4_QUAY_USER}/ubi-sleep:1.0
Trying to pull quay.io/youruser/ubi-sleep:1.0...Getting image source signatures
...output omitted...
```

- 2.2. Verify that the new container is running:

```
[student@workstation ~]$ sudo podman ps
CONTAINER ID        IMAGE               ...   NAMES
63c5167376e5      quay.io/youruser/ubi-sleep:1.0 ... sleep
```

- 2.3. Verify that the new container produces log output:

```
[student@workstation ~]$ sudo podman logs sleep
...output omitted...
sleeping
sleeping
```

- 2.4. Stop and remove the test container:

```
[student@workstation ~]$ sudo podman stop sleep
...output omitted...
[student@workstation ~]$ sudo podman rm sleep
...output omitted...
```

▶ 3. Deploy an application to OpenShift based on the image from the external registry:

- 3.1. Log in to OpenShift and create a new project. Prefix the project's name with your developer username.

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful.
...output omitted...
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-external-registry
Now using project "youruser-docker-build" on server "https://
api.cluster.domain.example.com:6443".
```

- 3.2. Try to deploy an application from the container image in the external registry. It will fail because OpenShift needs credentials to access the external registry.

```
[student@workstation ~]$ oc new-app --as-deployment-config --name sleep \
> --docker-image quay.io/${RHT_OCP4_QUAY_USER}/ubi-sleep:1.0
error: unable to locate any local docker images with name "quay.io/yourquayuser/
ubi-sleep:1.0"
...output omitted...
```

- 3.3. Create a secret from the container registry API access token that was stored by Podman.

You can also execute or cut and paste the following **oc create secret** command from the **create-secret.sh** script in the **/home/student/D0288/labs/external-registry** folder.

```
[student@workstation ~]$ oc create secret generic quayio \
> --from-file .dockerconfigjson=${XDG_RUNTIME_DIR}/containers/auth.json \
> --type kubernetes.io/dockerconfigjson
secret/quayio created
```

- 3.4. Link the new secret to the **default** service account.

```
[student@workstation ~]$ oc secrets link default quayio --for pull  
...output omitted...
```

- 3.5. Deploy an application from the container image in the external registry. This time OpenShift can access the external registry.

```
[student@workstation ~]$ oc new-app --as-deployment-config --name sleep \  
> --docker-image quay.io/${RHT_OCP4_QUAY_USER}/ubi-sleep:1.0  
...output omitted...  
--> Creating resources ...  
imagestream.image.openshift.io "sleep" created  
deploymentconfig.apps.openshift.io "sleep" created  
--> Success  
...output omitted...
```

- 3.6. Wait until the application pod is ready and running:

```
[student@workstation ~]$ oc get pod  
NAME        READY   STATUS    RESTARTS   AGE  
sleep-1-deploy  1/1     Running   0          17s  
sleep-1-mmtf8   1/1     Running   0          24s
```

- 3.7. Verify that the pod produces log output:

```
[student@workstation ~]$ oc logs sleep-1-mmtf8  
...output omitted...  
sleeping  
sleeping
```

- 4. Delete the project in OpenShift and the container and image in the external container registry. Because Quay.io allows recovering old container images, you also need to delete your repository on Quay.io.

- 4.1. Delete the OpenShift project:

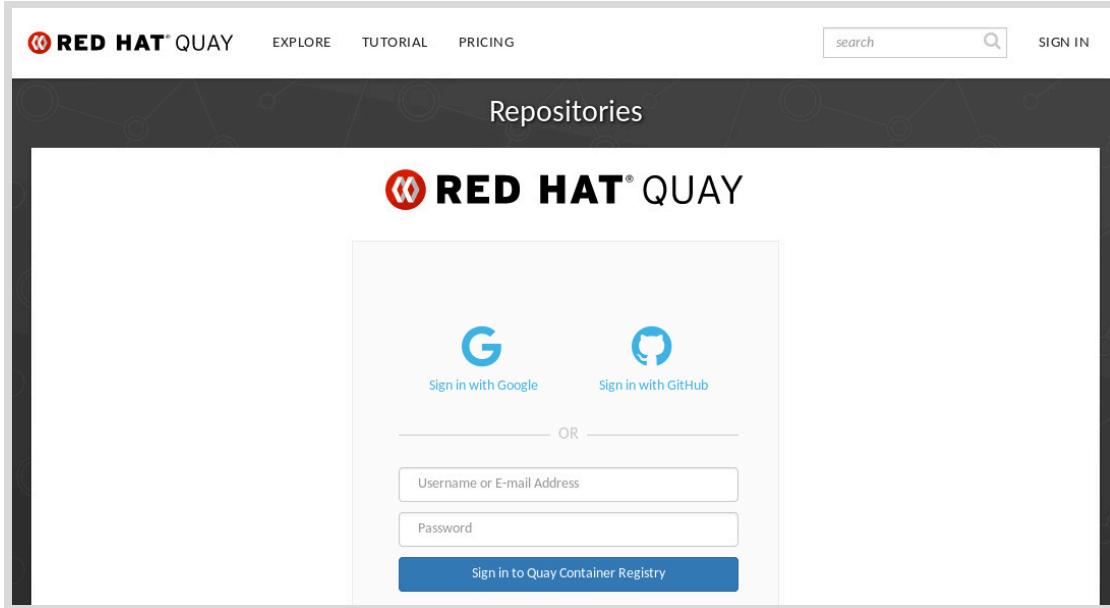
```
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-external-registry  
project.project.openshift.io "external-registry" deleted
```

- 4.2. Delete the container image from the external registry:

```
[student@workstation ~]$ skopeo delete \  
> docker://quay.io/${RHT_OCP4_QUAY_USER}/ubi-sleep:1.0
```

- 4.3. Log in to Quay.io using your personal free account.

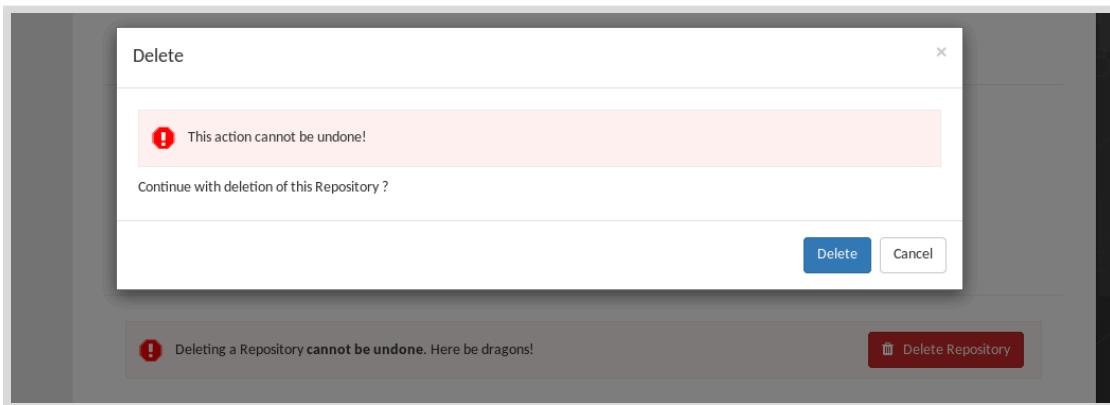
Navigate to <http://quay.io> and click **Sign In** to provide your user credentials. Click **Sign in to Quay Container Registry** to log in to Quay.io.



- 4.4. On the Quay.io main menu, click **Repositories** and look for **ubi-sleep**. The lock icon indicates that it is a private repository that requires authentication for both pulls and pushes. Click **ubi-sleep** to display the **Repository Activity** page.

- 4.5. On the **Repository Activity** page for the **ubi-sleep** repository, scroll down and click the gear icon to display the **Settings** tab. Scroll down and click **Delete Repository**.

- 4.6. In the **Delete** dialog box, click **Delete** to confirm you want to delete the **ubi-sleep** repository. After a few moments you are returned to the **Repositories** page. You can now sign out of Quay.io.



Finish

On the **workstation** VM, run the **lab external-registry finish** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises. The finish action releases this project and its resources.

```
[student@workstation ~]$ lab external-registry finish
```

This concludes the guided exercise.

Allowing Access to the OpenShift Registry

Objectives

After completing this section, you should be able to access the OpenShift internal registry from Linux container tools.

Reviewing the Internal Registry

OpenShift runs an internal registry server to support developer workflows based on Source-to-Image (S2I). Developers create new container images using S2I by creating a build configuration using the `oc new-app` command and other means. A build configuration creates container images from either source code or a Dockerfile and stores them in the internal registry.

Developers are not actually required to use the internal registry. They could build their container images locally, using the Red Hat Enterprise Linux container tools, and push them to an external public or private registry that OpenShift can access. They could also set up their build configurations to push the final image directly to an external public or private registry.

The OpenShift installer deploys an internal registry by default so that developers can start development work as soon as the OpenShift cluster is made available to them. Without an internal registry, developers would need to wait until someone deploys a registry server and provides them with access credentials. They would also have to learn about configuring secrets to access private registries before being able to perform any development work.

Some use cases exist for accessing an OpenShift internal registry outside of the S2I process. For example:

- An organization already builds container images locally, and is not yet ready to change its development workflow. These organizations may have a private registry with limited features and want to replace it with the OpenShift internal registry.
- An organization maintains multiple OpenShift clusters and needs to copy container images from a development to a production cluster. These organizations may have a CI/CD tool that promotes images from an internal registry to either an external registry or another internal registry.
- An Independent Software Vendor (ISV) creates container images for its customers and publishes them to a private registry maintained by a cloud services provider, such as Quay.io, or to an enterprise registry that the ISV maintains.

The OpenShift internal registry may provide all the features that a customer requires, such as fine-grained access controls based on OpenShift users, groups, and roles. The internal registry may provide better features, or improved ease of use, compared to an organization's current enterprise registry, for example if it is based on the Docker-Distribution registry server. These organizations could phase out their current registry and use the OpenShift internal registry as their new enterprise registry.

Other customers may require advanced features, such as image security scanning and geo-replication, and adopt a more powerful enterprise registry server such as Red Hat Quay Enterprise. Customers that adopt a more powerful enterprise registry may still need to expose the internal registry to be able to copy images to the organization's enterprise registry.

The Image Registry Operator

The OpenShift installer configures the internal registry to be accessible only from inside its OpenShift cluster. Exposing the internal registry for external access is a simple procedure, but requires cluster administration privileges.

The OpenShift *Image Registry Operator* manages the internal registry. All configuration settings for the Image Registry operator are in the **cluster** configuration resource in the **openshift-image-registry** project. Change the **spec.defaultRoute** attribute to **true**, and the Image Registry operator creates a route to expose the internal registry. One way to perform that change uses the following **oc patch** command:

```
[user@host ~] oc patch config cluster -n openshift-image-registry \
> --type merge -p '{"spec":{"defaultRoute":true}}'
```

The **default-route** route uses the default wildcard domain name for application deployed to the cluster:

```
[user@host ~] oc get route -n openshift-image-registry
NAME           HOST/PORT
default-route  default-route-openshift-image-registry.domain.example.com ...
```

Authenticating to an Internal Registry

To log in to an internal registry using the Linux container tools, you need to fetch your user's OpenShift authentication token.

Use the **oc whoami -t** command to fetch the token. The token is a long, random string. It is easier to type commands if you save the token as a shell variable:

```
[user@host ~] TOKEN=$(oc whoami -t)
```

Use the token as part of a **login** subcommand from Podman:

```
[user@host ~] podman login -u myuser -p ${TOKEN} \
> default-route-openshift-image-registry.domain.example.com
```

You can also use the token as the value of the **--[src|dst]creds** options from Skopeo.

```
[user@host ~] skopeo inspect --creds=myuser:${TOKEN} \
> docker://default-route-openshift-image-registry.domain.example.com/...
```

Accessing the Internal Registry as a Secure or Insecure Registry

If your OpenShift cluster is configured with a valid TLS certificate for its wildcard domain, you can use the Linux container tools to work with images inside any project you have access to.

The following example uses Skopeo to inspect the container image for the **myapp** application inside the **myproj** project. It assumes that a previous **podman login** was successful.

```
[user@host ~] skopeo inspect \
> docker://default-route-openshift-image-registry.domain.example.com/myproj/myapp
```

If your OpenShift cluster uses the Certification Authority (CA) that the OpenShift installer generates by default, you need to access the internal registry as an insecure registry:

```
[user@host ~] skopeo inspect --tls-verify=false \
> docker://default-route-openshift-image-registry.domain.example.com/myproj/myapp
```

A cluster administrator can configure the route for the internal registry in different ways, for example using an internal CA maintained by your organization. In this scenario, your developer workstation may or may not be already configured to trust the TLS certificate of the internal registry.

Your organization might also retrieve the public certificate of its OpenShift cluster's internal CA and declare it trusted inside your organization. Your cluster administrator might set up an alternative route, with a shorter host name, to expose the internal registry.

These scenarios are outside the scope of this course. Refer to Red Hat Training courses on the OpenShift administration track, such as *Red Hat OpenShift Administration I* (DO280) and *Red Hat Security: Securing Containers and OpenShift* (DO425), for more information about configuring TLS certificates for OpenShift and your local container engine.

Granting Access to Images in an Internal Registry

Any user with access to an OpenShift project can push and pull images to and from that project, according to their access level. If a user has either the **admin** or **edit** roles on the project, they can both pull and push images to that project. If instead they have only the **view** role on the project, they can only pull images from that project.

OpenShift also offers a few specialized roles for when you want to grant access only to images inside a project, and not grant access to perform other development tasks such as building and deploying applications inside the project. The most common of these roles are:

registry-viewer and system:image-puller

These roles allow users to pull and inspect images from the internal registry.

registry-editor and system:image-pusher

These roles allow users to push and tag images to the internal registry.

The **system:*** roles provide the minimum capabilities required to pull and push images to the internal registry. As stated before, OpenShift users who already have **admin** or **edit** roles in a project do not need these **system:*** roles.

The **registry-*** roles provide more comprehensive capabilities around registry management for organizations that want to use the internal registry as their enterprise registry. These roles grant additional rights such as creating new projects but do not grant other rights, such as building and deploying applications. The OCI standards do not specify how to manage an image registry, so whoever manages an OpenShift internal registry needs to know about OpenShift administration concepts and the **oc** command. This makes the **registry-*** less useful.

The following example allows a user to pull images from the internal registry in a given project. You need to have either project or cluster-wide administrator access to use the **oc policy** command.

```
[user@host ~] oc policy add-role-to-user system:image-puller \
> user_name -n project_name
```

General OpenShift authentication and authorization concepts are outside the scope of this course. Refer to Red Hat Training courses on the OpenShift administration track, such as *Red Hat OpenShift Administration I (DO280)* and *Red Hat Security: Securing Containers and OpenShift (DO425)*, for more information about configuring TLS certificates for OpenShift and your local container engine.



References

Further information about exposing the internal registry is available in the *Image Registry Operator in OpenShift Container Platform* chapter of the *Registry* guide for Red Hat OpenShift Container Platform 4.5; at
https://access.redhat.com/documentation/en-us/openshift_container_platform/4.5/html-single/registry/index#configuring-registry-operator

Further information granting access to images in the internal registry is available in the *Accessing the registry* chapter of the *Registry* guide for Red Hat OpenShift Container Platform 4.5; at
https://access.redhat.com/documentation/en-us/openshift_container_platform/4.5/html-single/registry/index#accessing-the-registry

► Guided Exercise

Using the OpenShift Registry

In this exercise, you will access the OpenShift internal registry using Linux container tools.

Outcomes

You should be able to:

- Push an image to an internal registry using the Linux container tools.
- Create a container from an internal registry using the Linux container tools.

Before You Begin

To perform this exercise, ensure you have access to:

- A running OpenShift cluster with its internal registry exposed.
- The **podman** and **skopeo** commands.
- OCI-compliant files for the sample **ubi-info** container image.

Run the following command on the **workstation** VM to validate the prerequisites and to download the solution files:

```
[student@workstation ~]$ lab expose-registry start
```

► 1. Verify that your OpenShift cluster's internal registry is exposed.

1.1. Load your classroom environment configuration.

Run the following command to load the environment variables created in the first guided exercise:

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

1.2. Log in to OpenShift using your developer user account.

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful.
...output omitted...
```

1.3. Verify that there is a route in the **openshift-image-registry** project. The output of the **oc route** command was edited to display each column on a single line for readability purposes because the host name is expected to be too long to fit the paper width.

```
[student@workstation ~]$ oc get route -n openshift-image-registry
NAME          HOST/PORT
...output omitted...
default-route  default-route-openshift-image-
registry.apps.cluster.domain.example.com
...output omitted...
```

**Note**

A default Red Hat OpenShift Container Platform installation does not allow a regular user to view any resources in the **openshift-*** projects. This classroom's cluster assigns all student's developer user accounts additional rights so they perform the previous operation.

- 1.4. To ease typing, save the host name of the internal registry's route into a shell variable.

You can cut and paste the following **oc get** command from the **push-image.sh** script in the **/home/student/D0288/labs/expose-registry** folder.

```
[student@workstation ~]$ INTERNAL_REGISTRY=$( oc get route default-route \
> -n openshift-image-registry -o jsonpath='{.spec.host}' )
```

- 1.5. Verify the value of your **INTERNAL_REGISTRY** shell variable. It should match the output of the first **oc get route** command.

```
[student@workstation ~]$ echo ${INTERNAL_REGISTRY}
default-route-openshift-image-registry.apps.cluster.domain.example.com
```

- ▶ 2. Push an image to the OpenShift internal registry using your developer user account.

- 2.1. Create a project to host the image streams that manage the container images you push to the OpenShift internal registry:

```
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-common
Now using project "youruser-common" on server
"https://api.cluster.domain.example.com:6443".
```

- 2.2. Retrieve your developer user account's OpenShift authentication token to use in later commands:

```
[student@workstation ~]$ TOKEN=$(oc whoami -t)
```

- 2.3. Verify that the **ubi-info** folder contains an OCI-formatted container image:

```
[student@workstation ~]$ ls ~/D0288/labs/expose-registry/ubi-info
blobs  index.json  oci-layout
```

- 2.4. Copy the OCI image to the classroom cluster's internal registry using Skopeo and tag it as **1.0**. Use the host name and the token retrieved in previous steps.

You can cut and paste the following **skopeo copy** command from the **push-image.sh** script in the **/home/student/D0288/labs/expose-registry** folder.

```
[student@workstation ~]$ skopeo copy \
> --dest-creds=${RHT_OCP4_DEV_USER}:${TOKEN} \
> oci:/home/student/D0288/labs/expose-registry/ubi-info \
> docker:///${INTERNAL_REGISTRY}/${RHT_OCP4_DEV_USER}-common/ubi-info:1.0
...output omitted...
Writing manifest to image destination
Storing signatures
```

- 2.5. Verify that an image stream was created to manage the new container image. The output of the **oc get is** command was edited to show each column on a single line for readability purposes because the image repository name is expected to be too long to fit the paper width.

```
[student@workstation ~]$ oc get is
NAME      IMAGE REPOSITORY
ubi-info  default-route-openshift-image-registry.apps....output omitted...
```

- 3. Create a local container from the image in the OpenShift internal registry.
- 3.1. Log in to the OpenShift internal registry using the authentication token from Step 2.2 and the registry host name from Step 1.4:

```
[student@workstation ~]$ sudo podman login -u ${RHT_OCP4_DEV_USER} \
> -p ${TOKEN} ${INTERNAL_REGISTRY}
Login Succeeded!
```

- 3.2. Download the **ubi-info:1.0** container image into the local container engine.

```
[student@workstation ~]$ sudo podman pull \
> ${INTERNAL_REGISTRY}/.${RHT_OCP4_DEV_USER}-common/ubi-info:1.0
...output omitted...
Writing manifest to image destination
Storing signatures
...output omitted...
```

- 3.3. Start a new container from the **ubi-info:1.0** container image. The container displays system information such as the host name and free memory, and then exits. Information that is specific to the running container is omitted in the following output:

```
[student@workstation ~]$ sudo podman run --name info \
> ${INTERNAL_REGISTRY}/.${RHT_OCP4_DEV_USER}-common/ubi-info:1.0
...output omitted...
--- Host name:
...output omitted...
--- Free memory
...output omitted...
--- Mounted file systems (partial)
...output omitted...
```

- 4. Clean up. Delete all resources created during this exercise.

- 4.1. Delete the container image from your classroom cluster's internal registry by deleting its image stream:

```
[student@workstation ~]$ oc delete is ubi-info  
imagestream.image.openshift.io "ubi-info" deleted
```

- 4.2. Delete the OpenShift project:

```
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-common  
project.project.openshift.io "youruser-common" deleted
```

- 4.3. Delete the test container and image from the local container engine:

```
[student@workstation ~]$ sudo podman rm info  
...output omitted...  
[student@workstation ~]$ sudo podman rmi -f \  
> ${INTERNAL_REGISTRY}/${RHT_OCP4_DEV_USER}-common/ubi-info:1.0  
...output omitted...
```

Finish

On the **workstation** VM, run the **lab expose-registry finish** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises. The finish action releases this project and its resources.

```
[student@workstation ~]$ lab expose-registry finish
```

This concludes the guided exercise.

Creating Image Streams

Objectives

After completing this section, you should be able to create image streams for container images in external registries.

Describing Image Streams

Image streams are one of the main differentiators between OpenShift and upstream Kubernetes. Kubernetes resources reference container images directly, but OpenShift resources, such as deployment configurations and build configurations, reference image streams. OpenShift also extends Kubernetes resources, such as StatefulSet and CronJob resources, with annotations that make them work with OpenShift image streams.

Image streams allow OpenShift to ensure reproducible, stable deployments of containerized applications and also rollbacks of deployments to their latest known-good state.

Image streams provide a stable, short name to reference a container image that is independent of any registry server and container runtime configuration.

As an example, an organization could start by downloading container images directly from the Red Hat public registry and later set up an enterprise registry as a mirror of those images to save bandwidth. OpenShift users would not notice any change because they still refer to these images using the same image stream name. Users of the RHEL container tools would notice the change because they would be required to either change the registry names in their commands or change their container engine configurations to search for the local mirror first.

There are other scenarios where the indirection provided by an image stream proves to be helpful. Suppose you start with a database container image that has security issues and the vendor takes too long to update the image with fixes. Later you find an alternative vendor that provides an alternative container image for the same database, with those security issues already fixed, and even better, with a track record of providing timely updates to them. If those container images are compatible regarding configuration of environment variables and volumes, you could simply change your image stream to point to the image from the alternative vendor.

Red Hat provides hardened, supported container images that work mostly as drop-in replacements of container images from some popular open source projects, such as the MariaDB database.

Describing Image Stream Tags

An image stream represents one or more sets of container images. Each set, or stream, is identified by an *image stream tag*. Unlike container images in a registry server, which have multiple tags from the same image repository (or user or organization), an image stream can have multiple image stream tags that reference container images from different registry servers and from different image repositories.

An image stream provides default configurations for a set of image stream tags. Each image stream tag references one stream of container images, and can override most configurations from its associated image stream.

Chapter 3 | Publishing Enterprise Container Images

An image stream tag stores a copy of the metadata about its current container image and can optionally store a copy of its current and past container image layers. Storing metadata allows faster search and inspection of container images, because you do not need to reach its source registry server.

Storing image layers allows an image stream tag to act as a local image cache, avoiding the need to fetch these layers from their source registry server. The image stream tag stores its cached image layers in the OpenShift internal registry. Consumers of the cached image, such as pods and deployment configurations, just reference the internal registry as the source registry of the image.

There are a few other OpenShift resource types related to image streams, but a developer can usually dismiss them as implementation details of the internal registry and only care about image streams and image stream tags.

To better visualize the relationship between image streams and image stream tags, you can explore the **openshift** project that is pre-created in all OpenShift clusters. You can see there are a number of image streams in that project, including the **php** image stream:

```
[user@host ~]$ oc get is -n openshift -o name
...output omitted...
imagestream.image.openshift.io/nodejs
imagestream.image.openshift.io/perl
imagestream.image.openshift.io/php
imagestream.image.openshift.io/postgresql
imagestream.image.openshift.io/python
...output omitted...
```

A number of tags exist for the **php** image stream, and an image stream resource exists for each:

```
[user@host ~]$ oc get istag -n openshift | grep php
php:7.2      image-registry...output omitted...    6 days ago
php:7.3      image-registry...output omitted...    6 days ago
php:latest   image-registry...output omitted...    6 days ago
```

If you use the **oc describe** command on an image stream, it shows information from both the image stream and its image stream tags:

```
[user@host ~]$ oc describe is php -n openshift
Name:                  php
Namespace:             openshift
...output omitted...
Tags:                  3

7.3 (latest)
tagged from registry.redhat.io/rhscl/php-73-rhel7:latest
...output omitted...

7.2
tagged from registry.redhat.io/rhscl/php-72-rhel7:latest
...output omitted...
```

In the previous example, each of the **php** image stream tags refers to a different image name.

Describing Image Names, Tags, and IDs

The textual name of a container image is simply a string. This name is sometimes interpreted as being made of multiple components, such as **registry-host-name/repository-or-organization-or-user-name/image-name:tag-name**, but splitting the image name into its components is just a matter of convention, not of structure.

An image ID uniquely identifies an immutable container image using a SHA-256 hash. Remember that you can not modify a container image. Instead, you create a new container image that has a new ID. When you push a new container image to a registry server, the server associates the existing textual name with the new image ID.

When you start a container from an image name, you download the image that is currently associated to that image name. The actual image ID behind that name may change at any moment, and the next container you start may have a different image ID. If the image associated with an image name has any issues, and you only know the image name, you cannot rollback to an earlier image.

OpenShift image stream tags keep a history of the latest image IDs that they fetched from a registry server. The history of image IDs is the *stream* of images from an image stream tag. You can use the history inside an image stream tag to roll back to a previous image, if for example a new container image causes a deployment error.

Updating a container image in an external registry does not automatically update an image stream tag. The image stream tag keeps the reference to the last image ID it fetched. This behavior is crucial to scaling applications because it isolates OpenShift from changes happening at a registry server.

Suppose you deploy an application from an external registry, and after a few days of testing with a few users, you decide to scale its deployment to enable a larger user population. In the meantime, your vendor updates the container image on the external registry. If OpenShift had no image stream tags, the new pods would get the new container image, which is different from the image on the original pod. Depending on the changes, this could cause your application to fail. Because OpenShift stores the image ID of the original image in an image stream tag, it can create new pods using the same image ID and avoid any incompatibility between the original and updated image.

OpenShift keeps the image ID used for the first pod and ensures that new pods use the same image ID. OpenShift ensures that all pods use exactly the same image.

To better visualize the relationship between an image stream, an image stream tag, an image name, and an image ID, refer to the following **oc describe is** command, which shows the source image and current image ID for each image stream tag:

```
[user@host ~]$ oc describe is php -n openshift
Name:          php
Namespace:     openshift
...output omitted...
7.3 (latest)
  tagged from registry.redhat.io/rhscl/php-73-rhel7:latest
  ...output omitted...
  * registry.redhat.io/rhscl/php-73-rhel7@sha256:22ba...09b5
  ...output omitted...
7.2
  tagged from registry.redhat.io/rhscl/php-72-rhel7:latest
```

```
...output omitted...
* registry.redhat.io/rhscl/php-72-rhel7@sha256:e8d6...e615
...output omitted...
```

If your OpenShift cluster administrator already updated the **php:7.3** image stream tag, the **oc describe is** command shows multiple image IDs for that tag:

```
[user@host ~]$ oc describe is php -n openshift
Name:                  php
Namespace:             openshift
...output omitted...
7.3 (latest)
tagged from registry.redhat.io/rhscl/php-73-rhel7:latest
...output omitted...
* registry.redhat.io/rhscl/php-73-rhel7@sha256:22ba...09b5
...output omitted...
registry.redhat.io/rhscl/php-73-rhel7@sha256:bc61...1e91
...output omitted...
7.2
tagged from registry.redhat.io/rhscl/php-72-rhel7:latest
...output omitted...
* registry.redhat.io/rhscl/php-72-rhel7@sha256:e8d6...e615
...output omitted...
```

In the previous example, the asterisk (*) shows which image ID is the current one for each image stream tag. It is usually the latest one to be imported, which is the first one listed.

When an OpenShift image stream tag references a container image from an external registry, you need to explicitly update the image stream tag to get new image IDs from the external registry. By default, OpenShift does not monitor external registries for changes to the image ID that is associated with an image name.

You can configure an image stream tag to check the external registry for updates on a defined schedule. By default, new image stream tags do not check for updated images.

Build configurations automatically update the image stream tag that they use as the output image. This forces redeployment of application pods using that image stream tag.

Managing Image Streams and Tags

To create an image stream tag resource for a container image hosted on an external registry, use the **oc import-image** command with both the **--confirm** and **--from** options. The following command updates an image stream tag or creates one if it does not exist:

```
[user@host ~]$ oc import-image myimagestream[:tag] --confirm \
> --from registry/myorg/myimage[:tag]
```

If you do not specify a tag name, the **latest** tag is used by default. In this example, the image stream tag references the **myimagestream** image stream. If the corresponding image stream does not yet exist, OpenShift creates it.

**Note**

To create an image stream for container images hosted on a registry server that is not set up with a trusted TLS certificate, add the **--insecure** option to the **oc import-image** command.

The tag name for the image stream tag can be different from the container image tag on the source registry server. The following example creates a **1.0** image stream tag from the source registry server **latest** tag (by omission):

```
[user@host ~]$ oc import-image myimagestream:1.0 --confirm \
> --from registry/myorg/myimage
```

To create one image stream tag resource for each container image tag that exists in the source registry server, add the **--all** option to the **oc import-image** command:

```
[user@host ~]$ oc import-image myimagestream --confirm --all \
> --from registry/myorg/myimage
```

You can run the **oc import-image** command on an existing image stream to update one of its current image stream tags to the current image IDs on the source registry server.

```
[user@host ~]$ oc import-image myimagestream[:tag] --confirm
```

Updating an image stream with the **--all** option updates all image stream tags and also creates new image stream tags for new tags it finds on the source registry server.

You can exert finer control over an image stream tag using the **oc tag** command. It allows changes such as:

- Associating an image stream tag to a different registry server than the server associated with its image stream.
- Associating an image stream tag to a different container image name and tag.
- Associating an image stream tag to a given image ID, which may not be the one currently associated with that image tag on the registry server.
- Associating an image stream with another image stream tag. For example, the previous example of the **oc describe is** command shows that the **php:latest** image stream tag follows the **php:7.3** image stream tag. This is a way of creating aliases for image stream tags.

It is outside the scope of this course to teach all aspects of image stream management, although some of them, such as image change events, are explored in later chapters.

Using Image Streams with Private Registries

To create image streams and image stream tags that refer to a private registry, OpenShift needs an access token to that registry server.

You provide that access token as a secret, the same way you would to deploy an application from a private registry, and you do not need to link the secret to any service account. The **oc import-image** command searches the secrets in the current project for one that matches the registry host name.

The following example uses Podman to log in to a private registry, create a secret to store the access token, and then create an image stream that points to the private registry:

```
[user@host ~]$ podman login -u myuser registry.example.com
[user@host ~]$ oc create secret generic regtoken \
> --from-file .dockerconfigjson=${XDG_RUNTIME_DIR}/containers/auth.json \
> --type kubernetes.io/dockerconfigjson
[user@host ~]$ oc import-image myis --confirm \
> --from registry.example.com/myorg/myimage
```

After you create an image stream you can use it to deploy an application using the **oc new-app --as-deployment-config -i myis** command. You can also use that image stream as a builder image in the **oc new-app --as-deployment-config myis~giturl** command.

By default, an image stream resource is only available to create applications or builds in the same project.

Sharing an Image Stream Between Multiple Projects

It is a common practice to create projects in OpenShift to store resources that are shared between multiple users and development teams. These shared projects store resources such as image streams and templates, which developers refer to when deploying applications to their projects.

OpenShift comes with a shared project named **openshift**, which provides quick-start application templates and also image streams for S2I builders for popular programming languages such as Python and Ruby. Some organizations create similar projects for their teams instead of adding resources to the **openshift** project, to avoid issues when upgrading the cluster to a new release of OpenShift.



Important

You had the option of adding new resources to the **openshift** project in earlier releases, but since Red Hat OpenShift Container Platform 4, the *Samples* operator manages the **openshift** project and may remove resources you manually add to it at any time.

To build and deploy applications using an image stream that is defined in another project, you have two options:

- Create a secret with an access token to the private registry on each project that uses the image stream, and link that secret to each project's service accounts.
- Create a secret with an access token to the private registry only on the project where you create the image stream, and configure that image stream with a local reference policy. Grant rights to use the image stream to service accounts from each project that uses the image stream.

The first option resembles what you would do to deploy an application from a container image in a private registry. It negates some of the benefits of using image streams because if the image stream tag that you refer to changes to reference another registry server, you need to create a new secret for the new registry server in all projects.

The second option allows projects that reference the image stream to remain isolated from changes in the image stream tags they consume. The extra work of assigning permissions to service accounts comes from the fact that service accounts have rights that are more restricted than the user account that creates them.

The following example demonstrates the second option. It creates an image stream in the **shared** project and uses that image stream to deploy an application in the **myapp** project.

```
[user@host ~]$ podman login -u myuser registry.example.com
[user@host ~]$ oc project shared
[user@host ~]$ oc create secret generic regtoken \
> --from-file .dockerconfigjson=${XDG_RUNTIME_DIR}/containers/auth.json \
> --type kubernetes.io/dockerconfigjson
[user@host ~]$ oc import-image myis --confirm \
> --reference-policy local \ ❶
> --from registry.example.com/myorg/myimage
[user@host ~]$ oc policy add-role-to-group system:image-puller \ ❷
> system:serviceaccounts:myapp \ ❸
[user@host ~]$ oc project myapp
[user@host ~]$ oc new-app --as-deployment-config -i shared/myis
```

There are three crucial details in the previous example:

- ❶ The **--reference-policy local** option of the **oc import-image** command. It configures the image stream to cache image layers in the internal registry, so projects that reference the image stream do not need an access token to the external private registry.
- ❷ The **system:image-puller** role allows a service account to pull the image layers that the image stream cached in the internal registry.
- ❸ The **system:serviceaccounts:myapp** group. This group includes all service accounts from the **myapp** project. The **oc policy** command can refer to users and groups that do not exist yet.

The **oc policy** command does not require cluster administrator privileges; it requires only project administrator privileges.

The image stream from the previous example can also provide the builder image for the **oc new-app --as-deployment-config shared/myis~giturl** command.



References

Further information about image streams, image stream tags, and image IDs is available in the *Understanding Containers, Images, and Imagestreams* chapter of the *Images* guide for Red Hat OpenShift Container Platform 4.5; at https://access.redhat.com/documentation/en-us/openshift_container_platform/4.5/html-single/images/index#understanding-images

► Guided Exercise

Creating an Image Stream

In this exercise, you will deploy a "hello, world" application based on Nginx, using an image stream.

Outcomes

You should be able to:

- Publish an image from an external registry as an image stream in OpenShift.
- Deploy an application using the image stream.

Before You Begin

To perform this exercise, ensure that you have access to:

- A running OpenShift cluster.
- The "hello, world" application container image (**redhattraining/hello-world-nginx**).

Run the following command on the **workstation** VM to validate the prerequisites:

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

- 1. Log in to OpenShift and create a project to host the image stream for the Nginx container image.

- 1.1. Load your classroom environment configuration.

Run the following command to load the environment variables created in the first guided exercise:

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

- 1.2. Log in to OpenShift using your developer user account.

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful.
...output omitted...
```

- 1.3. Create a project to host the image streams that are potentially shared among multiple projects:

```
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-common
Now using project "youruser-common" on server
"https://api.cluster.domain.example.com:6443".
```

- 2. Create an image stream that points to the Nginx image from the external registry.

- 2.1. Verify that the **redhattraining/hello-world-nginx** image from Quay.io has a single tag named **latest**.

```
[student@workstation ~]$ skopeo inspect \
> docker://quay.io/redhattraining/hello-world-nginx
{
  "Name": "quay.io/redhattraining/hello-world-nginx",
  "Tag": "latest",
  "Digest": "sha256:4f4f...acc1",
  "RepoTags": [
    "latest"
  ],
  ...output omitted...
```

- 2.2. Create the **hello-world** image stream that points to the **redhattraining/hello-world-nginx** container image from Quay.io:

```
[student@workstation ~]$ oc import-image hello-world --confirm \
> --from quay.io/redhattraining/hello-world-nginx
imagestream.image.openshift.io/hello-world imported
...output omitted...
Name:      hello-world
Namespace: youruser-common
...output omitted...
Unique Images: 1
Tags:      1

latest
tagged from quay.io/redhattraining/hello-world-nginx

...output omitted...
```

- 2.3. Verify that the **hello-world:latest** image stream tag is created:

```
[student@workstation ~]$ oc get istag
NAME          IMAGE REF
...
hello-world:latest  quay.io/redhattraining/hello-world-nginx@sha256:4f4f...acc1
```

- 2.4. Verify that the image stream and its tag contain metadata about the Nginx container image:

```
[student@workstation ~]$ oc describe is hello-world
Name:      hello-world
Namespace: youruser-common
...output omitted...
Tags:      1

latest
tagged from quay.io/redhattraining/hello-world-nginx ①
* quay.io/redhattraining/hello-world-nginx@sha256:4f4f...acc1 ②
```

```
2 minutes ago
```

```
...output omitted...
```

- ❶ The image stream tag **hello-world:latest** references an image from Quay.io.
- ❷ The asterisk (*) indicates that the **latest** image stream tag references only the particular image with the given SHA-256 identifier.

► 3. Create a new project and deploy an application using the **hello-world** image stream from the **youruser-common** project.

3.1. Create a project to host the test application:

```
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-image-stream
Now using project "youruser-image-stream" on server
"https://api.cluster.domain.example.com:6443".
```

3.2. Deploy an application from the image stream:

```
[student@workstation ~]$ oc new-app --as-deployment-config --name hello \
> -i ${RHT_OCP4_DEV_USER}-common/hello-world
--> Found image 44eaa13 (20 hours old) in image stream "youruser-common/hello-
world" under tag "latest" for "youruser-common/hello-world"
...output omitted...
--> Creating resources ...
  imagestreamtag.image.openshift.io "hello:latest" created
  deploymentconfig.apps.openshift.io "hello" created
  service "hello" created
--> Success
...output omitted...
```

3.3. Wait until the application pod is ready and running:

```
[student@workstation ~]$ oc get pod
NAME        READY   STATUS    RESTARTS   AGE
hello-1-deploy  1/1     Completed   0          1m
hello-1-tzjwn  1/1     Running    0          1m
```

3.4. Create a route to expose the application:

```
[student@workstation ~]$ oc expose svc hello
route.route.openshift.io/hello exposed
```

3.5. Get the host name of the route:

```
[student@workstation ~]$ oc get route
NAME      HOST/PORT
hello     hello-youruser-image-stream.apps.cluster.domain.example.com ...
```

3.6. Test the application using the **curl** command and the host name from the previous step.

```
[student@workstation ~]$ curl \
> http://hello-${RHT_OCP4_DEV_USER}-image-stream.${RHT_OCP4_WILDCARD_DOMAIN}
...output omitted...
<h1>Hello, world from nginx!</h1>
...output omitted...
```

- 4. Delete the **youruser-commons** and **youruser-image-stream** projects.

```
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-image-stream
project.project.openshift.io "youruser-image-stream" deleted
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-common
project.project.openshift.io "youruser-image-common" deleted
```

Finish

On the **workstation** VM, run the **lab image-stream finish** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises. The finish action releases this project and its resources.

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

This concludes the guided exercise.

► Lab

Publishing Enterprise Container Images

Performance Checklist

In this lab, you will publish an OCI-formatted container image to an external registry and deploy an application from that image using an image stream.



Note

The **grade** command used at the end of each chapter lab requires that you use exact project names and other identifiers, as stated in the specification of the lab.

Outcomes

You should be able to create an application from an image stored in an external registry.

Before You Begin

To perform this exercise, ensure that you have access to:

- A running OpenShift cluster.
- The OCI-compliant files for the sample container image (**php-info**).

Run the following command on the **workstation** VM to validate the prerequisites and to download the solution files:

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

Requirements

The application image contains a PHP application that displays the web server environment and the PHP interpreter configuration. Deploy the application as follows:

- Host the image stream for the image built outside of OpenShift in a project called **youruser-common**.

The container image name is **php-info**. The OCI-formatted image layers and manifest are in the **/home/student/D0288/labs/expose-image/php-info** folder. Push that image into a private image repository in your Quay.io account.

- Deploy the application in a project called **youruser-expose-image**.

The application's resources are called info. Access the application using the default host name assigned by OpenShift.

- Use the **/usr/local/etc/ocp4.config** configuration file to get classroom configuration data such as the OpenShift cluster's Master API URL.

Steps

1. Push the **php-info** OCI-formatted container image to Quay.io.
2. As your developer user, create the **youruser-common** project to host the image stream that points to the image in the external registry. Create a secret with login credentials to Quay.io and create the **php-info** image stream.

Create the image stream using the **--reference-policy local** option so that other projects that use that image stream can also use the secret stored in the **youruser-common** project.

3. Create a new project named **youruser-expose-image**. Then create a new application by deploying the container image from the image stream.

Grant the **system:image-puller** role on the **youruser-common** project to all service accounts in the **youruser-expose-image** project. This role allows pods from one project to use image streams from another project. The **grant-puller-role.sh** script in the **/home/student/D0288/labs/expose-image** folder contains the **oc policy** command that performs this operation.

4. Expose and test the application. Verify that the application returns the PHP interpreter configuration page.
5. Grade your work.

Run the following command on the **workstation** VM to verify that all tasks were accomplished:

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

6. Delete the OpenShift projects and remove the image repository from Quay.io.

Finish

On the **workstation** VM, run the **lab expose-image finish** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises. The finish action releases this project and its resources.

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

This concludes the lab.

► Solution

Publishing Enterprise Container Images

Performance Checklist

In this lab, you will publish an OCI-formatted container image to an external registry and deploy an application from that image using an image stream.



Note

The **grade** command used at the end of each chapter lab requires that you use exact project names and other identifiers, as stated in the specification of the lab.

Outcomes

You should be able to create an application from an image stored in an external registry.

Before You Begin

To perform this exercise, ensure that you have access to:

- A running OpenShift cluster.
- The OCI-compliant files for the sample container image (**php-info**).

Run the following command on the **workstation** VM to validate the prerequisites and to download the solution files:

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

Requirements

The application image contains a PHP application that displays the web server environment and the PHP interpreter configuration. Deploy the application as follows:

- Host the image stream for the image built outside of OpenShift in a project called **youruser-common**.

The container image name is **php-info**. The OCI-formatted image layers and manifest are in the **/home/student/D0288/labs/expose-image/php-info** folder. Push that image into a private image repository in your Quay.io account.

- Deploy the application in a project called **youruser-expose-image**.

The application's resources are called info. Access the application using the default host name assigned by OpenShift.

- Use the **/usr/local/etc/ocp4.config** configuration file to get classroom configuration data such as the OpenShift cluster's Master API URL.

Steps

- Push the **php-info** OCI-formatted container image to Quay.io.

- Verify that the **php-info** folder contains an OCI-formatted container image:

```
[student@workstation ~]$ ls ~/D0288/labs/expose-image/php-info
blobs    index.json    oci-layout
```

- Load your classroom environment configuration.

Run the following command to load the configuration variables created in the first guided exercise:

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

- Use the **podman** command to log in to Quay.io

```
[student@workstation ~]$ podman login -u ${RHT_OCP4_QUAY_USER} quay.io
Password:
Login Succeeded!
```

- Use the **skopeo** command to push the OCI-formatted container image to Quay.io. You can copy or run the **skopeo** command from the **push-image.sh** script in the **/home/student/D0288/labs/expose-image** folder.

```
[student@workstation ~]$ skopeo copy \
> oci:/home/student/D0288/labs/expose-image/php-info \
> docker://quay.io/${RHT_OCP4_QUAY_USER}/php-info
...output omitted...
Writing manifest to image destination
Storing signatures
```

- Inspect the image in the external registry using Skopeo to verify it is tagged as **latest**.

```
[student@workstation ~]$ skopeo inspect \
> docker://quay.io/${RHT_OCP4_QUAY_USER}/php-info
{
  "Name": "quay.io/yourquayuser/php-info",
  "Tag": "latest",
  ...output omitted...
```

- As your developer user, create the **youruser-common** project to host the image stream that points to the image in the external registry. Create a secret with login credentials to Quay.io and create the **php-info** image stream.

Create the image stream using the **--reference-policy local** option so that other projects that use that image stream can also use the secret stored in the **youruser-common** project.

- Log in to OpenShift using your developer user account:

```
[student@workstation D0288-apps]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful
...output omitted...
```

- 2.2. Create a project to host the image stream.

```
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-common
Now using project "youruser-common" on server
"https://api.cluster.domain.example.com:6443".
```

- 2.3. Create a secret from the container registry API access token that was stored by Podman.

You can copy or run the **oc create secret** command from the **create-secret.sh** script in the **/home/student/D0288/labs/expose-image** folder.

```
[student@workstation ~]$ oc create secret generic quayio \
> --from-file .dockerconfigjson=${XDG_RUNTIME_DIR}/containers/auth.json \
> --type kubernetes.io/dockerconfigjson
secret/quayio created
```

- 2.4. Import the container image using the **oc import-image** command:

```
[student@workstation ~]$ oc import-image php-info --confirm \
> --reference-policy local \
> --from quay.io/${RHT_OCP4_QUAY_USER}/php-info
imagestream.image.openshift.io/php-info imported
...output omitted...
latest
tagged from quay.io/youruser/php-info
will use insecure HTTPS or HTTP connections

* quay.io/youruser/php-info@sha256:4366...f937
  Less than a second ago
...output omitted...
```

- 2.5. Verify that an image stream tag was created and contains metadata about the **php-info** container image:

```
[student@workstation ~]$ oc get istag
NAME          IMAGE REF   ...
php-info:latest  image-registry.openshift-image-registry.svc:5000/youruser-
common/php-info@sha256:4366...f937  ...
```

3. Create a new project named **youruser-expose-image**. Then create a new application by deploying the container image from the image stream.

Grant the **system:image-puller** role on the **youruser-common** project to all service accounts in the **youruser-expose-image** project. This role allows pods from one project to use image streams from another project. The **grant-puller-role.sh** script in the **/**

home/student/D0288/labs/expose-image folder contains the **oc policy** command that performs this operation.

3.1. Create a project to host the application:

```
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-expose-image
```

3.2. Grant service accounts from the new **youruser-expose-image** project access to image streams from the **youruser-common** project. You can copy or run the following **oc policy** command from the **grant-puller-role.sh** script in the **/home/student/D0288/labs/expose-image** folder.

```
[student@workstation ~]$ oc policy add-role-to-group \
> -n ${RHT_OCP4_DEV_USER}-common system:image-puller \
> system:serviceaccounts:${RHT_OCP4_DEV_USER}-expose-image
clusterrole.rbac.authorization.k8s.io/system:image-puller added:
"system:serviceaccounts:youruser-expose-image"
```

3.3. Deploy the test application from the image stream in the **common** project:

```
[student@workstation ~]$ oc new-app --as-deployment-config --name info \
> -i ${RHT_OCP4_DEV_USER}-common/php-info
...output omitted...
--> Creating resources ...
imagestreamtag.image.openshift.io "info:latest" created
deploymentconfig.apps.openshift.io "info" created
service "info" created
--> Success
...output omitted...
```

3.4. Wait for the application pod to be ready and running:

[student@workstation ~]\$ oc get pod				
NAME	READY	STATUS	RESTARTS	AGE
info-1-deploy	1/1	Running	0	22s
info-1-80rkd	1/1	Running	0	22s

4. Expose and test the application. Verify that the application returns the PHP interpreter configuration page.

4.1. Expose the info application:

```
[student@workstation ~]$ oc expose svc info
route.route.openshift.io/info exposed
```

4.2. Get the host name that OpenShift assigns to the route:

```
[student@workstation ~]$ oc get route info
NAME      HOST/PORT
info      info-youruser-expose-image.apps.cluster.domain.example.com
```

- 4.3. Test the application using the **curl** command and the route from the previous step.
Alternatively you can use a web browser.

```
[student@workstation ~]$ curl \
> http://info-${RHT_OCP4_DEV_USER}-expose-image.${RHT_OCP4_WILDCARD_DOMAIN}
...output omitted...
<title>phpinfo()</title><meta name="ROBOTS" content="NOINDEX, NOFOLLOW, NOARCHIVE" />
</head>
...output omitted...
<tr><td class="e">Server API </td><td class="v">FPM/FastCGI </td></tr>
...output omitted...
<tr><td class="e">PHP API </td><td class="v">20170718 </td></tr>
...output omitted...
```

5. Grade your work.

Run the following command on the **workstation** VM to verify that all tasks were accomplished:

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

6. Delete the OpenShift projects and remove the image repository from Quay.io.

- 6.1. Delete the **youruser-expose-image** project:

```
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-expose-image
project.project.openshift.io "youruser-expose-image" deleted
```

- 6.2. Delete the **youruser-common** project:

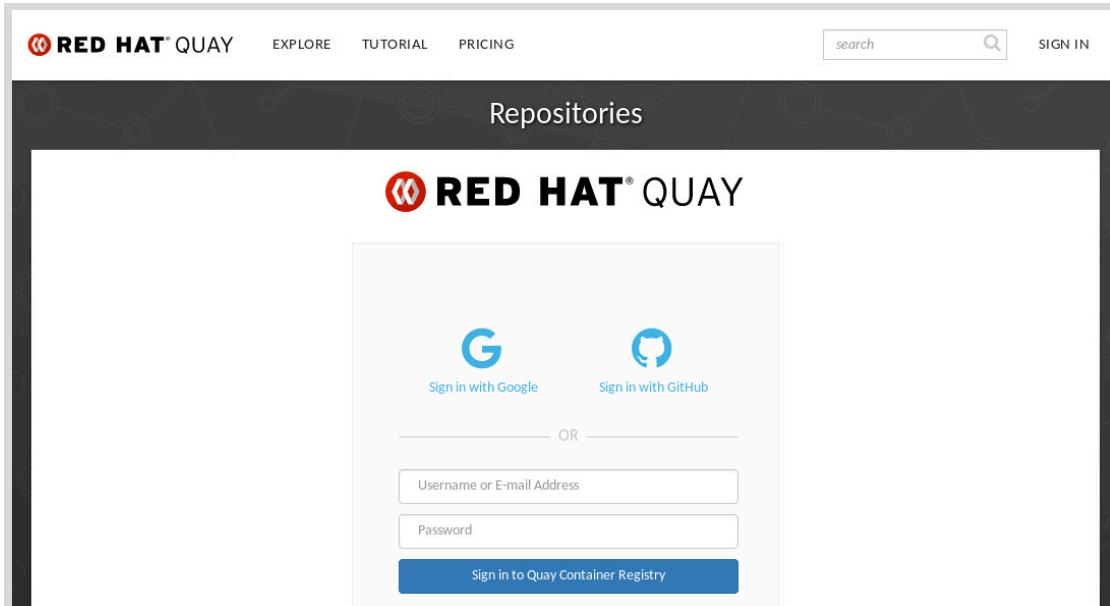
```
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-common
project.project.openshift.io "youruser-common" deleted
```

- 6.3. Delete the container image from the external registry:

```
[student@workstation ~]$ skopeo delete \
> docker://quay.io/${RHT_OCP4_QUAY_USER}/php-info:latest
```

- 6.4. Log in to Quay.io using your personal free account.

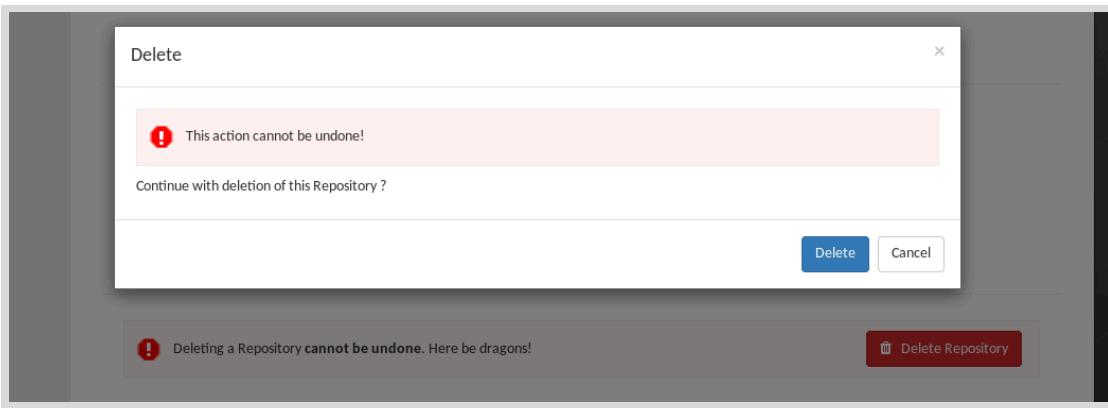
Navigate to <http://quay.io> and click **Sign In** to provide your user credentials. Click **Sign in to Quay Container Registry** to log in on Quay.io.



- 6.5. On the Quay.io main menu, click **Repositories** and look for **php-info**. Click **php-info** to open the **Repository Activity** page.

- 6.6. On the **Repository Activity** page for the **php-info** repository, scroll down and click the gear icon to display the **Settings** tab. Scroll down and click **Delete Repository**.

- 6.7. In the **Delete** dialog box, click **Delete** to confirm you want to delete the **php-info** repository. After a few moments you are returned to the **Repositories** page. You can now sign out of Quay.io.



Finish

On the **workstation** VM, run the **lab expose-image finish** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises. The finish action releases this project and its resources.

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

This concludes the lab.

Summary

In this chapter, you learned:

- OpenShift developers interact with many kinds of registry servers that may or may not require authentication, including the OpenShift internal registry.
- OpenShift requires that developers create secrets to store access tokens to private, external registries.
- The Linux container tools (Skopeo, Podman, and Buildah) can access the OpenShift internal registry, like any other registry server, using the OpenShift user access token, as either a secure or insecure registry.
- OpenShift image streams and image stream tag resources provide stable references to container images and also isolate developers from registry server addresses.

Chapter 4

Building Applications

Goal

Describe the OpenShift build process, trigger and manage builds.

Objectives

- Describe the OpenShift build process.
- Manage application builds using the BuildConfig resource and CLI commands.
- Trigger the build process with supported methods.
- Process post build logic with a post-commit build hook.

Sections

- Describing the OpenShift Build Process (and Quiz)
- Managing Application Builds (and Guided Exercise)
- Triggering Builds (and Guided Exercise)
- Implementing Post-commit Build Hooks (and Guided Exercise)

Lab

Building Applications

Describing the OpenShift Build Process

Objectives

After completing this section, you should be able to describe the OpenShift build process.

The Build Process

The Red Hat OpenShift Container Platform build process transforms either source code or binaries and other input parameters into container images that become available for deployment on the platform. To build a container image, OpenShift requires a **BuildConfig** resource. The configuration for this resource includes one build strategy and one or more input sources such as git, binary or inline definitions.

Build Strategies

The following are the available build strategies in OpenShift:

- Source
- Pipeline
- Docker
- Custom

Each strategy requires a container image.

Source

The **Source** strategy creates a new container image based on application source code or application binaries. OpenShift clones the application source code, or copies the application binaries into a compatible builder image, and assembles a new container image that is ready for deployment on the platform.

This strategy simplifies how developers build container images because it works with the tools that are familiar to them instead of using low-level OS commands such as **yum** in **Dockerfiles**.

OpenShift bases the **Source** strategy upon the source-to-image (S2I) process.

Pipeline

The **JenkinsPipeline** strategy creates a new container image using a Jenkins pipeline plug-in. Although Jenkins builds the container images, OpenShift can start, monitor and manage the build.

The **BuildConfig** resource references a **Jenkinsfile** containing the pipeline workflows. You can define the **Jenkinsfile** directly in the build configuration or pull it from an external Git repository.

OpenShift starts a new Jenkins server to execute the pipeline in the first build using the pipeline strategy. Subsequent Pipeline build configurations in the project share this Jenkins server.

Docker

The **docker** strategy uses the **buildah** command to build a new container image given a **Dockerfile** file. The docker strategy can retrieve the **Dockerfile** and the artifacts to build the container image from a Git repository, or can use a **Dockerfile** provided inline in the build configuration as a build source.

The Docker build runs as a pod inside the OpenShift cluster. Developers do not need to have Docker tooling on their workstation.



Note

Docker builds require elevated privileges and the OpenShift cluster administrator might deny some or all users the right to start Docker builds.

Custom

The **custom** strategy specifies a builder image responsible for the build process. This strategy allows developers to customize the build process. See the references section of this unit to find more information about how to create a custom builder image.

Build Input Sources

A build input source provides source content for builds. OpenShift supports the following six types of input sources, listed in order of precedence:

- **Dockerfile**: Specifies the Dockerfile inline to build an image.
- **Image**: You can provide additional files to the build process when you build from images.
- **Git**: OpenShift clones the input application source code from a Git repository. It is possible to configure the default location inside the repository where the build looks for application source code.
- **Binary**: Allows streaming binary content from a local file system to the builder.
- **Input secrets**: You can use input secrets to allow creating credentials for the build that are not available in the final application image.
- **External artifacts**: Allow copying binary files to the build process.

You can combine multiple inputs in a single build. However, as the inline Dockerfile takes precedence, it overrides any other Dockerfile provided by another input. Additionally, binary input and Git repositories are mutually exclusive inputs.



Note

Although OpenShift offers many strategies and input sources, the most common scenarios are using either the **Source** or **Docker** strategies, with a Git repository as the input source.

BuildConfig Resource

The build configuration defines how the build process happens. The **BuildConfig** resource defines a single build configuration and a set of triggers for when OpenShift must create a new build.

OpenShift generates the **BuildConfig** using the `oc new-app` command. It can also be generated using the **Add to Project** button from the web console or by creating an application from a template.

The following example builds a PHP application using the **Source** strategy and a **Git** input source:

```
{
  "kind": "BuildConfig",
  "apiVersion": "v1",
  "metadata": {
    "name": "php-example", ①
    ...
  },
  "spec": {
    "triggers": [ ②
      {
        "type": "GitHub",
        "github": {
          "secret": "gukAWHzq1On4AJlMjvjS" ③
        }
      },
      ...
    ],
    "runPolicy": "Serial", ④
    "source": { ⑤
      "type": "Git",
      "git": {
        "uri": "http://services.lab.example.com/php-helloworld"
      }
    },
    "strategy": { ⑥
      "type": "Source",
      "sourceStrategy": {
        "from": {
          "kind": "ImageStreamTag",
          "namespace": "openshift",
          "name": "php:7.0"
        }
      }
    },
    "output": { ⑦
      "to": {
        "kind": "ImageStreamTag",
        "name": "php-example:latest"
      }
    },
    ...
  },
  ...
}
```

① Defines a new **BuildConfig** named **php-example**.

② Defines the triggers that start new builds.

- ③ Authorization string for the webhook, randomly generated by OpenShift. External applications send this string as part of the webhook URL to trigger new builds.
- ④ The **runPolicy** attribute defines whether a build can start simultaneously. The **Serial** value represents that it is not possible to build simultaneously.
- ⑤ The **source** attribute is responsible for defining the input source of the build. In this example, it uses a Git repository.
- ⑥ Defines the build strategy to build the final container image. In this example, it uses the **Source** strategy.
- ⑦ The **output** attribute defines where to push the new container image after a successful build.



References

Further information about custom build images and build input sources is available in the *Creating Build Inputs* chapter of the *Builds* guide for Red Hat OpenShift Container Platform 4.5; at
https://access.redhat.com/documentation/en-us/openshift_container_platform/4.5/html/builds/creating-build-inputs

Source versus binary S2I builds are explained in more detail at
<https://developers.redhat.com/blog/2018/09/26/source-versus-binary-s2i-workflows-with-red-hat-openshift-application-runtimes/>

► Quiz

The OpenShift Build Process

Choose the correct answer(s) to the following questions:

► 1. Which three of the following options are valid strategies for building a container image?

(Choose three.)

- a. Docker
- b. Git
- c. Pipeline
- d. Custom

► 2. Which two of the following options are valid types of input sources for building a container image? (Choose two.)

- a. Git
- b. SVN
- c. Filesystem
- d. Dockerfile

- 3. Given the **BuildConfig** in the below exhibit, which three of the following statements are true? (Choose three.)

```
{  
    "kind": "BuildConfig",  
    "apiVersion": "v1",  
    "metadata": {  
        "name": "php-example",  
    },  
    "spec": {  
        ...  
        "runPolicy": "Serial",  
        "source": {  
            "type": "Git",  
            "git": {  
                "uri": "http://services.lab.example.com/php-helloworld"  
            }  
        },  
        "strategy": {  
            "type": "Source",  
            "sourceStrategy": {  
                "from": {  
                    "kind": "ImageStreamTag",  
                    "namespace": "openshift",  
                    "name": "php:7.0"  
                }  
            }  
        },  
        "output": {  
            "to": {  
                "kind": "ImageStreamTag",  
                "name": "php-example:latest"  
            }  
        },  
        ...  
    },  
}
```

- a. Multiple builds can run simultaneously.
- b. After successfully built, the **php-example:latest** container image is available for deployment.
- c. The build uses a Git input source to create the final container image.
- d. The **BuildConfig** uses the **Docker** strategy.
- e. The **BuildConfig** uses the **Source** strategy.

- 4. For the Docker strategy, which of the following is the input source declared in a build configuration generated by the `oc new-app` command?
- a. Dockerfile
 - b. Binary
 - c. Git
 - d. None of these
- 5. A developer wants to build a PHP application using the Source strategy. Which of the following options configure the build configuration to use the S2I builder image for a PHP application?
- a. Use the `from` attribute for the **Source** strategy.
 - b. Use the `from` attribute from the **Image** input source.
 - c. Use the `from` attribute from the **Dockerfile** input source.
 - d. It is not required to configure the S2I builder image. During the build, the source-to-image process will recognize the source language and auto select a PHP builder image.
 - e. None of these.

► Solution

The OpenShift Build Process

Choose the correct answer(s) to the following questions:

- ▶ 1. Which three of the following options are valid strategies for building a container image? (Choose three.)
 - a. Docker
 - b. Git
 - c. Pipeline
 - d. Custom

- ▶ 2. Which two of the following options are valid types of input sources for building a container image? (Choose two.)
 - a. Git
 - b. SVN
 - c. Filesystem
 - d. Dockerfile

► 3. Given the **BuildConfig** in the below exhibit, which three of the following statements are true? (Choose three.)

```
{
  "kind": "BuildConfig",
  "apiVersion": "v1",
  "metadata": {
    "name": "php-example",
  },
  "spec": {
    ...
    "runPolicy": "Serial",
    "source": {
      "type": "Git",
      "git": {
        "uri": "http://services.lab.example.com/php-helloworld"
      }
    },
    "strategy": {
      "type": "Source",
      "sourceStrategy": {
        "from": {
          "kind": "ImageStreamTag",
          "namespace": "openshift",
          "name": "php:7.0"
        }
      }
    },
    "output": {
      "to": {
        "kind": "ImageStreamTag",
        "name": "php-example:latest"
      }
    },
    ...
  },
}
```

- a. Multiple builds can run simultaneously.
- b. After successfully built, the **php-example:latest** container image is available for deployment.
- c. The build uses a Git input source to create the final container image.
- d. The **BuildConfig** uses the **Docker** strategy.
- e. The **BuildConfig** uses the **Source** strategy.

► 4. For the Docker strategy, which of the following is the input source declared in a build configuration generated by the `oc new-app` command?

- a. Dockerfile
- b. Binary
- c. Git
- d. None of these

► 5. A developer wants to build a PHP application using the Source strategy. Which of the following options configure the build configuration to use the S2I builder image for a PHP application?

- a. Use the `from` attribute for the `Source` strategy.
- b. Use the `from` attribute from the `Image` input source.
- c. Use the `from` attribute from the `Dockerfile` input source.
- d. It is not required to configure the S2I builder image. During the build, the source-to-image process will recognize the source language and auto select a PHP builder image.
- e. None of these.

Managing Application Builds

Objectives

After completing this section, you should be able to manage application builds using the build configuration resource and CLI commands.

Creating a Build Configuration

The Red Hat OpenShift Container Platform manages builds through the build configuration resource. There are two ways to create a build configuration:

Using the `oc new-app` command

This command can create a build configuration according to the arguments specified. For example, if a Git repository is defined, then a build configuration using the source strategy is created. Also, if a template is the argument, and the template has a build configuration defined, it creates a build configuration based on template parameters.

Using a custom build configuration

Create a custom build configuration using **YAML** or **JSON** syntax and import it to OpenShift using the `oc create -f` command.

Managing Application Builds Using CLI

OpenShift provides several options allowing developers to manage application builds and build configurations using the CLI. These commands are used to manage the lifecycle of an application, especially during development. Note that build configuration is a separate phase when compared to deployment. A successful build in OpenShift results in a new container image, which triggers a new deployment.

`oc start-build`

Starts a new build manually. The build configuration resource name is the only required argument to start a new build.

```
[user@host ~]$ oc start-build name
```

A successful build creates a new container image in the output ImageStreamTag. If a deployment configuration defines a trigger on that ImageStreamTag, the deployment process starts.

`oc cancel-build`

Cancels a build. If a build started with a wrong version of the application, for example, and the application cannot be deployed, you may want to cancel the build before it fails.

It is only possible to cancel builds that are in running or pending state. Canceling a build means that the build pod terminates, so there is no new container image to push. A deployment is therefore not triggered.

Provide the build configuration resource name to cancel a build:

```
[user@host ~]$ oc cancel-build name
```

oc delete

Deletes a build configuration. Generally, you delete a build configuration when you need to import a new one from a file.

```
[user@host ~]$ oc delete bc/name
```

This command deletes a build (not a build configuration) to reclaim the space used by the build. A build configuration can have multiple builds. Provide the build name to delete a build:

```
[user@host ~]$ oc delete build/name-1
```

oc describe

Describes details about a build configuration resource and the associated builds, giving you information such as labels, the strategy, webhooks, and so on.

```
[user@host ~]$ oc describe bc name
```

Describe a build providing the build name:

```
[user@host ~]$ oc describe build name-1
```

oc logs

Shows the build logs. You can check if your application is building correctly. This command can also display logs from a finished build. It is not possible to check logs from deleted or pruned builds. There are two ways to display a build log:

- Display the build logs from the most recent build.

```
[user@host ~]$ oc logs -f bc/name
```

The **-f** option follows the log until you terminate the command with **Ctrl+C**.

- Display the build logs from a specific build:

```
[user@host ~]$ oc logs build/name-1
```

Pruning Builds

By default, builds that have completed are persisted indefinitely. You can limit the number of previous builds that are retained using the **successfulBuildsHistoryLimit** attribute, and the **failedBuildsHistoryLimit** attribute, as shown in the following YAML snippet of a build configuration:

```
apiVersion: "v1"
kind: "BuildConfig"
metadata:
  name: "sample-build"
spec:
  successfulBuildsHistoryLimit: 2
  failedBuildsHistoryLimit: 2
```

Failed builds include builds with a status of **failed**, **canceled**, or **error**. Builds are sorted by their creation time with the oldest builds being pruned first.



Note

OpenShift administrators can manually prune builds using the **oc adm** object pruning command.

Log Verbosity

OpenShift provides two different mechanisms to configure build log verbosity. The first one is global, and an administrator can define the build log verbosity configuration for the whole cluster. The second one affects only the build log verbosity of the builds from a specific build configuration. This means developers maintain the ability to override the global configuration for a more specific build configuration when they require a different level of log verbosity.

Edit the build configuration resource and add the **BUILD_LOGLEVEL** environment variable as part of the source strategy or Docker strategy to configure a specific log level:

```
[user@host ~]$ oc set env bc/name BUILD_LOGLEVEL="4"
```

The value must be a number between zero and five. Zero is the default value and displays fewer logs than five. When you increase the number, the verbosity of logging messages is greater, and the logs contain more details.

Pruning Builds

To "prune" a build means to delete a completed or failed build. OpenShift can do it automatically according to the configuration, or you can manually prune a build with either the **oc delete** or the **oc adm prune** commands.

Administrators can prune builds and free up disk space to avoid accumulation in the **etcd** data store. There are some configurations to prune builds like pruning orphans, pruning based on the number of successful builds, pruning based on the number of failed builds, and more.

Sometimes you may need to ask administrators to change the pruning configuration so that build logs are retained longer than the default so that you can troubleshoot build issues. These topics are out of scope for the current course.



References

Further information about managing application builds is available in the *Performing Basic Builds* chapter of the *Builds* guide for Red Hat OpenShift Container Platform 4.5; at
https://access.redhat.com/documentation/en-us/openshift_container_platform/4.5/html/builds/basic-build-operations

► Guided Exercise

Managing Application Builds

In this exercise, you will manage application builds with OpenShift, using the source strategy with a Git input source.

Outcomes

You should be able to use the CLI to manage builds on OpenShift.

Before You Begin

To perform this exercise, you need to ensure you have access to:

- A running OpenShift cluster.
- The OpenJDK 1.8 S2I builder image and image stream (**redhat-openjdk18-openshift:1.5**).
- The sample application in the Git repository (**java-serverhost**).

Run the following command on the **workstation** VM to validate the prerequisites and download files required to complete this exercise:

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

► 1. Inspect the Java source code for the sample application.

- 1.1. Enter your local clone of the **D0288-apps** Git repository and checkout the **master** branch of the course's repository to ensure you start this exercise from a known good state:

```
[student@workstation ~]$ cd D0288-apps
[student@workstation D0288-apps]$ git checkout master
...output omitted...
```

- 1.2. Create a new branch to save any changes you make during this exercise:

```
[student@workstation D0288-apps]$ git checkout -b manage-builds
Switched to a new branch 'manage-builds'
[student@workstation D0288-apps]$ git push -u origin manage-builds
...output omitted...
* [new branch]      manage-builds -> manage-builds
Branch manage-builds set up to track remote branch manage-builds from origin.
```

- 1.3. Review the Java source code for the application, inside the the **java-serverhost** folder.
Open the **ServerHostEndPoint.java** file in the **/home/student/D0288-apps/java-serverhost/src/main/java/com/redhat/training/example/javaserverhost/rest** folder:

```

package com.redhat.training.example.javaserverhost.rest;

import javax.ws.rs.Path;
import javax.ws.rs.core.Response;
import javax.ws.rs.GET;
import javax.ws.rs.Produces;
import java.net.InetAddress;

@Path("/")
public class ServerHostEndPoint {

    @GET
    @Produces("text/plain")
    public Response doGet() {
        String host = "";
        try {
            host = InetAddress.getLocalHost().getHostName();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        String msg = "I am running on server "+host+" Version 1.0 \n";
        return Response.ok(msg).build();
    }
}

```

The application implements a simple REST service which returns the hostname of the server it is running on.

► 2. Create a new project.

- 2.1. Load your classroom environment configuration.

Run the following command to load the environment variables created in the first guided exercise:

```
[student@workstation D0288-apps]$ source /usr/local/etc/ocp4.config
```

- 2.2. Log in to OpenShift using your developer user:

```
[student@workstation D0288-apps]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
```

- 2.3. Create a new project to host the application:

```
[student@workstation D0288-apps]$ oc new-project \
> ${RHT_OCP4_DEV_USER}-manage-builds
```

► 3. Create a new application.

- 3.1. Create a new application from sources in Git. Use the branch you created in a previous step. Name the application **jhost** and use the **--build-env** option from

the **oc new-app** command to define a build environment variable with the maven repository location.

The complete command can be either copied or executed directly from the **oc-new-app.sh** script in the **/home/student/D0288/labs/manage-builds** folder:

```
[student@workstation D0288-apps]$ oc new-app --as-deployment-config --name jhost \
> --build-env MAVEN_MIRROR_URL=http://${RHT_OCP4_NEXUS_SERVER}/repository/java \
> -i redhat-openjdk18-openshift:1.5 \
> https://github.com/${RHT_OCP4_GITHUB_USER}/D0288-apps#manage-builds \
> --context-dir java-serverhost
...output omitted...
imagestream.image.openshift.io "jhost" created
buildconfig.build.openshift.io "jhost" created
deploymentconfig.apps.openshift.io "jhost" created
service "jhost" created
...output omitted...
```

Notice there is no space before or after the equals sign (=) after **MAVEN_MIRROR_URL**. The complete key=value pair for the build environment variable is too long for the paper width.

3.2. Check the build logs from the **jhost** build using the **oc logs** command:

```
[student@workstation D0288-apps]$ oc logs -f bc/jhost
...output omitted...
Writing manifest to image destination
Storing signatures
...output omitted...
Push successful
```

3.3. Wait for the application to be ready and running:

```
[student@workstation D0288-apps]$ oc get pods
NAME          READY   STATUS    RESTARTS   AGE
jhost-1-build  0/1     Completed  0          32m
jhost-1-deploy 0/1     Completed  0          30m
jhost-1-10mbp  1/1     Running   0          30m
```

3.4. Expose the application to external access:

```
[student@workstation D0288-apps]$ oc expose svc/jhost
route.route.openshift.io/jhost exposed
```

3.5. Get the hostname generated by OpenShift for the new route:

```
[student@workstation D0288-apps]$ oc get route
NAME      HOST/PORT           ...
jhost     jhost-youruser-manage-builds.apps.cluster.domain.example.com ...
```

3.6. Test the application:

```
[student@workstation D0288-apps]$ curl \
> http://jhost-${RHT_OCP4_DEV_USER}-manage-builds.${RHT_OCP4_WILDCARD_DOMAIN}
I am running on server jhost-1-10mbp Version 1.0
```

**Note**

Even if the application container is ready, the application may take some seconds before being able to respond. If the previous command returns an HTML page with the text **Application is not available**, wait a couple of seconds and try again.

▶ 4. List build configurations and builds.

- 4.1. List all build configurations in the project by running the **oc get bc** command:

```
[student@workstation D0288-apps]$ oc get bc
NAME      TYPE      FROM          LATEST
jhost     Source    Git@manage-builds  1
```

A build configuration resource named **jhost** was created by the **oc new-app** command using the **Source** strategy with a **Git** input source.

- 4.2. The build configuration created by the **oc new-app** command started a build. List all builds available in the project:

```
[student@workstation D0288-apps]$ oc get builds
NAME      TYPE      FROM          STATUS      STARTED           DURATION
jhost-1   Source    Git@cb73a3d  Complete   18 minutes ago  2m4s
```

▶ 5. Update the application to version 2.0.

- 5.1. Edit the **ServerHostEndPoint.java** file in the **/home/student/D0288-apps/java-serverhost/src/main/java/com/redhat/training/example/javaserverhost/rest** folder and update to version **2.0**:

```
String msg = "I am running on server "+host+" Version 2.0 \n";
```

- 5.2. Commit the changes to the Git server.

```
[student@workstation D0288-apps]$ cd java-serverhost
[student@workstation java-serverhost]$ git commit -a -m 'Update the version'
...output omitted...
1 file changed, 1 insertion(+), 1 deletion(-)
```

- 5.3. Start a new build with the **oc start-build** command:

```
[student@workstation java-serverhost]$ oc start-build bc/jhost
build.build.openshift.io/jhost-2 started
```

- 5.4. The application being built is still the older version, without the changes that you committed to your local Git repository. You need to push them before you start the OpenShift build. Cancel the build to avoid a new deployment using the older version.

```
[student@workstation java-serverhost]$ oc cancel-build bc/jhost
build.build.openshift.io/jhost-2 marked for cancellation, waiting to be cancelled
build.build.openshift.io/jhost-2 cancelled
```

- 5.5. Check that the build was canceled:

```
[student@workstation java-serverhost]$ oc get builds
NAME      TYPE      FROM      STATUS      ...
jhost-1   Source    Git@cb73a3d Complete   ...
jhost-2   Source    Git@cb73a3d Cancelled (CancelledBuild) ...
```

- 5.6. Push the updated source code to the Git server:

```
[student@workstation java-serverhost]$ git push
...output omitted...
2aa4b3a..f70977b manage-builds -> manage-builds
[student@workstation java-serverhost]$ cd ~
```

- 5.7. Start a new build to deploy the updated version of the application:

```
[student@workstation ~]$ oc start-build bc/jhost
build.build.openshift.io/jhost-3 started
```

- 5.8. List all builds:

```
[student@workstation ~]$ oc get builds
NAME      TYPE      FROM      STATUS      ...
jhost-1   Source    Git@cb73a3d Complete   ...
jhost-2   Source    Git@cb73a3d Cancelled (CancelledBuild) ...
jhost-3   Source    Git        Running    ...
```

Observe that three builds are available. The first one was created by the **oc new-app** command, the second was canceled by you, and the third features the updated application.

- 5.9. Follow the application build logs:

```
[student@workstation ~]$ oc logs -f build/jhost-3
...output omitted...
Push successful
```

- 5.10. Wait for the application to be ready and running:

```
[student@workstation ~]$ oc get pods
NAME        READY   STATUS    RESTARTS   AGE
jhost-1-build  0/1     Completed  0          53m
jhost-1-deploy 0/1     Completed  0          51m
jhost-2-29zh5  1/1     Running   0          4m40s
jhost-2-deploy 0/1     Completed  0          4m49s
jhost-3-build  0/1     Completed  0          6m21s
```

5.11. Test the updated application:

```
[student@workstation ~]$ curl \
> http://jhost-${RHT_OCP4_DEV_USER}-manage-builds.${RHT_OCP4_WILDCARD_DOMAIN}
I am running on server jhost-2-10mbp Version 2.0
```

► 6. Clean up and delete the project:

```
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-manage-builds
project.project.openshift.io "youruser-manage-builds" deleted
```

Finish

On **workstation**, run the **lab manage-builds finish** script to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises. The finish action releases this project and its resources.

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

This concludes the guided exercise.

Triggering Builds

Objectives

After completing this section, you should be able to trigger the build process with supported methods.

Defining Build Triggers

In OpenShift you can define build triggers to allow the platform to start new builds automatically based on certain events. You can use these build triggers to keep your application containers updated with any new container images or code changes that affect your application. OpenShift defines two kinds of build triggers:

Image change triggers

An image change trigger rebuilds an application container image to incorporate changes made by its parent image.

Webhook triggers

OpenShift webhook triggers are HTTP API endpoints that start a new build. Use a webhook to integrate OpenShift with your Version Control System (VCS), such as Github or BitBucket, to trigger new builds on code changes.

Image change triggers free a developer from watching for changes in an application parent image. Image change triggers can be automatically activated by the OpenShift internal registry when it detects a new image. If the image is from an external registry, you must periodically run the **oc import-image** command to verify whether the container image changed in the registry server in order to stay up-to-date.

The **oc new-app** command automatically creates image change triggers for applications using either the source or Docker build strategies:

- For the source strategy, the parent image is the S2I builder image for the application programming language.
- For the Docker strategy, the parent image is the image referenced by the **FROM** instruction in the application Dockerfile.

To view the triggers associated with a build configuration use the **oc describe bc** command, as shown in the following example:

```
[user@host ~]$ oc describe bc/name
```

To add an image change trigger to a build configuration, use the **oc set triggers** command:

```
[user@host ~]$ oc set triggers bc/name --from-image=project/image:tag
```

A single build configuration cannot include multiple image change triggers.

To remove an image change trigger from a build configuration, use the **oc set triggers** command with the **--remove** option:

```
[user@host ~]$ oc set triggers bc/name --from-image=project/image:tag --remove
```

Use the **oc set triggers --help** command to see the options used to add and remove a configuration change trigger.

Starting New Builds with Webhook Triggers

OpenShift webhook triggers are HTTP API endpoints that start new builds. Use a webhook to integrate OpenShift with a Version Control System (VCS), such as Git, in a way that changes to the application's source code trigger a new build in OpenShift with the latest code.

Software does not have to be a VCS to use these API endpoints, but OpenShift builds can only fetch source code from a Git server.

Red Hat OpenShift Container Platform provides specialized webhook types that support API endpoints compatible with the following VCS services:

- GitLab
- GitHub
- Bitbucket

Red Hat OpenShift Container Platform also provides a generic webhook type that takes a payload defined by OpenShift. A generic webhook can be used by any software to start an OpenShift build. See the product documentation references at the end of this section for the syntax of the generic webhook payload and the HTTP API requests for each type of webhook.

For all webhooks, you must define a secret with a key named **WebHookSecretKey** and the value being the value to be supplied when invoking the webhook. The webhook definition must then reference the secret. The secret ensures the uniqueness of the URL, preventing others from triggering the build. The value of the key will be compared to the secret provided during the webhook invocation. Any time you create a trigger, or OpenShift creates one automatically, a secret is also created by default.

The **oc new-app** command creates a generic and a Git webhook. To add other types of webhook to a build configuration, use the **oc set triggers** command. For example, to add a GitLab webhook to a build configuration:

```
[user@host ~]$ oc set triggers bc/name --from-gitlab
```

If the build configuration already includes a GitLab webhook, the previous command resets the authentication secret embedded in the URL. You must update your GitLab projects to use the new webhook URL.

To remove an existing webhook from a build configuration, use the **oc set triggers** command with the **--remove** option. For example, to remove a GitLab webhook from a build configuration:

```
[user@host ~]$ oc set triggers bc/name --from-gitlab --remove
```

The **oc set triggers bc** command also supports **--from-github** and **--from-bitbucket** options to create triggers specific to each VCS platform.

To retrieve a webhook URL and the secret, use the **oc describe** command and look for the specific type of webhook you need.



References

Further information about build triggers is available in the *Triggering and Modifying Builds* chapter of the *Builds* guide for Red Hat OpenShift Container Platform 4.5; at https://access.redhat.com/documentation/en-us/openshift_container_platform/4.5/html/builds/triggering-builds-build-hooks

► Guided Exercise

Triggering Builds

In this exercise, you will trigger a new build of an application in OpenShift to incorporate updates made to the application S2I builder image.

Outcomes

You should be able to:

- Create an application that relies on an S2I builder image.
- Push a new version of the S2I builder image.
- Update metadata inside an image stream to react to the image update.
- Verify that the application is rebuilt to use the new S2I builder image.

Before You Begin

To perform this exercise, you need to ensure you have access to:

- A running OpenShift cluster.
- The original version of the PHP S2I builder image (**php-70-rhel7-original.tar.gz**)
- The new version of the PHP S2I builder image (**php-70-rhel7-newer.tar.gz**)
- The sample application in the Git repository (**trigger-builds**).

Run the following command on the **workstation** VM to validate the prerequisites and download files required to complete this exercise:

```
[student@workstation ~]$ lab trigger-builds start
```

Steps

► 1. Prepare the lab environment.

- 1.1. Load your classroom environment configuration.

Run the following command to load the environment variables created in the first guided exercise:

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

- 1.2. Log in to OpenShift using your developer user:

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
```

- 1.3. Create a new project for the application:

```
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-trigger-builds
```

- 2. Add a image stream to the project to be used with the new application.

- 2.1. Log in into your personal Quay.io account using Podman.

```
[student@workstation ~]$ podman login -u ${RHT_OCP4_QUAY_USER} quay.io
Password:
Login Succeeded!
```

- 2.2. Push the original PHP 7.0 builder image to your Quay.io public registry.

The commands you need to push the image are available in the **push-original.sh** script in the **/home/student/D0288/labs/trigger-builds** directory. You can either copy and paste the commands in the script, or you can execute the script.

```
[student@workstation ~]$ cd /home/student/D0288/labs/trigger-builds
[student@workstation trigger-builds]$ skopeo copy \
> docker-archive:php-70-rhel7-original.tar.gz \
> docker://quay.io/${RHT_OCP4_QUAY_USER}/php-70-rhel7:latest
Getting image source signatures
...output omitted...
Writing manifest to image destination
Storing signatures
```

- 2.3. The Quay.io registry defaults to private images, so you will have to add a secret to a the builder service account in order to access it.

Create a secret from the container registry API access token that was stored by Podman.

```
[student@workstation trigger-builds]$ oc create secret generic quay-registry \
> --from-file .dockerconfigjson=${XDG_RUNTIME_DIR}/containers/auth.json \
> --type kubernetes.io/dockerconfigjson
secret/quay-registry created
```

Add the Quay.io registry secret to the builder service account.

```
[student@workstation trigger-builds]$ oc secrets link builder quay-registry
```

- 2.4. Update the **php** image stream to fetch the metadata for the new container image. The external registry uses the *docker-distribution* package and does not notify OpenShift about image changes.

```
[student@workstation trigger-builds]$ oc import-image php \
> --from quay.io/${RHT_OCP4_QUAY_USER}/php-70-rhel7 --confirm
imagestream.image.openshift.io/php-70-rhel7 imported

Name:          php
...output omitted...
latest
tagged from quay.io/youruser/php-70-rhel7
```

```
* quay.io/youruser/php-70-rhel7@sha256:c919...8cb2
  Less than a second ago
...output omitted...
```

► 3. Create a new application.

- 3.1. Create a new application from sources in Git. Name the application as **trigger**.

The complete command can be either copied or executed directly from the **oc-new-app.sh** script in the **/home/student/D0288/labs/trigger-builds** folder:

```
[student@workstation trigger-builds]$ oc new-app --as-deployment-config \
> --name trigger \
> php-http://github.com/${RHT_OCP4_GITHUB_USER}/D0288-apps \
> --context-dir trigger-builds
--> Found image ... "youruser-trigger-builds/php" ... for "php"

...output omitted...

--> Success
Build scheduled, use 'oc logs -f bc/trigger' to track its progress.
...output omitted...
```

- 3.2. Wait until the build finishes.

```
[student@workstation trigger-builds]$ oc logs -f bc/trigger
...output omitted...
Push successful
```

- 3.3. Wait for the application to be ready and running:

```
[student@workstation trigger-builds]$ oc get pods
NAME        READY   STATUS    RESTARTS   AGE
trigger-1-build  0/1     Completed   0          10m
trigger-1-deploy 0/1     Completed   0          9m10s
trigger-1-q6bln  1/1     Running    0          9m2s
```

- 3.4. Verify that an image change trigger is defined as part of the build configuration:

```
[student@workstation trigger-builds]$ oc describe bc/trigger | grep Triggered
Triggered by: Config, ImageChange
```

The last trigger, the **ImageChange** trigger, starts a new build if the image stream detects that its base image has changed.

► 4. Update the image stream to start a new build.

- 4.1. Upload the new version of the PHP S2I builder image to the Quay.io registry

The commands you need to push the new image are available in the **push-newer.sh** script in the **/home/student/D0288/labs/trigger-builds** directory. You can either copy and paste the commands in the script, or you can execute the script.

```
[student@workstation trigger-builds]$ skopeo copy \
> docker-archive:php-70-rhel7-newer.tar.gz \
> docker://quay.io/${RHT_OCP4_QUAY_USER}/php-70-rhel7:latest
Getting image source signatures
...output omitted...
Writing manifest to image destination
Storing signatures
```

4.2. Update the **php** image stream to fetch the metadata for the new container image.

```
[student@workstation trigger-builds]$ oc import-image php
imagestream.image.openshift.io/php-70-rhel7 imported

Name:          php
...output omitted...
latest
tagged from quay.io/youruser/php-70-rhel7

* quay.io/youruser/php-70-rhel7@sha256:3f5e...6ab7
  Less than a second ago
quay.io/youruser/php-70-rhel7@sha256:c919...8cb2
  2 minutes ago
...output omitted...
```

► 5. Check the new build.

5.1. The image stream update triggered a new build. List all builds to verify that a second build started:

```
[student@workstation trigger-builds]$ oc get builds
NAME      TYPE      FROM      STATUS      STARTED      DURATION
trigger-1  Source    Git@da010df  Complete   13 minutes ago  58s
trigger-2  Source    Git@da010df  Complete   28 seconds ago  15s
```

5.2. Confirm that the **trigger-2** build was started by an image stream update:

```
[student@workstation trigger-builds]$ oc describe build trigger-2 | grep cause
Build trigger cause: Image change
```

► 6. Finish.

6.1. Change to the home directory.

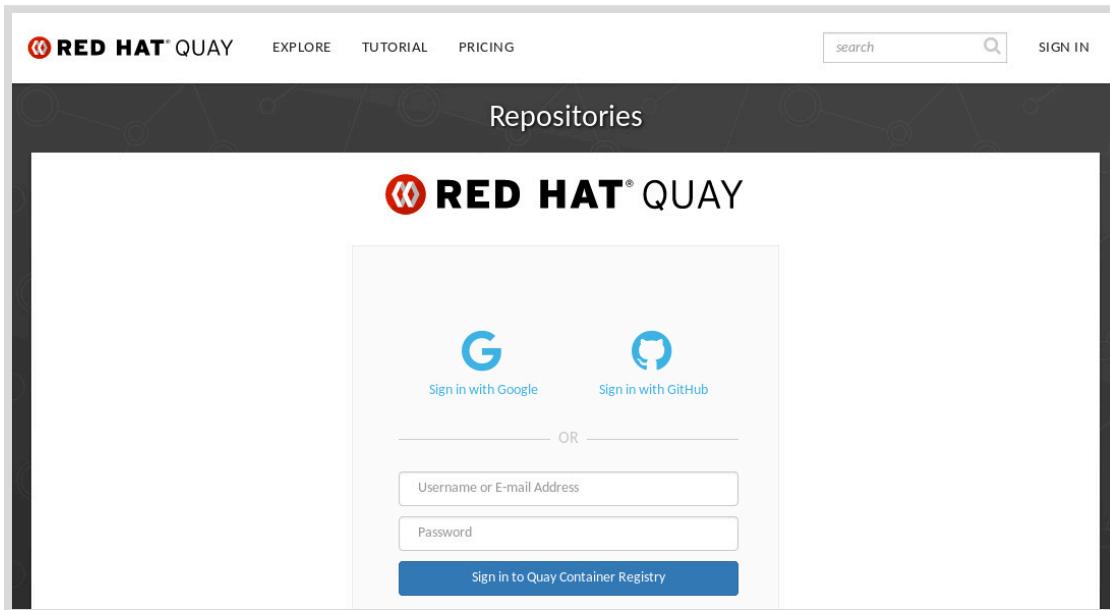
```
[student@workstation trigger-builds]$ cd ~
[student@workstation ~]$
```

6.2. Delete the container image from the external registry:

```
[student@workstation ~]$ skopeo delete \
> docker://quay.io/${RHT_OCP4_QUAY_USER}/php-70-rhel7
```

- 6.3. Log in on Quay.io using your personal free account.

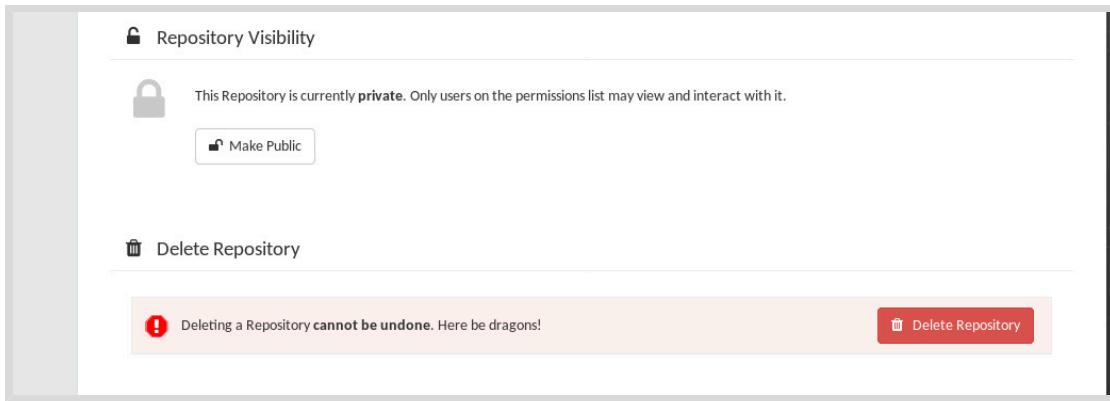
Visit <http://quay.io> and click **Sign In** to provide your user credentials. Click **Sign in to Quay Container Registry** to log in on Quay.io.



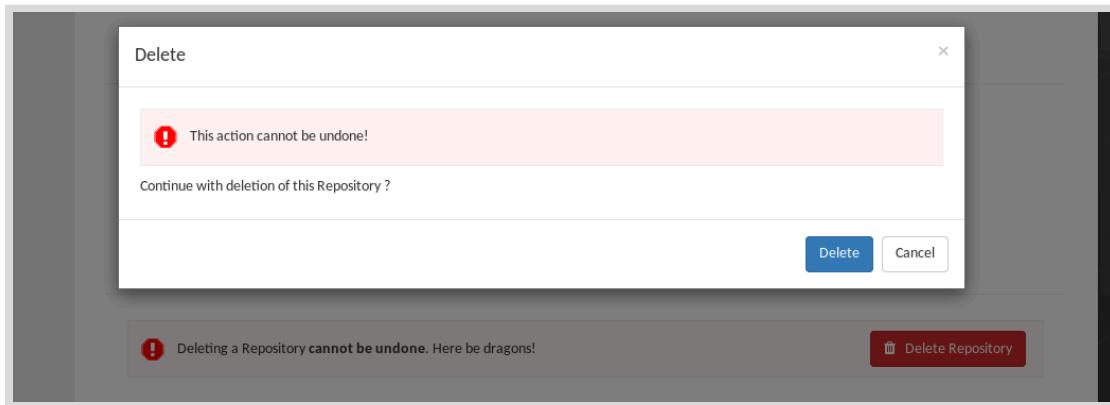
- 6.4. On Quay.io's top-level menu, click **Repositories** and look for **php-70-rhel7**. The lock icon next to it indicates it is a private repository, that requires authentication for both pulls and pushes. Click **php-70-rhel7** to enter the **Repository Activity** page.

REPOSITORY NAME	LAST MODIFIED	ACTIVITY	STAR
yourquayuser / php-70-rhel7	(Empty Repository)		
yourquayuser / do180-custom-https	10/25/2019		
yourquayuser / nexus	10/16/2019		
yourquayuser / do180-mysql-57-rhel7	11/01/2019		
yourquayuser / do180-todonodejs	10/31/2019		
yourquayuser / do180-quote-php	11/02/2019		

- 6.5. On the **Repository Activity** page for the **php-70-rhel7** repository, scroll down until you see the gear icon and click it to enter the **Settings tab**. Scroll down until you find the **Delete Repository** button and click it.



- 6.6. On the **Delete** pop-up window, click **Delete** to confirm you want to delete the **php-70-rhel7** repository. After a few moments you will be back to your **Repositories** page. You can now sign out of Quay.io.



Finish

On **workstation**, run the **lab trigger-builds finish** script to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises. The **finish** action releases this project and its resources.

```
[student@workstation ~]$ lab trigger-builds finish
```

This concludes the guided exercise.

Implementing Post-commit Build Hooks

Objectives

After completing this section, you should be able to process post build logic with a post-commit build hook.

Describing Post-commit Build Hooks

OpenShift provides the **post-commit** build hook functionality to perform validation tasks during builds. The **post-commit** build hook runs commands in a temporary container before pushing the new container image generated by the build to the registry. The hook starts a temporary container using the container image created by the build.

Depending upon the exit code from the command execution in the temporary container, OpenShift either will or will not push the image to the registry. If the command returns a non-zero exit code, indicating failure, OpenShift does not push the image and marks the build as failed. If the command succeeds, OpenShift pushes the image to the registry.

You can verify that a build failed because of a post-commit hook by examining the build logs using the **oc logs** command:

```
[user@host ~]$ oc logs bc/name
...
Running post commit hook ...
...
error: build error: container "openshift_s2i-build_hook-2_post-commit_post-
commit_45ec0816" returned non-zero exit code: 35
```

Reviewing Use Cases for Post-commit Build Hooks

A typical scenario to use a post-commit build hook is to execute some tests in your application. This way, before OpenShift pushes the image to the registry and starts a new deployment, tests can check if the application is working correctly. If the test fails, the build fails and does not continue with a deployment.

There are a few other common use cases where it can be useful to leverage a post-commit build hook. For example:

- To integrate a build with an external application through an HTTP API.
- To validate a non-functional requirement such as application performance, security, usability, or compatibility.
- To send an email to the developer's team informing them of the new build.

Configuring a Post-commit Build Hook

There are two types of post-commit build hooks you can configure:

Command

A command is executed using the **exec** system call. Create a command post-commit build hook using the **--command** option as shown in the following command:

```
[user@host ~]$ oc set build-hook bc/name --post-commit \
> --command -- bundle exec rake test --verbose
```



Note

The space right after the **--** argument is not a mistake. The **--** argument separates the OpenShift command line that configures a post-commit hook from the command executed by the hook.

Shell script

Runs a build hook with the **/bin/sh -ic** command. This is more convenient since it has all the features provided by a shell, such as argument expansion, redirection, and so on. It only works if the base image has the **sh** shell. Create a shell script post-commit build hook using the **--script** as shown in the following command:

```
[user@host ~]$ oc set build-hook bc/name --post-commit \
> --script="curl http://api.com/user/${USER}"
```



References

Further information about post-commit build hooks is available in the *Triggering and Modifying Builds* chapter of the *Builds* guide for Red Hat OpenShift Container Platform 4.5; at
https://access.redhat.com/documentation/en-us/openshift_container_platform/4.5/html/builds/triggering-builds-build-hooks

► Guided Exercise

Implementing Post-Commit Build Hooks

In this exercise, you will set a post-commit build hook to integrate a build with an external application.

Outcomes

You should be able to add a post-commit build hook to a build configuration resource. The post-commit build hook sends an HTTP API request to the **builds-for-managers** application.

Before You Begin

The **builds-for-managers** application is used by managers to track application builds made by a team of developers. The application displays information like the project, the Git URL, and the developer who started the build. You need to integrate your builds with the application so managers are aware of your work.

To perform this exercise, you need to ensure you have access to:

- A running OpenShift cluster.
- The PHP S2I builder image (**php-73-rhel7:7.3**).
- The **builds-for-managers** container image (**quay.io/redhattraining/builds-for-managers**)
- The application Git repository (**post-commit**).

Run the following command on the **workstation** VM to validate the prerequisites, create the **post-commit** project, start the **builds-for-managers** application, and download files required to complete this exercise:

```
[student@workstation ~]$ lab post-commit start
```

► 1. Prepare the lab environment.

1.1. Load your classroom environment configuration.

Run the following command to load the environment variables created in the first guided exercise:

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

1.2. Log in to the OpenShift cluster.

```
[student@workstation D0288-apps]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful
...output omitted...
```

13. Ensure you are on the **youruser-post-commit** project:

```
[student@workstation ~]$ oc project ${RHT_OCP4_DEV_USER}-post-commit
Already on project "youruser-post-commit" on server "https://...".
```

Because the **lab post-commit start** command creates this project, you should be on this project already. If not, your output may differ from above.

14. Verify that the **builds-for-managers** is running in the **youruser-post-commit** project:

```
[student@workstation ~]$ oc status
In project youruser-post-commit on server https://api.cluster.domain....

http://builds-for-managers... to pod port 8080-tcp (svc/builds-for-managers)
  dc/builds-for-managers deploys istag/builds-for-managers:latest
    deployment #1 deployed 5 minutes ago - 1 pod

...output omitted...
```

▶ 2. Create a new application.

21. Create a new application from sources in Git. Name the application as **hook** and prepend the **php:7.3** image stream to the Git repository URL using a tilde (~).

The complete command can be either copied or executed directly from the **oc-new-app.sh** script in the **/home/student/D0288/labs/post-commit** folder:

```
[student@workstation ~]$ oc new-app --as-deployment-config --name hook \
> php:7.3~http://github.com/${RHT_OCP4_GITHUB_USER}/D0288-apps \
> --context-dir post-commit
```

22. Wait for the build to finish:

```
[student@workstation ~]$ oc logs -f bc/hook
...output omitted...
Push successful
```

23. Wait for the application to be ready and running:

```
[student@workstation ~]$ oc get pods
NAME                  READY   STATUS    RESTARTS   AGE
builds-for-managers-1-deploy  0/1     Completed   0          7m
builds-for-managers-1-w1s8f   1/1     Running    0          8m
hook-1-build             0/1     Completed   0          1m
hook-1-deploy            0/1     Completed   0          3m
hook-1-lmn12              1/1     Running    0          11s
```

There are two different applications currently running. The first one is the **builds-for-managers** application used by managers and the second one is your PHP application.

▶ 3. Integrate the PHP application build with the **builds-for-managers** application.

- 3.1. Inspect the **create-hook.sh** script provided in the **/home/student/D0288/labs/post-commit** folder. It creates a build hook that integrates your PHP application builds with the **builds-for-managers** application using the **curl** command.

```
[student@workstation ~]$ cat ~/D0288/labs/post-commit/create-hook.sh
...output omitted...
oc set build-hook bc/hook --post-commit --command -- \
    bash -c "curl -s -S -i -X POST http://builds-for-managers-
${RHT_OCP4_DEV_USER}-post-commit.${RHT_OCP4_WILDCARD_DOMAIN}/api/builds
-f -d 'developer=\${DEVELOPER}&git=\${OPENSHIFT_BUILD_SOURCE}&project=\
\${OPENSHIFT_BUILD_NAMESPACE}'"
```

The **curl** command sends three environment variables to the **builds-for-managers** application:

- **DEVELOPER**: Contains the developer's name.
- **OPENSHIFT_BUILD_SOURCE**: Contains the project Git URL.
- **OPENSHIFT_BUILD_NAMESPACE**: Contains the project's name.

- 3.2. Run the **create-hook.sh** script:

```
[student@workstation ~]$ ~/D0288/labs/post-commit/create-hook.sh
buildconfig.build.openshift.io/hook hooks updated
```

- 3.3. Verify that the post-commit build hook was created:

```
[student@workstation ~]$ oc describe bc/hook | grep Post
Post Commit Hook: ["bash", "-c", "\"curl -s -S -i -X POST http://builds-..."
```

- 3.4. Start a new build using the **oc start-build** command with the **-F** option enabled to display the logs and verify the HTTP API response code. You will see an HTTP 400 status code returned by the **curl** command executed by the post-commit hook:

```
[student@workstation ~]$ oc start-build bc/hook -F
build.build.openshift.io/hook-2 started
...output omitted...
STEP 11: RUN bash -c "curl -s -S -i -X POST http://builds-for-managers-...
curl: (22) The requested URL returned error: 400 Bad Request
subprocess exited with status 22
subprocess exited with status 22
error: build error: error building at STEP "RUN bash -c "curl -s -S -i ..."
```

The **builds-for-managers** application rejected the HTTP API request because the **DEVELOPER** environment variable is not defined.

- 3.5. List the builds and verify that one build failed due to a post-commit hook failure:

```
[student@workstation ~]$ oc get builds
NAME      TYPE      FROM          STATUS   ...output omitted...
hook-1    Source    Git@c2166cc  Complete
hook-2    Source    Git@c2166cc  Failed (GenericBuildFailed)
```

- 4. Fix the missing environment variable problem.

- 4.1. Create the **DEVELOPER** build environment variable using your name with the **oc set env** command:

```
[student@workstation ~]$ oc set env bc/hook DEVELOPER="Your Name"
buildconfig.build.openshift.io/hook updated
```

- 4.2. Verify that the **DEVELOPER** environment variable is available in the **hook** build configuration:

```
[student@workstation ~]$ oc set env bc/hook --list
# buildconfigs hook
DEVELOPER=Your Name
```

- 4.3. Start a new build and display the logs to verify the HTTP API response code. You will see an HTTP 200 status code:

```
[student@workstation ~]$ oc start-build bc/hook -F
build.build.openshift.io/hook-3 started
...output omitted...
STEP 11: RUN bash -c "curl -s -S -i -X POST http://builds-for-managers-...
HTTP/1.1 200 OK
...output omitted...
Push successful
```

The HTTP 200 status code signals that the **builds-for-managers** application accepted the HTTP API request.

- 4.4. Get the **builds-for-managers** exposed route:

```
[student@workstation ~]$ oc get route/builds-for-managers \
> -o jsonpath='{.spec.host}{"\n"}'
builds-for-managers-youruser-post-commit.apps.cluster.domain.example.com
```

- 4.5. Open a web browser to access <http://builds-for-managers-youruser-post-commit.apps.cluster.domain.example.com>. The page displays all the builds and the developer who started each one.

Builds for Managers

Date	Developer	Project	Git Project
2019-07-10 07:08:43	Your Name	youruser-post-commit	http://github.com/youruser/DO288-apps

- 5. Clean up: Delete the **youruser-post-commit** project in OpenShift.

```
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-post-commit
```

Finish

On **workstation**, run the **lab post-commit finish** script to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises. The finish action releases this project and its resources.

```
[student@workstation ~]$ lab post-commit finish
```

This concludes the guided exercise.

► Lab

Building Applications

Performance Checklist

In this lab, you will use OpenShift to manage application builds, troubleshoot an application, and trigger a new build using webhooks.



Note

The grade script at the end of each chapter lab requires that you use the exact project names and other identifiers as given in the lab specification.

Outcomes

You should be able to:

- Troubleshoot an application by managing the lifecycle of a build.
- Trigger a new build using webhooks.

Before You Begin

To perform this exercise, you need access to:

- A running OpenShift cluster.
- The S2I builder image and image stream for Node.js applications (**nodejs**).
- The application in the Git repository (**build-app**).

Run the following command on the **workstation** VM to validate the prerequisites and download files required to complete this exercise:

```
[student@workstation ~]$ lab build-app start
```

Requirements

The provided application is written in JavaScript, using the Node.js runtime. It is a simple application based on the Express framework. You should build and deploy the application to an OpenShift cluster according to the following requirements:

- The project name is **youruser-build-app**.
- The application name is **simple**. Use the **oc-new-app.sh** script in the lab's directory to create and deploy the application. This script contains an intentional error that you fix in a later step. Do not modify or edit the **oc-new-app.sh** script.
- The deployed application is created from the source code in the **build-app** subdirectory of the Git repository:

<https://github.com/youruser/D0288-apps>.

- The NPM modules required to build the application are available from the Nexus server URL at:

`http://${RHT_OCP4_NEXUS_SERVER}/repository/nodejs.`

Use the **npm_config_registry** environment variable to pass this information to the S2I builder image for Node.js.

- The application is accessible from the default route:

`simple-youruser-build-app.apps.cluster.domain.example.com.`

Steps

- Create the `youruser-build-app` project.
- Execute the `oc-new-app.sh` script in the `/home/student/D0288/labs/build-app` directory to create the application.



Important

An intentional error exists in the `oc-new-app.sh` script that you fix in a later step.
Do not modify or edit the `oc-new-app.sh` script.

- Verify that the application build fails. Set the correct value for the **npm_config_registry** variable in the application build configuration to fix the problem.
- Expose the application service for external access and obtain the route URL.
- Start a new build and verify that the application is ready and running. Verify that the application is accessible using the route URL you obtained in the previous step.
- Use the generic webhook for the build configuration to start a new application build.
- Grade your work.

Run the following command on the **workstation** VM to verify that all tasks were accomplished:

```
[student@workstation ~]$ lab build-app grade
```

- Clean up and delete the project.

```
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-build-app
```

Finish

On **workstation**, run the `lab build-app finish` script to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises. The `finish` action releases this project and its resources.

```
[student@workstation ~]$ lab build-app finish
```

This concludes the lab.

► Solution

Building Applications

Performance Checklist

In this lab, you will use OpenShift to manage application builds, troubleshoot an application, and trigger a new build using webhooks.



Note

The grade script at the end of each chapter lab requires that you use the exact project names and other identifiers as given in the lab specification.

Outcomes

You should be able to:

- Troubleshoot an application by managing the lifecycle of a build.
- Trigger a new build using webhooks.

Before You Begin

To perform this exercise, you need access to:

- A running OpenShift cluster.
- The S2I builder image and image stream for Node.js applications (**nodejs**).
- The application in the Git repository (**build-app**).

Run the following command on the **workstation** VM to validate the prerequisites and download files required to complete this exercise:

```
[student@workstation ~]$ lab build-app start
```

Requirements

The provided application is written in JavaScript, using the Node.js runtime. It is a simple application based on the Express framework. You should build and deploy the application to an OpenShift cluster according to the following requirements:

- The project name is **youruser-build-app**.
- The application name is **simple**. Use the **oc-new-app.sh** script in the lab's directory to create and deploy the application. This script contains an intentional error that you fix in a later step. Do not modify or edit the **oc-new-app.sh** script.
- The deployed application is created from the source code in the **build-app** subdirectory of the Git repository:

<https://github.com/youruser/D0288-apps>.

- The NPM modules required to build the application are available from the Nexus server URL at:

[http://\\${RHT_OCP4_NEXUS_SERVER}/repository/nodejs](http://${RHT_OCP4_NEXUS_SERVER}/repository/nodejs).

Use the **npm_config_registry** environment variable to pass this information to the S2I builder image for Node.js.

- The application is accessible from the default route:

<simple-youruser-build-app.apps.cluster.domain.example.com>.

Steps

- Create the **youruser-build-app** project.

- Load your classroom environment configuration.

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

- Log in to OpenShift using your developer user account:

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
```

- Create a new project to host the application:

```
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-build-app
```

- Execute the **oc-new-app.sh** script in the **/home/student/D0288/labs/build-app** directory to create the application.



Important

An intentional error exists in the **oc-new-app.sh** script that you fix in a later step.
Do not modify or edit the **oc-new-app.sh** script.

- Review the script that creates the application:

```
[student@workstation ~]$ cat ~/D0288/labs/build-app/oc-new-app.sh
...output omitted...
oc new-app --as-deployment-config --name simple --build-env \
  npm_config_registry=http://invalid-server:8081/nexus/content/groups/nodejs \
  https://github.com/${RHT_OCP4_GITHUB_USER}/D0288-apps \
  --context-dir build-app
```

- Execute the **oc-new-app.sh** script to create the application:

```
[student@workstation ~]$ ~/DO288/labs/build-app/oc-new-app.sh  
...output omitted...  
--> Creating resources ...  
...output omitted...  
--> Success  
...output omitted...
```

3. Verify that the application build fails. Set the correct value for the **npm_config_registry** variable in the application build configuration to fix the problem.

- 3.1. View the build logs to identify the build error. An error message may take some time to appear.

```
[student@workstation ~]$ oc logs -f bc/simple  
...output omitted...  
--> Using 'npm install -s --only=production'  
subprocess exited with status 1  
subprocess exited with status 1  
error: build error: error building at STEP  
...output omitted...
```

- 3.2. An invalid Nexus server URL caused the failure. Confirm that the Nexus server URL is wrong:

```
[student@workstation ~]$ oc set env bc simple --list  
# buildconfigs simple  
npm_config_registry=http://invalid-server:8081/nexus/content/groups/nodejs
```

- 3.3. Fix the variable to use the correct Nexus server URL:

```
[student@workstation ~]$ oc set env bc simple \  
> npm_config_registry=http://${RHT_OCP4_NEXUS_SERVER}/repository/nodejs  
buildconfig.build.openshift.io/simple updated
```

Notice that there is no space before or after the equals sign (=) after **npm_config_registry**.

- 3.4. Verify that the **npm_config_registry** variable has the correct value:

```
[student@workstation ~]$ oc set env bc simple --list  
# buildconfigs simple  
npm_config_registry=http://nexus-common.../repository/nodejs
```

Notice there is no space before or after the equals sign (=) after **npm_config_registry**. The complete key=value pair for the build environment variable is too long for the paper width.

4. Expose the application service for external access and obtain the route URL.

- 4.1. Expose the service:

Chapter 4 | Building Applications

```
[student@workstation ~]$ oc expose svc simple
route.route.openshift.io/simple exposed
```

- 4.2. Obtain the route URL:

```
[student@workstation ~]$ oc get route/simple \
> -o jsonpath='{.spec.host}{"\n"}'
simple-youruser-build-app.apps.cluster.domain.example.com
```

5. Start a new build and verify that the application is ready and running. Verify that the application is accessible using the route URL you obtained in the previous step.

- 5.1. Start a new build and follow the logs:

```
[student@workstation ~]$ oc start-build simple -F
build.build.openshift.io/simple-2 started
...output omitted...
Push successful
```

- 5.2. Wait for the application to be ready and running:

```
[student@workstation ~]$ oc get pods
NAME        READY   STATUS    RESTARTS   AGE
simple-1-build   0/1     Error      0          20m
simple-1-cpfqc   1/1     Running     0          4m5s
simple-1-deploy   0/1     Completed    0          4m13s
simple-2-build   0/1     Completed    0          5m41s
```

- 5.3. Test the application:

```
[student@workstation ~]$ curl \
> simple-${RHT_OCP4_DEV_USER}-build-app.${RHT_OCP4_WILDCARD_DOMAIN}
Simple app for the Building Applications Lab!
```

6. Use the generic webhook for the build configuration to start a new application build.

- 6.1. Get the generic webhook URL that starts a new build, with the **oc describe** command.

```
[student@workstation ~]$ oc describe bc simple
Name: simple
...output omitted...
Webhook Generic:
URL: https://apps.cluster.domain.example.com/apis/build.openshift.io/v1/
namespaces/youruser-build-app/buildconfigs/simple/webhooks/<secret>/generic
```

- 6.2. Get the secret for the webhook by running the **oc get bc** command, and pass the **-o json** option to dump the build config details in JSON.

```
[student@workstation ~]$ oc get bc simple \
> -o jsonpath=".spec.triggers[*].generic.secret\{\'\\n\'\}"
4R8kYYf3014kCSPcECmn
```

Note the secret for the generic webhook. You will need this secret in the next step to trigger a new build.

- 6.3. Start a new build using the webhook URL, and the secret discovered from the output of the previous steps. The error message about 'invalid Content-Type on payload' can be safely ignored.

```
[student@workstation ~]$ curl -X POST -k \
> ${RHT_OCP4_MASTER_API}\
> /apis/build.openshift.io/v1/namespaces/${RHT_OCP4_DEV_USER}-build-app\
> /buildconfigs/simple/webhooks/4R8kYYf3014kCSPcECmn/generic
...output omitted...
"status": "Success",
...output omitted...
```

Notice there is no space before or after the line breaks, the complete URL is too long for the paper width. If this command fails, check that the RHT_OCP4_MASTER_API does not have an ending slash symbol.

- 6.4. List all builds and verify that a new build has started:

```
[student@workstation ~]$ oc get builds
NAME      TYPE      FROM      STATUS      STARTED ...
simple-1  Source    Git@3e14daf  Failed (AssembleFailed)  About an hour ago
simple-2  Source    Git@3e14daf  Complete    20 minutes ago
simple-3  Source    Git@3e14daf  Complete    32 seconds ago
```

- 6.5. Wait for the new build to finish:

```
[student@workstation ~]$ oc logs -f bc/simple
...output omitted...
Push successful
```

- 6.6. Wait for the application to be ready and running:

```
[student@workstation ~]$ oc get pods
NAME      READY  STATUS      RESTARTS  AGE
simple-1-build  0/1   Error      0          29m
simple-1-deploy 0/1   Completed  0          24m
simple-2-build  0/1   Completed  0          26m
simple-2-c7jvq  1/1   Running    0          4m18s
simple-2-deploy 0/1   Completed  0          4m27s
simple-3-build  0/1   Completed  0          5m58s
```

7. Grade your work.

Run the following command on the **workstation** VM to verify that all tasks were accomplished:

```
[student@workstation ~]$ lab build-app grade
```

8. Clean up and delete the project.

```
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-build-app
```

Finish

On **workstation**, run the **lab build-app finish** script to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises. The **finish** action releases this project and its resources.

```
[student@workstation ~]$ lab build-app finish
```

This concludes the lab.

Summary

In this chapter, you learned:

- A **BuildConfig** resource includes one strategy and one or more input sources.
- There are four build strategies: **Source**, **Pipeline**, **Docker**, and **Custom**.
- The six build input sources, in order of precedence are: Dockerfile, Git, image, binary, input secrets, and external artifacts.
- Manage the build lifecycle with **oc** CLI commands such as **oc start-build**, **oc cancel-build**, **oc delete**, **oc describe**, and **oc logs**.
- Builds can start automatically through build triggers such as an image change trigger and webhooks.
- You can perform validation tasks during builds using a **post-commit** build hook.

Chapter 5

Customizing Source-to-Image Builds

Goal

Customize an existing S2I builder image and create a new one.

Objectives

- Describe the required and optional steps in the Source-to-Image build process.
- Customize an existing S2I builder image with scripts.
- Create a new S2I builder image with S2I tools.

Sections

- Describing the Source-to-Image Architecture (and Quiz)
- Customizing an Existing S2I Builder Image (and Guided Exercise)
- Creating an S2I Builder Image (and Guided Exercise)

Lab

Customizing Source-to-Image Builds

Describing the Source-to-Image Architecture

Objectives

After completing this section, you should be able to describe the required and optional steps in a Source-to-Image build.

Source-to-Image (S2I) Language Detection

OpenShift can create applications directly from source code stored in a Git repository. The simpler syntax for the **oc new-app** command requires only the Git repository URL as an argument, and then OpenShift tries to auto-detect the programming language used by the application and select a compatible builder image.

The autodetection logic is not perfect. The **oc new-app** command can use a range of command-line options to force a particular choice. These command-line options were presented earlier in this course.

The programming language detection feature relies on finding specific file names at the root of the Git repository. The following table displays some of the more common options, but it is not an extensive list of all source-to-image compatible languages. Refer to the product documentation to view all the rules for each Red Hat OpenShift Container Platform release:

Files	Language builder	Programming language
Dockerfile	N/A	Dockerfile build (not S2I)
pom.xml	jee	Java (with JBoss EAP)
app.json, package.json	nodejs	Node.js (JavaScript)
composer.json, index.php	php	PHP

OpenShift follows a multistep algorithm to determine if the URL points to a source code repository and if so which builder image should perform the build. The following is a simplified description of the algorithm:

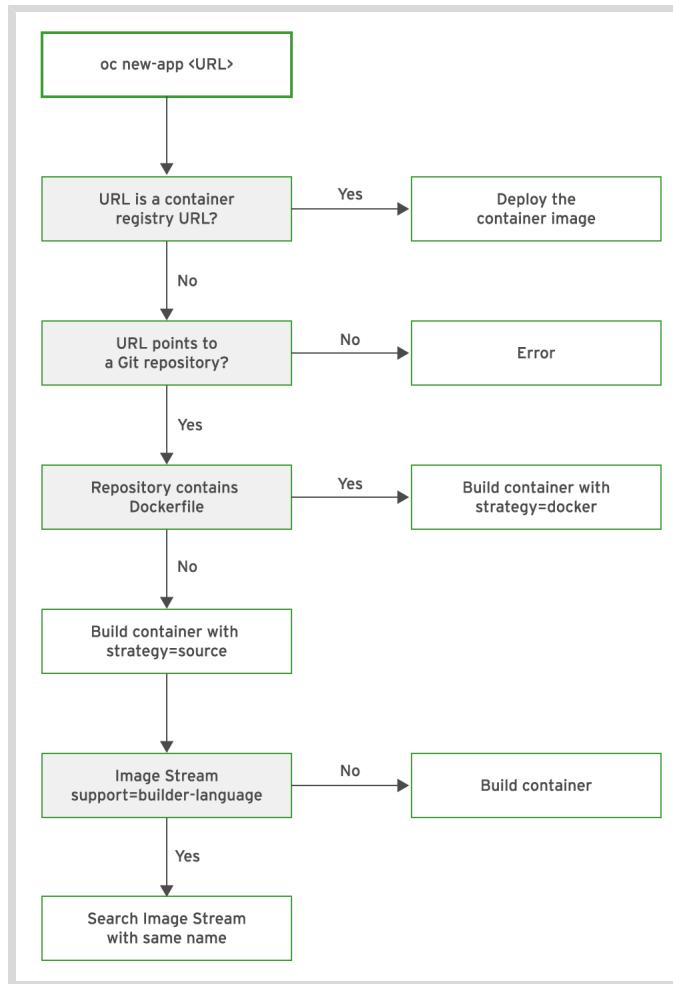


Figure 5.1: S2I build process selection

1. OpenShift accesses the URL as a container registry URL. If this succeeds, there is no need to create a build configuration. OpenShift creates the deployment configuration and other resources required to deploy a container image.
2. If the URL points to a Git repository, OpenShift retrieves a file list from the repository and searches for a **Dockerfile** file. If found, the build configuration uses a **docker** strategy. Otherwise, the build configuration uses a **source** strategy, which needs an S2I builder image.
3. OpenShift searches for image streams that match the language builder name as the value of the **supports** annotation. The first match becomes the S2I builder image.
4. If no annotation matches, OpenShift searches for an image stream whose name matches the language builder name. The first match becomes the S2I builder image.

Steps 3 and 4 make it easy to add new builder images to an OpenShift cluster. It also means that multiple builder images can be potential matches. Earlier in this chapter, the **oc new-app** command-line options were discussed to avoid such ambiguity and to ensure that OpenShift selects the correct image stream as the S2I builder image.

For example, when you run the following command, the **oc** command identifies that you are referring to a registry URL and it starts the container deployment:

```
[user@host ~]$ oc new-app --as-deployment-config registry.access.redhat.com/ubi8/
ubi:8.0
```

Alternatively, when you run the following command, the **oc** command identifies that you are using a Git repository and it is going to clone the repository to look for a **Dockerfile** file. If the OpenShift cluster finds a **Dockerfile** file in the root directory of the repository, then it triggers a new container build process.

```
[user@host ~]$ oc new-app --as-deployment-config \
> https://github.com/RedHatTraining/D0288-apps/ubi-echo
```

If your OpenShift cluster finds one of the files mentioned in the previous table in the root directory of the repository, then the OpenShift cluster starts an S2I process using its respective image builder.

To force the use of a certain image stream, you can use the **-i** option for a PHP 7.3 application:

```
[user@host ~]$ oc new-app --as-deployment-config -i php:7.3 \
> https://github.com/RedHatTraining/D0288-apps/php-helloworld
```

The OpenShift cluster looks for an image stream named **php** and looks for the 7.3 version to invoke the builder.

The Source-to-Image (S2I) Build Process

The S2I build process involves three fundamental components, which are combined to create a final container image:

Application source code

This is the source code for the application.

S2I scripts

S2I scripts are a set of scripts that the OpenShift build process executes to customize the S2I builder image. S2I scripts can be written in any programming language, as long as the scripts are executable inside the S2I builder image.

The S2I builder image

This is a container image that contains the required runtime environment for the application. It typically contains compilers, interpreters, scripts, and other dependencies that are needed to run the application.

The S2I build process relies on some S2I scripts, which it executes at various stages of the build workflow. The scripts, and a brief description of what they do, are listed in the following table:

Script	Mandatory?	Description
assemble	Yes	The assemble script builds the application from source and places it into appropriate directories inside the image.

Script	Mandatory?	Description
run	Yes	The run script executes your application. It is recommended to use the exec command when running any container processes in your run script. This ensures signal propagation and graceful shutdown of any process launched by the run script.
save-artifacts	No	The save-artifacts script gathers all dependencies required for the application and saves them to a tar file to speed up subsequent builds. For example, for Ruby, gems installed by Bundler, or for Java, .m2 contents. This means that the build does not have to redownload these contents, improving build time. These dependencies are gathered into a tar file and streamed to the standard output.
usage	No	The usage script provides a description of how to properly use your image.
test/run	No	The test/run script allows you to create a simple process to verify if the image is working correctly.

The S2I Build Workflow

The S2I build process is as follows:

1. OpenShift instantiates a container based on the S2I builder image, and then creates a tar file containing the source code and the S2I scripts. OpenShift then streams the tar file into the container.
2. Before running the **assemble** script, OpenShift extracts the tar file from the previous step, and saves the contents into the location specified by either the **--destination** option or by the **io.openshift.s2i.destination** label from the builder image. The default location is the **/tmp** directory.
3. If this is an incremental build, the **assemble** script restores the build artifacts previously saved in a tar file by the **save-artifacts** script.
4. The **assemble** script builds the application from source and places the binaries into the appropriate directories.
5. If this is an incremental build, the **save-artifacts** script is executed and saves all dependency build artifacts to a tar file.
6. After the **assemble** script has finished, the container is committed to create the final image, and OpenShift sets the **run** script as the **CMD** instruction for the final image.

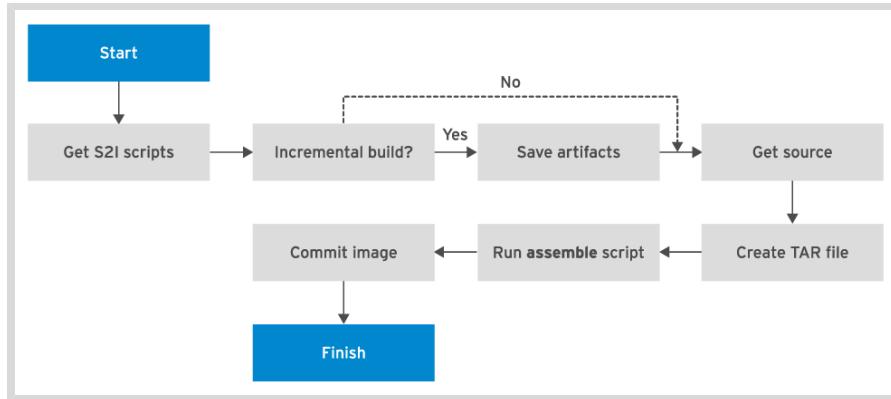


Figure 5.2: The S2I build workflow

To create a builder image, declare a **Dockerfile** file with the tools needed to create the application, such as compilers, build tools, and all the script files mentioned previously. The following **Dockerfile** builder file defines an NGINX server builder:

```

FROM registry.access.redhat.com/ubi8/ubi:8.0

LABEL io.k8s.description="My custom Builder" \①
      io.k8s.display-name="Nginx 1.6.3" \
      io.openshift.expose-services="8080:http" \
      io.openshift.tags="builder,webserver,html,nginx" \
      io.openshift.s2i.scripts-url="image:///usr/libexec/s2i" ②

RUN yum install -y epel-release && \③
    yum install -y --nodocs nginx && \
    yum clean all

EXPOSE 8080 ④

COPY ./s2i/bin/ /usr/libexec/s2i ⑤
  
```

- ① Set the labels that are used for OpenShift to describe the builder image.
- ② Tell S2I where to find its mandatory scripts (**run**, **assemble**).
- ③ Install the NGINX web server package and clean the Yum cache.
- ④ Set the default port for applications built using this image.
- ⑤ Copy the S2I scripts to the **/usr/libexec/s2i** directory.

The assemble script may be defined as follows:

```

#!/bin/bash -e

if [ "$(ls -A /tmp/src)" ]; then
    mv /tmp/src/* /usr/share/nginx/html/ ①
fi
  
```

- ① Override the default NGINX **index.html** file.

```
#!/bin/bash -e  
/usr/sbin/nginx -g "daemon off;"①
```

- ① Prevent the NGINX process from running as a daemon so that so that the container does not exit after the process runs `exec` script.

Overriding S2I Builder Scripts

S2I builder images provide default S2I scripts. You can override the default S2I scripts to change how your application is built and executed. You can override the default build behavior without needing to create a new S2I builder image by forking the source code for the original S2I builder.

The simplest way to override the default S2I scripts for an application is to include your S2I scripts in the source code repository for your application. You can provide S2I scripts in the `.s2i/bin` folder of the application source code repository.

When OpenShift starts the S2I process, it inspects the source code folder, the custom S2I scripts, and the application source code. OpenShift includes all of these files in the tar file injected into the S2I builder image. OpenShift then executes the custom `assemble` script instead of the default `assemble` script included with the S2I builder, followed by the other overridden custom scripts (if any).



References

How to override S2I builder scripts

<https://blog.openshift.com/override-s2i-builder-scripts/>

How to Create an S2I Builder Image

<https://blog.openshift.com/create-s2i-builder-image/>

Source-to-Image (S2I) Tool

<https://github.com/openshift/source-to-image>

s2i command line interface

<https://github.com/openshift/source-to-image/blob/master/docs/cli.md>

Further information about build environment variables from the standard OpenShift S2I builder images is available in the *Images* guide, in the documentation for Red Hat OpenShift Container Platform 4.5; at https://access.redhat.com/documentation/en-us/openshift_container_platform/4.5/html-single/images/

► Quiz

Describing the Source-to-Image Architecture

Choose the correct answers to the following questions:

- ▶ 1. Which of the following S2I scripts is responsible for building the application binaries in an S2I build?
 - a. **run**
 - b. **assemble**
 - c. **save-artifacts**
 - d. **test/run**
- ▶ 2. Which two of the following statements about the S2I build process are true? (Choose two.)
 - a. The **assemble** script is executed before the tar file containing the application source code and the S2I scripts is extracted.
 - b. The **assemble** script is executed after the tar file containing the application source code and the S2I scripts is extracted.
 - c. The **run** script is executed after the tar file containing the application source code and the S2I scripts is extracted.
 - d. The **run** script is set as the **CMD** instruction for the final container image.
 - e. The **assemble** script is set as the **CMD** instruction for the final container image.
- ▶ 3. In which of the following directories (relative to the root of your source code) would you provide your own custom S2I scripts?
 - a. **.scripts/bin**
 - b. **.s2i/bin**
 - c. **etc/bin**
 - d. **usr/local/bin**
- ▶ 4. Which two of the following scripts are mandatory in an S2I build? (Choose two.)
 - a. **usage**
 - b. **test/run**
 - c. **assemble**
 - d. **run**
 - e. **save-artifacts**

► **5. Which two of the following scenarios are good candidates for incremental S2I builds?**

(Choose two.)

- a. A Java EE application with a large number of JAR dependencies managed using Apache Maven.
- b. An application that depends on an SQL database dump file that is invoked at application startup.
- c. A Ruby web application that has a large number of static assets such as images, CSS, and HTML files.
- d. A Node.js application with dependencies managed using **npm**.

► **6. Which of the following labels indicates to the S2I builder image the directory where the scripts are stored?**

- a. io.openshift.s2i.scripts-url
- b. io.openshift.s2i.scripts-dir
- c. io.openshift.s2i.scripts-directory
- d. io.openshift.s2i.scripts-URL

► Solution

Describing the Source-to-Image Architecture

Choose the correct answers to the following questions:

- ▶ 1. Which of the following S2I scripts is responsible for building the application binaries in an S2I build?
 - a. `run`
 - b. `assemble`
 - c. `save-artifacts`
 - d. `test/run`

- ▶ 2. Which two of the following statements about the S2I build process are true? (Choose two.)
 - a. The `assemble` script is executed before the tar file containing the application source code and the S2I scripts is extracted.
 - b. The `assemble` script is executed after the tar file containing the application source code and the S2I scripts is extracted.
 - c. The `run` script is executed after the tar file containing the application source code and the S2I scripts is extracted.
 - d. The `run` script is set as the `CMD` instruction for the final container image.
 - e. The `assemble` script is set as the `CMD` instruction for the final container image.

- ▶ 3. In which of the following directories (relative to the root of your source code) would you provide your own custom S2I scripts?
 - a. `.scripts/bin`
 - b. `.s2i/bin`
 - c. `etc/bin`
 - d. `usr/local/bin`

- ▶ 4. Which two of the following scripts are mandatory in an S2I build? (Choose two.)
 - a. `usage`
 - b. `test/run`
 - c. `assemble`
 - d. `run`
 - e. `save-artifacts`

► **5. Which two of the following scenarios are good candidates for incremental S2I builds?**

(Choose two.)

- a. A Java EE application with a large number of JAR dependencies managed using Apache Maven.
- b. An application that depends on an SQL database dump file that is invoked at application startup.
- c. A Ruby web application that has a large number of static assets such as images, CSS, and HTML files.
- d. A Node.js application with dependencies managed using **npm**.

► **6. Which of the following labels indicates to the S2I builder image the directory where the scripts are stored?**

- a. io.openshift.s2i.scripts-url
- b. io.openshift.s2i.scripts-dir
- c. io.openshift.s2i.scripts-directory
- d. io.openshift.s2i.scripts-URL

Customizing an Existing S2I Base Image

Objectives

After completing this section, you should be able to customize the S2I scripts of an existing S2I builder image.

Customizing Scripts of an S2I Builder Image

The S2I scripts are packaged within the S2I builder images by default. In certain scenarios, you may want to customize these scripts to change the way your application is built and run, without rebuilding the image.

The S2I build process provides a method to override the default S2I scripts. You can provide your own S2I scripts in the **.s2i/bin** folder of the application source code. During the build process, the custom S2I scripts are automatically detected and run instead of the default S2I scripts packaged in the builder image.

Depending on the amount of customization that needs to be done to the overridden scripts, you may choose to completely replace the default S2I scripts with your own version. Alternatively, you can create a *wrapper* script that invokes the default scripts, and then adds the necessary customization before or after the invocation.

For example, suppose you want to customize the S2I scripts for the **rhscl/php-73-rhel7** S2I builder image, and change the way the application is built and run. You can use the following procedure to customize the S2I scripts provided in this builder image:

- Use the **podman pull** command to pull the container image from a conatiner registry to the local system. Use the **sudo podman inspect** command to get the value of the **io.openshift.s2i.scripts-url** label, in order to determine the default location of the S2I scripts in the image.

```
[user@host ~]$ sudo podman pull \
> myregistry.example.com/rhscl/php-73-rhel7
...
Digest: sha256:...
[user@host ~]$ sudo podman inspect --format='{{ index .Config.Labels
"io.openshift.s2i.scripts-url"}}' rhscl/php-73-rhel7
image:///usr/libexec/s2i
```

- You can also use the **skopeo inspect** command to retrieve the same information directly from a remote registry:

```
[user@host ~]$ skopeo inspect \
> docker://myregistry.example.com/rhscl/php-73-rhel7 \
> | grep io.openshift.s2i.scripts-url
"io.openshift.s2i.scripts-url": "image:///usr/libexec/s2i",
```

- Create a wrapper for the **assemble** script in the **.s2i/bin** folder:

```

#!/bin/bash
echo "Making pre-invocation changes..."

/usr/libexec/s2i/assemble
rc=$?

if [ $rc -eq 0 ]; then
    echo "Making post-invocation changes..."
else
    echo "Error: assemble failed!"
fi

exit $rc

```

- Similarly, create a wrapper for the **run** script in the **.s2i/bin** folder:

```

#!/bin/bash
echo "Before calling run..."
exec /usr/libexec/s2i/run

```



Note

When wrapping the **run** script, you must use **exec** to invoke it. This ensures that the default **run** script still runs with a process ID of 1. Failure to do this results in signal propagation errors during shutdown, and your application may not work correctly. This also implies that you cannot run additional commands in the wrapper script after invoking the default **run** script.

Incremental Builds in S2I

When building applications on an OpenShift cluster using S2I, it is very common to build, deploy, and test small incremental changes to your applications. Certain organizations have adopted *continuous integration (CI)* and *continuous delivery (CD)* techniques, where the application is built and deployed multiple times in a rapid iterative cycle, often without any manual intervention.

When your application is built in a modular fashion with several dependent components and libraries, S2I builds take up a lot of time due to the immutable nature of containers. The build process must fetch the dependencies and then build and deploy the application every time there is a change to the source code.

The S2I build process provides a mechanism to reduce build time by invoking the **save-artifacts** script after invoking the **assemble** script, as part of the S2I life cycle. The **save-artifacts** script ensures that dependent artifacts (libraries and components required for the application) are saved for future builds.

During the next build, the **assemble** script restores the cached artifacts before building the application from source code. Note that the **save-artifacts** script is responsible for streaming dependencies to stdout in a tar file.

**Important**

The **save-artifacts** script output should only include the tar stream output, and nothing else. You should redirect output of other commands in the script to **/dev/null**.

For example, assume you are developing a Java EE-based application with many dependencies managed using Apache Maven. Assuming you have built an S2I builder image where the **assemble** script compiles and packages the application JAR file, incremental builds that can reuse previously downloaded JAR files reduce the build time by a large margin. Apache Maven stores JAR dependencies in the **\$HOME/.m2** folder.

The **save-artifacts** script that caches Maven JAR files can be defined as follows:

```
#!/bin/sh -e

# Stream the .m2 folder tar archive to stdout
if [ -d ${HOME}/.m2 ]; then
    pushd ${HOME} > /dev/null
    tar cf - .m2
    popd > /dev/null
fi
```

The corresponding code to restore the artifacts before building is in the **assemble** script:

```
# Restore the .m2 folder
...
if [ -d /tmp/artifacts/.m2 ]; then
    echo "---> Restoring maven artifacts..."
    mv /tmp/artifacts/.m2 ${HOME}/
fi
...
```

**References**

Further information is available in the *Creating Images* chapter of the *Images* guide for Red Hat OpenShift Container Platform 4.5 at
https://access.redhat.com/documentation/en-us/openshift_container_platform/4.5/html-single/images/index#creating-images

How to override S2I builder scripts

<https://blog.openshift.com/override-s2i-builder-scripts>

► Guided Exercise

Customizing S2I Builds

In this exercise, you will customize the S2I scripts of an existing S2I builder image to add an information page to the application.

Outcomes

You should be able to customize the **assemble** and **run** scripts of an Apache HTTP server builder image. You will override the default built-in scripts with your own custom versions.

Before You Begin

To perform this exercise, ensure you have access to:

- A running OpenShift cluster.
- The **rhscl/httpd-24-rhel7** Apache HTTP server S2I builder image.
- A fork of the Git repository containing the **s2i-scripts** application source code.

Run the following command on the **workstation** VM to validate the exercise prerequisites and to download the lab and solution files:

```
[student@workstation ~]$ lab s2i-scripts start
```

► 1. Explore the S2I scripts packaged in the **rhscl/httpd-24-rhel7** builder image.

- 1.1. On the **workstation** VM, run the **rhscl/httpd-24-rhel7** image from a terminal window, and override the container entry point to run a shell:

```
[student@workstation ~]$ sudo podman run --name test -it rhscl/httpd-24-rhel7 bash
Trying to pull registry.access.redhat.com/rhscl/httpd-24-rhel7:latest...Getting
image source signatures
...output omitted...
bash-4.2$
```

- 1.2. Inspect the S2I scripts packaged in the builder image. The S2I scripts are located in the **/usr/libexec/s2i** folder:

```
bash-4.2$ cat /usr/libexec/s2i/assemble
...output omitted...
bash-4.2$ cat /usr/libexec/s2i/run
...output omitted...
bash-4.2$ cat /usr/libexec/s2i/usage
...output omitted...
```

Exit from the container:

```
bash-4.2$ exit
```

► 2. Review the application source code with the custom S2I scripts.

- 2.1. Enter your local clone of the **D0288-apps** Git repository and check out the **master** branch of the course's repository to ensure you start this exercise from a known good state:

```
[student@workstation ~]$ cd D0288-apps
[student@workstation D0288-apps]$ git checkout master
...output omitted...
```

- 2.2. Inspect the **/home/student/D0288-apps/s2i-scripts/index.html** file.

The HTML file contains a simple message:

```
Hello Class! D0288 rocks!!!
```

- 2.3. The custom S2I scripts are in the **/home/student/D0288-apps/s2i-scripts/.s2i/bin** folder. The **.s2i/bin/assemble** script copies the **index.html** file from the application source to the web server document root at **/opt/app-root/src**. It also creates an **info.html** file containing page build time and environment information:

```
...output omitted...
#####
# CUSTOMIZATION STARTS HERE #####
echo "---> Installing application source"
cp -Rf /tmp/src/*.html ./

DATE=`date "+%b %d, %Y @ %H:%M %p"`

echo "---> Creating info page"
echo "Page built on $DATE" >> ./info.html
echo "Proudly served by Apache HTTP Server version $HTTPD_VERSION" >> ./info.html

#####
# CUSTOMIZATION ENDS HERE #####
...output omitted...
```

- 2.4. The **.s2i/bin/run** script changes the default log level of the startup messages in the web server to **debug**:

```
# Make Apache show 'debug' level logs during startup
exec run-httpd -e debug $@
```

► 3. Deploy the application to an OpenShift cluster. Verify that the custom S2I scripts are executed.

- 3.1. Load your classroom environment configuration.

Run the following command to load the environment variables created in the first guided exercise:

```
[student@workstation D0288-apps]$ source /usr/local/etc/ocp4.config
```

- 3.2. Log in to OpenShift using your developer user account:

```
[student@workstation D0288-apps]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful.
...output omitted...
```

- 3.3. Create a new project for the application. Prefix the project's name with your developer username.

```
[student@workstation D0288-apps]$ oc new-project ${RHT_OCP4_DEV_USER}-s2i-scripts
Now using project "youruser-s2i-scripts" on server
"https://api.cluster.domain.example.com:6443".
...output omitted...
```

- 3.4. Create a new application called **bonjour** from sources in Git. You need to prefix the Git URL with the **httpd:2.4** image stream, using the tilde notation (~), to ensure that the application uses the **rhscl/httpd24-rhel7** builder image.

```
[student@workstation D0288-apps]$ oc new-app --as-deployment-config \
> --name bonjour \
> httpd:2.4-https://github.com/${RHT_OCP4_GITHUB_USER}/D0288-apps \
> --context-dir s2i-scripts
...output omitted...
--> Creating resources ...
imagestream.image.openshift.io "bonjour" created
buildconfig.build.openshift.io "bonjour" created
deploymentconfig.apps.openshift.io "bonjour" created
service "bonjour" created
--> Success
...output omitted...
```

- 3.5. View the build logs. Wait until the build finishes and the application container image is pushed to the OpenShift registry:

```
[student@workstation D0288-apps]$ oc logs -f bc/bonjour
Cloning "https://github.com/youruser/D0288-apps" ...
...output omitted...
--> Enabling s2i support in httpd24 image
AllowOverride All
--> Installing application source
--> Creating info page
Pushing image image-registry.openshift-image-registry.svc:5000/youruser-s2i-
scripts/bonjour:latest ... ...
...output omitted...
Push successful
```

Observe that the custom S2I scripts provided by the application are executed instead of the built-in S2I scripts from the builder image.

► 4. Test the application.

- 4.1. Wait until the application is deployed and then view the status of the application pod. The application pod should be in the **Running** state:

```
[student@workstation D0288-apps]$ oc get pods
NAME        READY   STATUS    RESTARTS   AGE
bonjour-1-build  0/1     Completed  0          2m3s
bonjour-1-deploy 0/1     Completed  0          66s
bonjour-1-km4bq   1/1     Running   0          58s
```

- 4.2. Expose the application for external access using a route.

```
[student@workstation D0288-apps]$ oc expose svc bonjour
route.route.openshift.io/bonjour exposed
```

- 4.3. Obtain the route URL using the **oc get route** command:

```
[student@workstation D0288-apps]$ oc get route
NAME      HOST/PORT
bonjour   bonjour-youruser-s2i-scripts.apps.cluster.domain.example.com ...
```

- 4.4. Invoke the index page of the application with the **curl** command and the route URL from the previous command:

```
[student@workstation D0288-apps]$ curl \
> http://bonjour-${RHT_OCP4_DEV_USER}-s2i-scripts.${RHT_OCP4_WILDCARD_DOMAIN}
Hello Class! D0288 rocks!!!
```

You should see the contents of the **index.html** file in the application source.

- 4.5. Invoke the info page of the application with the **curl** command:

```
[student@workstation D0288-apps]$ curl \
> http://bonjour-${RHT_OCP4_DEV_USER}-s2i-scripts.${RHT_OCP4_WILDCARD_DOMAIN}\ \
> /info.html
Page built on Jun 11, 2019 @ 16:12 PM
Proudly served by Apache HTTP Server version 2.4
```

You should see the contents of the **info.html** file with details about when the page was built and the version of the Apache HTTP server.

- 4.6. Inspect the logs for the application pod. Recall that the startup log level was changed to **debug** in the **run** script. You should see **debug** level log messages being displayed at startup:

```
[student@workstation D0288-apps]$ oc logs bonjour-1-km4bq
...output omitted...
[Fri Nov 03 16:12:21.690941 2017] [so:debug] [pid 9] mod_so.c(266): AH01575:
 loaded module systemd_module from /opt/rh/httpd24/root/etc/httpd/modules/
mod_systemd.so
[Fri Nov 03 16:12:21.691050 2017] [so:debug] [pid 9] mod_so.c(266): AH01575:
 loaded module cgi_module from /opt/rh/httpd24/root/etc/httpd/modules/mod_cgi.so
```

```
...output omitted...
[Fri Nov 03 16:12:21.742471 2017] [ssl:debug] [pid 9] ssl_engine_init.c(270):
AH01886: SSL FIPS mode disabled
...output omitted...
[Fri Nov 03 16:12:21.745520 2017] [mpm_prefork:notice] [pid 9] AH00163:
Apache/2.4.25 (Red Hat) OpenSSL/1.0.1e-fips configured -- resuming normal
operations
[Fri Nov 03 16:12:21.745530 2017] [core:notice] [pid 9] AH00094: Command line:
'httpd -D FOREGROUND -e debug'
...output omitted...
10.131.0.16 - - [03/Nov/2019:16:28:53 +0000] "GET / HTTP/1.1" 200 28 "-"
"curl/7.29.0"
10.128.2.216 - - [03/Nov/2019:16:29:03 +0000] "GET /info.html HTTP/1.1" 200 87 "-"
"curl/7.29.0"
```

- ▶ 5. Cleanup. Delete the project:

```
[student@workstation DO288-apps]$ cd ~
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-s2i-scripts
```

- ▶ 6. Remove the **test** container you created earlier to view the default S2I scripts.

```
[student@workstation ~]$ sudo podman rm test
```

Finish

On the **workstation** VM, run the **lab s2i-scripts finish** script to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab s2i-scripts finish
```

This concludes the guided exercise.

Creating an S2I Builder Image

Objectives

After completing this section, you should be able to create an S2I builder image using the **s2i** command-line tool.

Building and Publishing a Custom S2I Builder Image

The S2I build process combines application source code with an appropriate *S2I builder image* to produce the final application container image that is deployed to an OpenShift cluster.

Before deploying the S2I builder image to an OpenShift cluster, where other developers can use it for building applications, it is important to build and test the image using the **s2i** command-line tool. Install the **s2i** tool on your local machine to build and test your S2I builder images outside of an OpenShift cluster.

Installing the S2I Tool

On RHEL 7 systems, the **s2i** tool is available in the *source-to-image* package and can be installed using Yum. Ensure that your system has subscribed to and enabled the **rhel-server-rhscl-7-rpms** yum repository.

For other operating systems, the **s2i** tool can be downloaded from the upstream **source-to-image** project page at:

<https://github.com/openshift/source-to-image/releases>

Using the S2I Tool

Use the **s2i create** command to create the template files required to create a new S2I builder image:

```
[user@host ~]$ s2i create image_name directory
```

The above command creates a folder named **directory** and populates it with the following template files that you can update as needed:

```
directory
├── Dockerfile ①
├── Makefile
├── README.md
└── s2i ②
    └── bin
        ├── assemble
        ├── run
        ├── save-artifacts
        └── usage
└── test
```

```

└── run
  └── test-app ③
    └── index.html

```

- ① The Dockerfile for the S2I builder image
- ② The S2I scripts directory
- ③ Folder to copy your application source code to for testing locally

The **s2i create** command creates a template Dockerfile with comments, tailored for running on an OpenShift cluster with a random user ID and OpenShift-specific labels. It also creates stubs for the S2I scripts that you can customize to fit the needs of your application. After you update the Dockerfile and S2I scripts as needed, you can build the builder image using the **podman build** command.

```
[user@host ~]$ sudo podman build -t builder_image .
```

When the builder image is ready, you can build an application container image using the **s2i build** command. This allows you to test the S2I builder image locally, without the need to push it to a registry server and deploy an application using the builder image to an OpenShift cluster:

```
[user@host ~]$ s2i build src builder_image tag_name
```



Note

If you include any instructions that are not part of the OCI specification, such as the **ONBUILD** instruction in your builder image Dockerfile, be sure to use the **--format docker** option with the **podman build** command when building your S2I builder image. This option overrides the default **oci** format used by Podman.

The **s2i build** command combines the application source code defined in the **src** option with the **builder_image** container image to produce the application container image with a tag of **tag_name**. This command emulates the S2I process that the OpenShift cluster uses, by injecting the provided source code into the builder image automatically, but it uses the local Docker service to build a test container image, instead of deploying the image to the OpenShift cluster.

Test the application container image produced by the **s2i build** command by running the image using the **podman run** command. Note that if the application container image will be deployed to OpenShift, you need to simulate running the container with a random user ID using the **-u** flag for the **podman run** command.

Important

The **s2i build** command requires the use of a local Docker service because it uses the Docker API directly via the socket to build the S2I container image. In RHEL 8 and OpenShift 4 environments, Docker is not included, and this does not work.

To provide support for environments that do not have Docker available, the **s2i build** command now includes the **--as-dockerfile path/to/Dockerfile** option. This option configures the **s2i build** command to produce a Dockerfile and two supporting directories that you can use to build a test container image from a source repository and builder image using the **podman build** command. By using this option, no local Docker daemon is required to run the **s2i build** command.

You can provide either the location of a directory or a Git repository URL containing the application source as the **src** option.

For incremental builds, be sure to create a **save-artifacts** script and pass the **--incremental** flag to the **s2i build** command. If a **save-artifacts** script exists, and a prior image already exists, and you use the **--incremental=true** option, then the workflow is as follows:

1. S2I creates a new container image from the prior build image.
2. S2I runs the **save-artifacts** script in this container. This script is responsible for streaming out a tar file of the artifacts to stdout.
3. S2I builds the new output image:
 - a. The artifacts from the previous build are in the **artifacts** directory of the tar file passed to the build.
 - b. The S2I builder image's **assemble** script is responsible for detecting and using the build artifacts.

Run the **s2i --help** and **s2i subcommand --help** commands to learn about the various subcommands, their options, and examples on how to use them.

After you have tested the application container image locally, you can copy the S2I builder image to a registry for consumption. Before deploying applications to an OpenShift cluster using the builder image, you must create an image stream based on the builder image using the **oc import-image** command. You can then use the image stream to create applications in OpenShift.

Building an Nginx S2I Builder Image

To create an Nginx S2I builder image based on RHEL 7, perform the following steps:

Create and Populate the Dockerfile project

Use the **s2i** command create the S2I builder image project directory, and edit the files as needed:

- Run the **s2i create** command to create the skeleton directory structure for S2I builder image artifacts:

```
[user@host ~]$ s2i create s2i-do288-nginx s2i-do288-nginx
```

- Edit the Dockerfile to include instructions to install Nginx and the S2I scripts. The following Dockerfile for an Nginx S2I builder image uses the RHEL 8 Universal Base Image:

```
FROM registry.access.redhat.com/ubi8/ubi:8.0 1

ENV X_SCLS="rh-nginx18" \
    PATH="/opt/rh/rh-nginx18/root/usr/sbin:$PATH" \
    NGINX_DOCROOT="/opt/rh/rh-nginx18/root/usr/share/nginx/html"

LABEL io.k8s.description="A Nginx S2I builder image" \ 2
      io.k8s.display-name="Nginx 1.8 S2I builder image for DO288" \
      io.openshift.expose-services="8080:http" \
      io.openshift.s2i.scripts-url="image:///usr/libexec/s2i" \
      io.openshift.tags="builder,webserver,nginx,nginx18,html"

ADD nginxconf.sed /tmp/
COPY ./s2i/bin/ /usr/libexec/s2i 3

RUN yum install -y --nодocs rh-nginx18 \ 4
    && yum clean all \
    && sed -i -f /tmp/nginxconf.sed /etc/opt/rh/rh-nginx18/nginx/nginx.conf \
    && chgrp -R 0 /var/opt/rh/rh-nginx18 /opt/rh/rh-nginx18 \ 5
    && chmod -R g=u /var/opt/rh/rh-nginx18 /opt/rh/rh-nginx18 \ 6
    && echo 'Hello from the Nginx S2I builder image' > ${NGINX_DOCROOT}/index.html

EXPOSE 8080

USER 1001

CMD ["/usr/libexec/s2i/usage"]
```

- 1** Use the RHEL 8 Universal Base Image as the base for the S2I builder image.
- 2** Label metadata for S2I builder image consumers.
- 3** Copy the S2I scripts to the location indicated by the **io.openshift.s2i.scripts-url** label.
- 4** Install Nginx.
- 5** **6** Set permissions to run as a random user ID on an OpenShift cluster.

- Create an **assemble** script in the **.s2i/bin** directory of the application source with the following content, which copies the HTML source files to the Nginx web server document root:

```
#!/bin/bash -e

echo "--> Copying source HTML files to web server root..."
cp -Rf /tmp/src/. /opt/rh/rh-nginx18/root/usr/share/nginx/html/
```

- Create a **run** script in the **.s2i/bin** directory of the application source with the following content, which runs the Nginx web server in the foreground:

```
#!/bin/bash -e

exec nginx -g "daemon off;"
```

Building and Testing the S2I Builder Image

Use Podman and the **s2i** command to build the S2I builder image and a test application container image:

- Build the S2I builder image:

```
[user@host ~]$ sudo podman build -t s2i-do288-nginx .
```

- Run the **s2i build** command to build a test application container image. To override the default **index.html** file included in the builder image, create an **index.html** file under the **test/test-app** directory to inject this new file into the test Nginx container:

```
[user@host ~]$ s2i build test/test-app s2i-do288-nginx nginx-test \
> --as-docker-file /path/to/Dockerfile
```

- To build the test container, use the **podman build** command, this time using the Dockerfile generated by the **s2i build** command:

```
[user@host ~]$ sudo podman build -t nginx-test /path/to/Dockerfile
```

- To test the container image, use the **podman run** command with a user ID different from the one provided in the Dockerfile, to ensure that the container can be run with a randomly generated user ID on an OpenShift cluster:

```
[user@host ~]$ sudo podman run -u 1234 -d -p 8080:8080 nginx-test
```

When the container is running without errors, verify that the test **index.html** file you supplied in the **test** directory is rendered when testing the Nginx container.

Making the S2I Builder Image Available to OpenShift

Use the **skopeo** command to push the S2I builder image to a container registry, and create an image stream in OpenShift that points to that image.

- After you test the container locally, push the S2I builder image to an enterprise registry. For example, assume you have a Quay.io account and have logged in using the **podman login** command:

```
[user@host ~]$ sudo skopeo copy containers-storage:localhost/s2i-do288-httdp \
> docker://quay.io/${RHT_OCP4_QUAY_USER}/s2i-do288-httdp
```

- To create an image stream for the Nginx S2I builder image, create a new project and run the **oc import-image** command:

```
[user@host ~]$ oc import-image s2i-do288-nginx \
> --from quay.io/${RHT_OCP4_QUAY_USER}/s2i-do288-nginx \
> --confirm
```



Note

Recall from *Chapter 3, Publishing Enterprise Container Images* that you may need to create a pull secret containing your credentials for the remote registry where you publish your S2I builder image, if that image is not publicly available. You also need to link that secret to the default service account used to pull images in order to create the image stream using the **oc import-image** command.

- After you create the image stream, you can use it to create applications using the Nginx S2I builder image. Note that you need to use the tilde notation (~) unless your S2I builder image is an alternative to the languages supported by the OpenShift **oc new-app** command:

```
[user@host ~]$ oc new-app --as-deployment-config --name nginx-test \
> s2i-do288-nginx~git_repository
```



References

Further information is available in the *Creating Images* chapter of the *Images* guide for Red Hat OpenShift Container Platform 4.5 at
https://access.redhat.com/documentation/en-us/red_hat_openshift_container_platform/4.5/html-single/images/creating-images#creating-images

How to Create an S2I Builder Image

<https://blog.openshift.com/create-s2i-builder-image>

Source-to-Image (S2I) Tool

<https://github.com/openshift/source-to-image>

s2i tool subcommand reference

<https://github.com/openshift/source-to-image/blob/master/docs/cli.md>

► Guided Exercise

Creating an S2I Builder Image

In this exercise, you will build and test an Apache HTTP server S2I builder image and then build and deploy an application using the image.

Outcomes

You should be able to:

- Build an Apache HTTP server S2I builder image using the **s2i** tool.
- Test the S2I builder image locally using a simple application.
- Publish the builder image to the Quay.io container registry.
- Deploy and test an application on an OpenShift cluster using the S2I builder image.

Before You Begin

To perform this exercise, ensure you have access to:

- A running OpenShift cluster.
- The parent image (**ubi8/ubi**) for the sample application.
- The sample application (html-helloworld) in the Git repository.

Run the following command on the **workstation** VM to validate the exercise prerequisites and to download the lab and solution files:

```
[student@workstation ~]$ lab apache-s2i start
```

- 1. On the **workstation** VM, verify that the *source-to-image* package is installed, which provides the **s2i** command-line tool:

```
[student@workstation ~]$ s2i version
s2i v1.1.13
```

- 2. Use the **s2i** command to create the template files and directories needed for the S2I builder image.

- 2.1. On the **workstation** VM, use the **s2i create** command to create the template files for the builder image in the **/home/student/** directory:

```
[student@workstation ~]$ s2i create s2i-do288-httpd s2i-do288-httpd
```

- 2.2. Verify that the template files have been created. The **s2i create** command creates the following directory structure:

```
[student@workstation ~]$ tree -a s2i-do288-httdp
s2i-do288-httdp
├── Dockerfile
├── Makefile
└── README.md
└── s2i
    ├── bin
    │   ├── assemble
    │   ├── run
    │   ├── save-artifacts
    │   └── usage
    └── test
        ├── run
        └── test-app
            └── index.html
4 directories, 9 files
```

► 3. Create the Apache HTTP server S2I builder image.

- 3.1. An example Dockerfile for the Apache HTTP server builder image is provided for you at [~/D0288/labs/apache-s2i/Dockerfile](#). Briefly review this file:

```
[student@workstation ~]$ cat ~/D0288/labs/apache-s2i/Dockerfile
FROM registry.access.redhat.com/ubi8/ubi:8.0 ①

# Generic labels
LABEL Component="httpd" \
      Name="s2i-do288-httdp" \
      Version="1.0" \
      Release="1"

# Labels consumed by OpenShift
LABEL io.k8s.description="A basic Apache HTTP Server S2I builder image" \
      io.k8s.display-name="Apache HTTP Server S2I builder image for D0288" \
      io.openshift.expose-services="8080:http" \
      io.openshift.s2i.scripts-url="image:///usr/libexec/s2i" ② ③

# This label is used to categorize this image as a builder image in the
# OpenShift web console.
LABEL io.openshift.tags="builder, httpd, httpd24"

# Apache HTTP Server DocRoot
ENV DOCROOT /var/www/html

RUN yum install -y --nodocs --disableplugin=subscription-manager httpd && \
    yum clean all --disableplugin=subscription-manager -y && \
    echo "This is the default index page from the s2i-do288-httdp S2I builder
image." > ${DOCROOT}/index.html ④ ⑤

# Change web server port to 8080
RUN sed -i "s/Listen 80/Listen 8080/g" /etc/httpd/conf/httpd.conf

# Copy the S2I scripts to the default location indicated by the
```

```
# io.openshift.s2i.scripts-url LABEL (default is /usr/libexec/s2i)
COPY ./s2i/bin/ /usr/libexec/s2i ⑥
...output omitted...
```

- ① Use the RHEL 8 Universal Base Image as the base image for this container
- ② Set the labels that are used for OpenShift to describe the builder image.
- ③ Configure where the mandatory S2I scripts (**run**, **assemble**) are located
- ④ Install the httpd web server package and clean the Yum cache.
- ⑤ Set the content of the default **index.html** file for the builder image.
- ⑥ Copy the S2I scripts to the **/usr/libexec/s2i** directory.

Then, copy this Dockerfile to the **~/s2i-do288-httpd** directory and overwrite the generated template Dockerfile:

```
[student@workstation ~]$ cp ~/D0288/labs/apache-s2i/Dockerfile ~/s2i-do288-httpd/
```

- 3.2. Similarly, review and copy the S2I scripts for this builder image provided in the **~/D0288/labs/apache-s2i/s2i/bin** directory to the **~/s2i-do288-httpd/s2i/bin** directory and overwrite the generated scripts:

```
[student@workstation ~]$ cp -Rv ~/D0288/labs/apache-s2i/s2i ~/s2i-do288-httpd/
```

- 3.3. This exercise does not implement the **save-artifacts** script. Remove it from the **~/s2i-do288-httpd/s2i/bin** directory:

```
[student@workstation ~]$ rm -f ~/s2i-do288-httpd/s2i/bin/save-artifacts
```

- 3.4. Create the Apache HTTP server S2I builder image. Do not forget the trailing period at the end of the **podman build** command. It indicates that Docker should use the Dockerfile in the current directory to build the image:

```
[student@workstation ~]$ cd s2i-do288-httpd
[student@workstation s2i-do288-httpd]$ sudo podman build -t s2i-do288-httpd .
STEP 1: FROM registry.access.redhat.com/ubi8/ubi:8.0
...output omitted...
STEP 25: COMMIT s2i-do288-httpd
```



Note

You can safely ignore the following error in the build output: **ERRO[0000] HOSTNAME is not supported for OCI image format, hostname 1d561c58fd2b will be ignored. Must use `docker` format** This is a bug in the version of the Podman tool, and is a false error message, the container build proceeds without issue despite the error. More information can be found in this Bugzilla [https://bugzilla.redhat.com/show_bug.cgi?id=1661592]

- 3.5. Verify that the builder image was created:

```
[student@workstation s2i-do288-httdp]$ sudo podman images
REPOSITORY                                     TAG      IMAGE ID      CREATED
localhost/s2i-do288-httdp                     latest   82beb27428b7  9 seconds ago
registry.access.redhat.com/ubi8/ubi           8.0     7ae69d957d8b  2 weeks ago
...output omitted...
```

- ▶ 4. Build and test an application container image that combines the Apache HTTP server S2I builder image and the application source code.
- 4.1. When the builder image is ready, you can use the **s2i build** command to build the application container image. Before building the application container image, review the sample **~/D0288/labs/apache-s2i/index.html** HTML file:

```
[student@workstation s2i-do288-httdp]$ cat ~/D0288/labs/apache-s2i/index.html
This is the index page from the app
```

Then, copy this file to the **~/s2i-do288-httdp/test/test-app** directory to override the **index.html** file packaged inside the builder image:

```
[student@workstation s2i-do288-httdp]$ cp ~/D0288/labs/apache-s2i/index.html \
> ~/s2i-do288-httdp/test/test-app/
```

- 4.2. Create a new directory for the Dockerfile generated by the **s2i build** command.

```
[student@workstation s2i-do288-httdp]$ mkdir /home/student/s2i-sample-app
```

- 4.3. Build the application container image:

```
[student@workstation s2i-do288-httdp]$ s2i build test/test-app/ \
> s2i-do288-httdp s2i-sample-app \
> --as-dockerfile ~/s2i-sample-app/Dockerfile
Application dockerfile generated in /home/student/s2i-sample-app/Dockerfile
```

- 4.4. Review the generated application directory.

```
[student@workstation s2i-do288-httdp]$ cd ~/s2i-sample-app
[student@workstation s2i-sample-app]$ tree .
.
├── Dockerfile
├── downloads
│   ├── defaultScripts
│   └── scripts
└── upload
    ├── scripts
    └── src
        └── index.html

6 directories, 2 files
```

Observe the **index.html** file located in the **upload** directory. This is the same **index.html** file you copied into the **test/test-app** directory in a previous step.

4.5. Review the generated Dockerfile.

```
[student@workstation s2i-sample-app]$ cat Dockerfile
FROM s2i-do288-httdp ①
LABEL "io.k8s.display-name"="s2i-sample-app" \
      "io.openshift.s2i.build.image"="s2i-do288-httdp" \
      "io.openshift.s2i.build.source-location"="test/test-app/"

USER root
# Copying in source code
COPY upload/src /tmp/src ③
# Change file ownership to the assemble user. Builder image must support chown
# command.
RUN chown -R 1001:0 /tmp/src
USER 1001
# Assemble script sourced from builder image based on user input or image
# metadata.
# If this file does not exist in the image, the build will fail.
RUN /usr/libexec/s2i/assemble
# Run script sourced from builder image based on user input or image metadata.
# If this file does not exist in the image, the build will fail.
CMD /usr/libexec/s2i/run
```

- ① Use the **s2i-do288-httdp** builder image as the parent of this container image.
- ② Set the labels that are used for OpenShift to describe the application.
- ③ Copy in the source code contained in the **upload/src** directory.

4.6. Build a test container image from the generated Dockerfile.

```
[student@workstation s2i-sample-app]$ sudo podman build \
> --format docker -t s2i-sample-app .
STEP 1: FROM s2i-do288-httdp
STEP 2: LABEL "io.k8s.display-name"="s2i-sample-app"...output omitted...
...output omitted...
STEP 15: COMMIT s2i-sample-app
```

4.7. Verify that the application container image was created:

```
[student@workstation s2i-sample-app]$ sudo podman images
REPOSITORY                      TAG      IMAGE ID      CREATED
localhost/s2i-sample-app        latest   3c8637c4372d  About an hour ago
localhost/s2i-do288-httdp       latest   d06040a9eeeca About an hour ago
...output omitted...
```

4.8. Test the application container image locally on the **workstation** VM. Run the container as a random user with the **-u** flag to simulate running as a random user on an OpenShift cluster. You can copy the command, or run it directly from the **~/DO288/labs/apache-s2i/local-test.sh** script:

```
[student@workstation s2i-sample-app]$ sudo podman run --name test -u 1234 \
> -p 8080:8080 -d s2i-sample-app
a5f4b3e5bf...output omitted...
```

- 4.9. Verify that the container started successfully:

```
[student@workstation s2i-sample-app]$ sudo podman ps
CONTAINER ID   IMAGE               COMMAND   ...
a5f4b3e5bfaa   localhost/s2i-sample-app:latest   /bin/sh -c /usr/lib...
...output omitted...
```

- 4.10. Use the **curl** command to test the application:

```
[student@workstation s2i-sample-app]$ curl http://localhost:8080
This is the index page from the app
```

- 4.11. Stop the test application container:

```
[student@workstation s2i-sample-app]$ sudo podman stop test
a5f4b3e5bf...output omitted...
```

► 5. Push the Apache HTTP server S2I builder image to your Quay.io account.

- 5.1. Load your classroom environment configuration.

Run the following command to load the environment variables created in the first guided exercise:

```
[student@workstation s2i-sample-app]$ source /usr/local/etc/ocp4.config
```

- 5.2. Log in to your Quay.io account using the **podman** command. You will be prompted to enter your password.

```
[student@workstation s2i-sample-app]$ sudo podman login \
> -u ${RHT_OCP4_QUAY_USER} quay.io
Password:
Login Succeeded!
```

- 5.3. Use the **skopeo copy** command to publish the S2I builder image to your Quay.io account.

```
[student@workstation s2i-sample-app]$ sudo skopeo copy \
> containers-storage:localhost/s2i-do288-httd \
> docker://quay.io/${RHT_OCP4_QUAY_USER}/s2i-do288-httd
...output omitted...
Writing manifest to image destination
Storing signatures
```

► 6. Create an image stream for the Apache HTTP Server S2I builder image.

- 6.1. Log in to OpenShift and create a new project. Prefix the project's name with your developer username.

```
[student@workstation s2i-sample-app]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful.
...output omitted...
[student@workstation s2i-sample-app]$ oc new-project \
> ${RHT_OCP4_DEV_USER}-apache-s2i
Now using project "youruser-apache-s2i" on server "https://
api.cluster.domain.example.com:6443".
```

- 6.2. Log in to your personal Quay.io account using Podman, this time without the **sudo** command, so that you can export the **auth.json** file to use it in a pull secret in the next step. You need to enter your password again.

```
[student@workstation s2i-sample-app]$ podman login \
> -u ${RHT_OCP4_QUAY_USER} quay.io
Password:
Login Succeeded!
```

- 6.3. Create a secret from the container registry API access token that was stored by Podman.

You can also execute or cut and paste the following **oc create secret** command from the **create-secret.sh** script in the **/home/student/D0288/labs/apache-s2i** directory.

```
[student@workstation s2i-sample-app]$ oc create secret generic quayio \
> --from-file .dockerconfigjson=${XDG_RUNTIME_DIR}/containers/auth.json \
> --type=kubernetes.io/dockerconfigjson
secret/quayio created
```

- 6.4. Link the new secret to the **builder** service account.

```
[student@workstation s2i-sample-app]$ oc secrets link builder quayio
```

- 6.5. Create an image stream by importing the S2I builder image from your Quay.io container registry:

```
[student@workstation s2i-sample-app]$ oc import-image s2i-do288-httd \
> --from quay.io/${RHT_OCP4_QUAY_USER}/s2i-do288-httd --confirm
imagestream.image.openshift.io/s2i-do288-httd imported

Name: s2i-do288-httd
Namespace: youruser-apache-s2i
Created: Less than a second ago
Labels: <none>
Annotations: openshift.io/image.dockerRepositoryCheck=2019-06-26T02:30:59Z
Image Repository: image-registry.openshift-image-registry.svc:5000/youruser-
apache-s2i/s2i-do288-httd
Image Lookup: local=false
Unique Images: 1
Tags: 1
```

```
latest
tagged from quay.io/youruser/s2i-do288-httpd

* quay.io/youruser/s2i-do288-httpd@sha256:fe0cd09432...
  Less than a second ago

...output omitted...
```

6.6. Verify that the image stream was created:

```
[student@workstation s2i-sample-app]$ oc get is
NAME           IMAGE REPOSITORY      ...output omitted...
s2i-do288-httpd  ...youruser-apache-s2i/s2i-do288-httpd
```

- ▶ 7. Deploy and test the html-helloworld application from the classroom Git repository on an OpenShift cluster. This application consists of a single HTML file that displays a message.
- 7.1. Create a new application called **hello** from sources in Git. You need to prefix the Git URL with the **s2i-do288-httpd** image stream to ensure that the application uses the Apache HTTP server builder image you created earlier:

```
[student@workstation s2i-sample-app]$ oc new-app --as-deployment-config \
> --name hello-s2i \
> s2i-do288-httpd~https://github.com/${RHT_OCP4_GITHUB_USER}/D0288-apps \
> --context-dir=html-helloworld
--> Found image c7a496d (2 hours old) in image stream "youruser-apache-s2i/s2i-
do288-httpd" under tag "latest" for "s2i-do288-httpd"

Apache HTTP Server S2I builder image for D0288
-----
A basic Apache HTTP Server S2I builder image
...output omitted...
--> Creating resources ...
...output omitted...
--> Success
...output omitted...
```

7.2. View the build logs. Wait until the build finishes and the application container image is pushed to the OpenShift registry:

```
[student@workstation s2i-sample-app]$ oc logs -f bc/hello-s2i
Cloning "https://github.com/youruser/D0288-apps" ...
...output omitted...
--> Copying source files to web server directory...
Pushing image-registry.openshift-image-registry.svc:5000/youruser-apache-s2i/
hello-s2i:latest ...
...output omitted...
Push successful
```

7.3. Wait until the application is deployed. View the status of the application pod. The application pod should be in the **Running** state:

```
[student@workstation s2i-sample-app]$ oc get pods
NAME          READY   STATUS    RESTARTS   AGE
hello-s2i-1-build  0/1     Completed   0          2m
hello-s2i-1-deploy 0/1     Completed   0          72s
hello-s2i-1-w8nqp  1/1     Running    0          63s
```

7.4. Expose the application for external access by using a route:

```
[student@workstation s2i-sample-app]$ oc expose svc hello-s2i
route.route.openshift.io/hello-s2i exposed
```

7.5. Obtain the route URL using the **oc get route** command:

```
[student@workstation s2i-sample-app]$ oc get route/hello-s2i \
> -o jsonpath='{.spec.host}{"\n"}'
hello-s2i-youruser-apache-s2i.apps.cluster.domain.example.com
```

7.6. Test the application using the route URL you obtained in the previous step:

```
[student@workstation s2i-sample-app]$ curl \
> http://hello-s2i-${RHT_OCP4_DEV_USER}-apache-s2i.${RHT_OCP4_WILDCARD_DOMAIN}
<html>
<body>
  <h1>Hello, World!</h1>
</body>
</html>
```

► 8. Clean up.

8.1. Delete the **apache-s2i** project in OpenShift:

```
[student@workstation s2i-sample-app]$ oc delete project \
> ${RHT_OCP4_DEV_USER}-apache-s2i
```

8.2. Delete the **test** container created earlier when testing the application locally:

```
[student@workstation s2i-sample-app]$ sudo podman rm test
a5f4b3e5bf...output omitted...
```

8.3. Delete the container images from the **workstation** VM:

```
[student@workstation s2i-sample-app]$ sudo podman rmi -f \
> localhost/s2i-sample-app \
> localhost/s2i-do288-httdp \
> registry.access.redhat.com/ubi8/ubi:8.0
...output omitted...
```

8.4. Delete the **s2i-do288-httdp** image from the external registry:

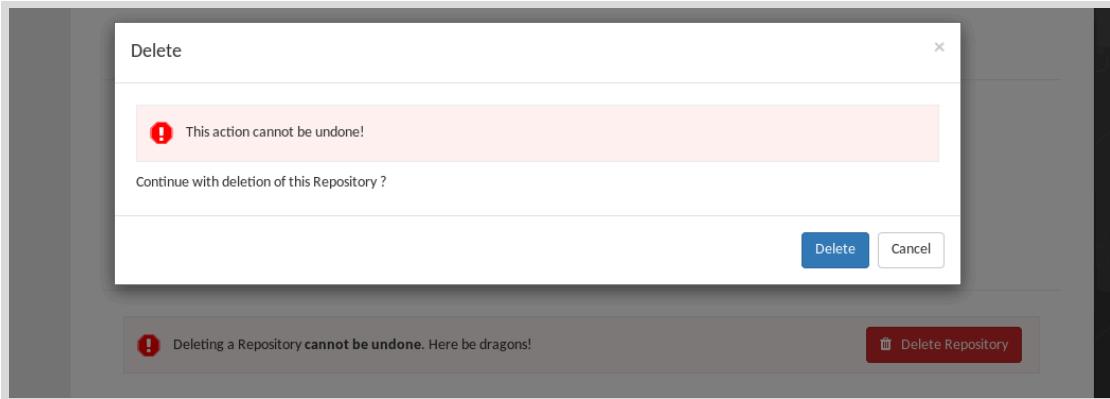
```
[student@workstation s2i-sample-app]$ sudo skopeo delete \
> docker://quay.io/${RHT_OCP4_QUAY_USER}/s2i-do288-httd:latest
```

- 8.5. Log in to Quay.io using your personal free account.

Navigate to `http://quay.io` and click **Sign In** to provide your user credentials.
Click **Sign in to Quay Container Registry** to log in to Quay.io.

- 8.6. On the Quay.io main menu, click **Repositories** and look for **s2i-do288-httd**. The lock icon next to it indicates it is a private repository that requires authentication for both pulls and pushes. Click **s2i-do288-httd** to display the **Repository Activity** page.
- 8.7. On the **Repository Activity** page for the **s2i-do288-httd** repository, scroll down and click the gear icon to display the **Settings** tab. Scroll down and click **Delete Repository**.

- 8.8. In the **Delete** dialog box, click **Delete** to confirm you want to delete the **s2i-do288-httd** repository. After a few moments you are returned to the **Repositories** page. You can now sign out of Quay.io.



Finish

On the **workstation** VM, run the **lab apache-s2i finish** script to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab apache-s2i finish
```

This concludes the guided exercise.

▶ Lab

Customizing Source-to-Image Builds

Performance Checklist

In this lab, you will create an S2I builder image for running applications based on the Go programming language.

Outcomes

You should be able to:

- Build a Go programming language S2I builder image based on the Universal Base Image for RHEL 8.
- Test the S2I builder image locally by creating a Dockerfile using the **s2i** tool and then building and running the resulting container image with Podman.
- Publish the builder image to your personal Quay.io account.
- Use the S2I builder image to deploy and test an application on an OpenShift cluster.
- Customize the **run** script to change the behavior of the application, and redeploy the application to test the changes.

Before You Begin

To perform this lab, you must have access to:

- A running OpenShift cluster.
- The **s2i** command-line tool.
- A fork of the Git repository containing the go-hello application source code.

Run the following command on the **workstation** VM to validate the prerequisites. This command also downloads helper files and solution files for the review lab:

```
[student@workstation ~]$ lab custom-s2i start
```

The setup script creates a folder called **custom-s2i** inside the **/home/student/D0288/labs** folder. This folder contains the template folder structure and files created by the **s2i create** command.

Requirements

In this lab, you need to deploy an application that is written in the Go programming language.

A small example Go application is provided for you to use to test your S2I builder image. The example Go application provides a simple HTTP API that responds to requests based on the resource requested in the HTTP request.

To complete the lab, build an S2I builder image for Go-based applications, and then test and deploy the example Go application on the classroom OpenShift cluster according to the following requirements:

- The S2I builder image must be named **s2i-do288-go**. The builder image must be available from your personal Quay.io account at:

`quay.io/youruser/s2i-do288-go`

- The image stream for the S2I builder image is named **s2i-do288-go**.
- The application name for OpenShift is **greet**.
- Both the image stream and the application must be created within a project named **youruser-custom-s2i**.

- The HTTP API for the application must be accessible at the following URL:

`http://greet-youruser-custom-s2i.apps.cluster.domain.example.com`

- The Go application source code is located inside the **D0288-apps** Git repository in the **go-hello** directory.
- Before pushing the builder image to your Quay.io account, test the application locally on the **workstation** VM. Note the following when testing the builder image:
 - The application source code is located in the `~/D0288/labs/custom-s2i/test/test-app` folder.
 - Name the test application container image **s2i-go-app**.
 - Name the test container **go-test**.
 - Ensure that when you test the container you use a random user ID, such as 1234, to simulate running on an OpenShift cluster.
 - Bind the container port 8080 to local port 8080.
 - The application returns a greeting based on the URL that made the request. For example:

`http://localhost:8080/user1`, returns the following response:

Hello user1!. Welcome!

- When testing is complete, delete the **go-test** container before proceeding to the next step.
- Test building the go-hello application from source using your **s2i-do288-go** builder image.
- After you test the go-hello application running on OpenShift, customize the **run** script for the **s2i-do288-go** builder image by overriding it in the go-hello application source code.
- Commit and push your custom **run** script to the your Github repository used as a source for the application build. Then, start a new build and verify the new version of the go-hello application returns the greeting in Spanish, like:

Hola user1!. Bienvenido!

Steps

- The S2I scripts for the builder image are provided in the `/home/student/D0288/labs/custom-s2i/s2i/bin` folder. Review the `assemble`, `run`, and `usage` scripts.
- Edit the Dockerfile for the builder image to include an instruction to copy the S2I scripts to the appropriate location in the builder image. Add this new instruction immediately following the `TODO` comment already present in the file.
- Build the S2I builder image. Name the image `s2i-do288-go`.
- Build a Dockerfile for an application container image that combines the S2I builder image and the application source code locally on the **workstation** VM in the `/home/student/D0288/labs/custom-s2i/test/test-app` directory.
- Push the `s2i-do288-go` S2I builder image to your personal Quay.io account.
- Create an image stream called `s2i-do288-go` for the `s2i-do288-go` S2I builder image. Create the image stream in a project named `youruser-custom-s2i`.
- Enter your local clone of the `D0288-apps` Git repository and check out the `master` branch of the course's repository to ensure you start this exercise from a known good state:
- Create a new branch where you can save any changes you make during this exercise, and push it to Github:
- Deploy and test the go-hello application from your personal GitHub fork of the `D0288-apps` repository to the classroom OpenShift cluster. Be sure to reference the `custom-s2i` branch you created in the previous step when you deploy the application. The application echoes back a greeting to the resource requested by the HTTP request. For example, invoking the application with the following URL:
`http://greet-youruser-custom-s2i.apps.cluster.domain.example.com/user1`, returns the following response:

Hello user1!. Welcome!

- Customize the `run` script for the `s2i-do288-go` builder image in the application source. Change how the application is started by adding a `--lang es` argument at startup. This changes the default language used by the application.
Commit and push your changes to the branch that you used as the input source for your application build.
- Rebuild and test the application. The application should now respond to requests in Spanish. For example, invoking the application with the following URL:
`http://greet-youruser-custom-s2i.apps.cluster.domain.example.com/user1`, returns the following response:

Hola user1!. Bienvenido!

- Grade your work.
Run the following command on the **workstation** VM to verify that all tasks were accomplished:

```
[student@workstation ~]$ lab custom-s2i grade
```

- Clean up. Perform the following steps:

- 13.1. Delete the **youruser-custom-s2i** project in OpenShift.

```
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-custom-s2i
```

- 13.2. Delete all test containers created earlier to test the application locally.

```
[student@workstation ~]$ sudo podman rm go-test
```

- 13.3. Delete all the container images built during this lab on the **workstation** VM.

```
[student@workstation ~]$ sudo podman rmi -f \
> localhost/s2i-go-app \
> localhost/s2i-do288-go \
> registry.access.redhat.com/ubi8/ubi:8.0
...output omitted...
```

- 13.4. Delete the **s2i-do288-go** image from the external registry:

```
[student@workstation ~]$ sudo skopeo delete \
> docker://quay.io/${RHT_OCP4_QUAY_USER}/s2i-do288-go:latest
```

- 13.5. Log in to Quay.io using your personal free account.

Navigate to <http://quay.io> and click **Sign In** to provide your user credentials. Click **Sign in to Quay Container Registry** to log in to Quay.io.

- 13.6. On the Quay.io main menu, click **Repositories** and look for **s2i-do288-go**. The lock icon indicates it is a private repository that requires authentication for both pulls and pushes. Click **s2i-do288-go** to display the **Repository Activity** page.
- 13.7. On the **Repository Activity** page for the **s2i-do288-go** repository, scroll down and click the gear icon to display the **Settings** tab. Scroll down and click **Delete Repository**.
- 13.8. In the **Delete** dialog box, click **Delete** to confirm you want to delete the **s2i-do288-go** repository. After a few moments you are returned to the **Repositories** page. You can now sign out of Quay.io.

Finish

On the **workstation** VM, run the **lab custom-s2i finish** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab custom-s2i finish
```

This concludes the lab.

► Solution

Customizing Source-to-Image Builds

Performance Checklist

In this lab, you will create an S2I builder image for running applications based on the Go programming language.

Outcomes

You should be able to:

- Build a Go programming language S2I builder image based on the Universal Base Image for RHEL 8.
- Test the S2I builder image locally by creating a Dockerfile using the **s2i** tool and then building and running the resulting container image with Podman.
- Publish the builder image to your personal Quay.io account.
- Use the S2I builder image to deploy and test an application on an OpenShift cluster.
- Customize the **run** script to change the behavior of the application, and redeploy the application to test the changes.

Before You Begin

To perform this lab, you must have access to:

- A running OpenShift cluster.
- The **s2i** command-line tool.
- A fork of the Git repository containing the go-hello application source code.

Run the following command on the **workstation** VM to validate the prerequisites. This command also downloads helper files and solution files for the review lab:

```
[student@workstation ~]$ lab custom-s2i start
```

The setup script creates a folder called **custom-s2i** inside the **/home/student/D0288/labs** folder. This folder contains the template folder structure and files created by the **s2i create** command.

Requirements

In this lab, you need to deploy an application that is written in the Go programming language.

A small example Go application is provided for you to use to test your S2I builder image. The example Go application provides a simple HTTP API that responds to requests based on the resource requested in the HTTP request.

To complete the lab, build an S2I builder image for Go-based applications, and then test and deploy the example Go application on the classroom OpenShift cluster according to the following requirements:

- The S2I builder image must be named **s2i-do288-go**. The builder image must be available from your personal Quay.io account at:

`quay.io/youruser/s2i-do288-go`

- The image stream for the S2I builder image is named **s2i-do288-go**.
- The application name for OpenShift is **greet**.
- Both the image stream and the application must be created within a project named **youruser-custom-s2i**.

- The HTTP API for the application must be accessible at the following URL:

`http://greet-youruser-custom-s2i.apps.cluster.domain.example.com`

- The Go application source code is located inside the **D0288-apps** Git repository in the **go-hello** directory.
- Before pushing the builder image to your Quay.io account, test the application locally on the **workstation** VM. Note the following when testing the builder image:
 - The application source code is located in the `~/D0288/labs/custom-s2i/test/test-app` folder.
 - Name the test application container image **s2i-go-app**.
 - Name the test container **go-test**.
 - Ensure that when you test the container you use a random user ID, such as 1234, to simulate running on an OpenShift cluster.
 - Bind the container port 8080 to local port 8080.
 - The application returns a greeting based on the URL that made the request. For example:

`http://localhost:8080/user1`, returns the following response:

Hello user1!. Welcome!

- When testing is complete, delete the **go-test** container before proceeding to the next step.
- Test building the go-hello application from source using your **s2i-do288-go** builder image.
- After you test the go-hello application running on OpenShift, customize the **run** script for the **s2i-do288-go** builder image by overriding it in the go-hello application source code.
- Commit and push your custom **run** script to the your Github repository used as a source for the application build. Then, start a new build and verify the new version of the go-hello application returns the greeting in Spanish, like:

Hola user1!. Bienvenido!

Steps

- The S2I scripts for the builder image are provided in the `/home/student/D0288/labs/custom-s2i/s2i/bin` folder. Review the `assemble`, `run`, and `usage` scripts.
- Edit the Dockerfile for the builder image to include an instruction to copy the S2I scripts to the appropriate location in the builder image. Add this new instruction immediately following the `TODO` comment already present in the file.
Edit the Dockerfile at `~/D0288/labs/custom-s2i/Dockerfile` and add the following `COPY` instruction after the `TODO` comment. You can also copy the instruction from the provided solution Dockerfile at `~/D0288/solutions/custom-s2i/Dockerfile`:

```
COPY ./s2i/bin/ /usr/libexec/s2i
```

- Build the S2I builder image. Name the image `s2i-do288-go`.

- Create the S2I builder image:

```
[student@workstation ~]$ cd ~/D0288/labs/custom-s2i
[student@workstation custom-s2i]$ sudo podman build -t s2i-do288-go .
STEP 1: FROM registry.access.redhat.com/ubi8/ubi:8.0
Getting image source signatures
...output omitted...
Installed:
  golang-1.12.8-2.module+el8.1.0+4089+be929cf8.x86_64
  ...output omitted...
STEP 23: COMMIT s2i-do288-go
```



Note

You can safely ignore the following error in the build output: **ERRO[0000] HOSTNAME is not supported for OCI image format, hostname 1d561c58fd2b will be ignored. Must use `docker` format** This is a bug in the version of the Podman tool, and is a false error message, the container build proceeds without issue despite the error. More information can be found in this Bugzilla [https://bugzilla.redhat.com/show_bug.cgi?id=1661592]

- Verify that the builder image was created:

```
[student@workstation custom-s2i]$ sudo podman images
REPOSITORY                                     TAG      IMAGE ID      ...
localhost/s2i-do288-go                         latest   d1f856d10fa7  ...
...output omitted...
```

- Build a Dockerfile for an application container image that combines the S2I builder image and the application source code locally on the **workstation** VM in the `/home/student/D0288/labs/custom-s2i/test/test-app` directory.

- Create a directory named `s2i-go-app` where the `s2i` command can save the Dockerfile.

```
[student@workstation custom-s2i]$ mkdir /home/student/s2i-go-app
```

- 4.2. Use the **s2i build** command to produce a Dockerfile for the application container image:

```
[student@workstation custom-s2i]$ s2i build test/test-app/ \
> s2i-do288-go s2i-go-app \
> --as-dockerfile /home/student/s2i-go-app/Dockerfile
Application dockerfile generated in /home/student/s2i-go-app/Dockerfile
```

- 4.3. Build a test container image from the generated Dockerfile.

```
[student@workstation custom-s2i]$ cd ~/s2i-go-app
[student@workstation s2i-go-app]$ sudo podman build -t s2i-go-app .
STEP 1: FROM s2i-do288-go
STEP 2: LABEL "io.k8s.display-name"="s2i-go-app"
...output omitted...
STEP 15: COMMIT s2i-go-app
```



Note

You can safely ignore the following error in the build output: **ERRO[0000]** **HOSTNAME is not supported for OCI image format, hostname 1d561c58fd2b will be ignored. Must use `docker` format** This is a bug in the version of the Podman tool, and is a false error message, the container build proceeds without issue despite the error. More information can be found in this Bugzilla [https://bugzilla.redhat.com/show_bug.cgi?id=1661592]

- 4.4. Verify that the application container image was created:

```
[student@workstation s2i-go-app]$ sudo podman images
REPOSITORY                      TAG      IMAGE ID      ...
localhost/s2i-go-app             latest   7d3d8f894f2f  ...
...output omitted...
```

- 4.5. Test the application container image locally on the **workstation** VM. Run the container with the **-u** flag to simulate running as a random user on an OpenShift cluster. You can copy the command or run it directly from the **~/DO288/labs/custom-s2i/local-test.sh** script:

```
[student@workstation s2i-go-app]$ sudo podman run --name go-test -u 1234 \
> -p 8080:8080 -d s2i-go-app
18b903cfaae4...output omitted...
```

- 4.6. Verify that the container started successfully:

```
[student@workstation s2i-go-app]$ sudo podman ps
CONTAINER ID  IMAGE                  COMMAND
18b903cfaae4  localhost/s2i-go-app:latest  /bin/sh -c /usr/lib...
```

- 4.7. Use the **curl** command to test the application:

```
[student@workstation s2i-go-app]$ curl http://localhost:8080/user1
Hello user1!. Welcome!
```

- 4.8. Stop the **go-test** application container:

```
[student@workstation s2i-go-app]$ sudo podman stop go-test
18b903cfaae4...output omitted...
[student@workstation s2i-go-app]$ cd ~
```

5. Push the **s2i-do288-go** S2I builder image to your personal Quay.io account.

- 5.1. Load your classroom environment configuration.

Run the following command to load the environment variables created in the first guided exercise:

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

- 5.2. Log in to your Quay.io account using the **podman** command. Enter your Quay.io password when you are prompted.

```
[student@workstation ~]$ sudo podman login \
> -u ${RHT_OCP4_QUAY_USER} quay.io
Password:
Login Succeeded!
```

- 5.3. Use the **skopeo copy** command to publish the S2I builder image to your Quay.io account.

```
[student@workstation ~]$ sudo skopeo copy \
> containers-storage:localhost/s2i-do288-go \
> docker://quay.io/${RHT_OCP4_QUAY_USER}/s2i-do288-go
...output omitted...
Writing manifest to image destination
Storing signatures
```

6. Create an image stream called **s2i-do288-go** for the **s2i-do288-go** S2I builder image. Create the image stream in a project named **youruser-custom-s2i**.

- 6.1. Log in to OpenShift and create a new project. Prefix the project's name with your developer user's name.

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful.
...output omitted...
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-custom-s2i
Now using project "youruser-custom-s2i" on server "https://
api.cluster.domain.example.com:6443".
```

- 6.2. Log in to your personal Quay.io account using Podman, this time without the **sudo** command, so that you can export the **auth.json** file to use it in a pull secret in the next step. You need to enter your password again.

```
[student@workstation ~]$ podman login -u ${RHT_OCP4_QUAY_USER} quay.io
Password:
Login Succeeded!
```

- 6.3. Create a secret from the container registry API access token that was stored by Podman.

You can also execute or cut and paste the following **oc create secret** command from the **create-secret.sh** script in the **/home/student/D0288/labs/custom-s2i** directory.

```
[student@workstation ~]$ oc create secret generic quayio \
> --from-file .dockerconfigjson=${XDG_RUNTIME_DIR}/containers/auth.json \
> --type=kubernetes.io/dockerconfigjson
secret/quayio created
```

- 6.4. Link the new secret to the **builder** service account.

```
[student@workstation ~]$ oc secrets link builder quayio
```

- 6.5. Create an image stream by importing the S2I builder image from the private classroom registry:

```
[student@workstation ~]$ oc import-image s2i-do288-go \
> --from quay.io/${RHT_OCP4_QUAY_USER}/s2i-do288-go \
> --confirm
imagestream.image.openshift.io/s2i-do288-go imported
...output omitted...
```

- 6.6. Verify that the image stream was created:

```
[student@workstation ~]$ oc get is
NAME          IMAGE REPOSITORY           ...output omitted...
s2i-do288-go  ...youruser-custom-s2i/s2i-do288-go ...output omitted...
```

7. Enter your local clone of the **D0288-apps** Git repository and check out the **master** branch of the course's repository to ensure you start this exercise from a known good state:

```
[student@workstation ~]$ cd D0288-apps
[student@workstation D0288-apps]$ git checkout master
...output omitted...
```

8. Create a new branch where you can save any changes you make during this exercise, and push it to Github:

```
[student@workstation D0288-apps]$ git checkout -b custom-s2i
Switched to a new branch 'custom-s2i'
[student@workstation D0288-apps]$ git push -u origin custom-s2i
...output omitted...
* [new branch]      custom-s2i -> custom-s2i
Branch custom-s2i set up to track remote branch custom-s2i from origin.
[student@workstation D0288-apps]$ cd ~
```

9. Deploy and test the go-hello application from your personal GitHub fork of the **D0288-apps** repository to the classroom OpenShift cluster. Be sure to reference the **custom-s2i** branch you created in the previous step when you deploy the application. The application echoes back a greeting to the resource requested by the HTTP request. For example, invoking the application with the following URL:

`http://greet-youruser-custom-s2i.apps.cluster.domain.example.com/user1`, returns the following response:

```
Hello user1!. Welcome!
```

- 9.1. Create a new application from source code in GitHub and using the **s2i-do288-go** image stream:

```
[student@workstation ~]$ oc new-app --as-deployment-config --name greet \
> s2i-do288-go~https://github.com/${RHT_OCP4_GITHUB_USER}/D0288-apps#custom-s2i \
> --context-dir=go-hello
--> Found image a29d3e7 (About an hour old) in image stream "youruser-custom-s2i/
s2i-do288-go" under tag "latest" for "s2i-do288-go"

    Go programming language S2I builder image for D0288
...output omitted...
--> Creating resources ...
    imagestream.image.openshift.io "greet" created
    buildconfig.build.openshift.io "greet" created
    deploymentconfig.apps.openshift.io "greet" created
    service "greet" created
--> Success
...output omitted...
```

- 9.2. View the build logs. Wait until the build finishes and the application container image is pushed to the OpenShift registry:

```
[student@workstation ~]$ oc logs -f bc/greet
Cloning "https://github.com/youruser/D0288-apps" ...
...output omitted...
--> Installing application source...
--> Building application from source...
...output omitted...
Push successful
```

- 9.3. Wait until the application is deployed. View the status of the application pod. The application pod must be in the **Running** state:

```
[student@workstation ~]$ oc get pods
NAME        READY   STATUS    RESTARTS   AGE
greet-1-6zx8p  1/1     Running   0          35s
greet-1-build  0/1     Completed  0          2m30s
greet-1-deploy 0/1     Completed  0          43s
```

9.4. Expose the application for external access by using a route:

```
[student@workstation ~]$ oc expose svc greet
route.route.openshift.io/greet exposed
```

9.5. Obtain the route URL using the **oc get route** command:

```
[student@workstation ~]$ oc get route/greet -o jsonpath='{.spec.host}{"\n"}'
greet-youruser-custom-s2i.apps.cluster.domain.example.com
```

9.6. Test the application using the route URL you obtained in the previous step:

```
[student@workstation ~]$ curl \
> http://greet-${RHT_OCP4_DEV_USER}-custom-s2i.${RHT_OCP4_WILDCARD_DOMAIN}/user1
Hello user1!. Welcome!
```

- 10.** Customize the **run** script for the **s2i-do288-go** builder image in the application source. Change how the application is started by adding a **--lang es** argument at startup. This changes the default language used by the application.
Commit and push your changes to the branch that you used as the input source for your application build.

10.1. The S2I scripts can be customized in the **.s2i/bin** directory of the application source located in the **DO288-apps/go-hello** directory. Create the directory:

```
[student@workstation ~]$ mkdir -p ~/DO288-apps/go-hello/.s2i/bin
```

10.2. Copy the **run** script in the S2I builder image from **~/DO288/labs/custom-s2i/s2i/bin/run**:

```
[student@workstation ~]$ cp ~/DO288/labs/custom-s2i/s2i/bin/run \
> ~/DO288-apps/go-hello/.s2i/bin/
```

10.3. Customize the **run** script and add the **--lang es** option to the application startup. You can also copy the complete script from the **~/DO288/solutions/custom-s2i/s2i/bin/run.es** file:

```
...output omitted...
echo "Starting app with lang option 'es'..."
exec /opt/app-root/app --lang es
```

10.4. Commit the changes to Git:

```
[student@workstation ~]$ cd ~/D0288-apps/go-hello
[student@workstation go-hello]$ git add .
[student@workstation go-hello]$ git commit -m "Customized run script"
...output omitted...
[student@workstation go-hello]$ git push
...output omitted...
[student@workstation go-hello]$ cd ~
```

11. Rebuild and test the application. The application should now respond to requests in Spanish. For example, invoking the application with the following URL:
<http://greet-youruser-custom-s2i.apps.cluster.domain.example.com/> user1, returns the following response:

```
Hola user1!. Bienvenido!
```

- 11.1. Start a new build for the application:

```
[student@workstation ~]$ oc start-build greet
build.build.openshift.io/greet-2 started
```

- 11.2. Follow the build log and verify that a new container image is created and pushed to the OpenShift internal registry:

```
[student@workstation ~]$ oc logs -f bc/greet
Cloning "https://github.com/youruser/D0288-apps" ...
Commit: 0023a1b02342b633aad49e58a2eeba11dff33c3d (customized run script)
...output omitted...
Push successful
```

- 11.3. Wait for the application pod to be deployed. The pod must be in the **Running** state. Verify the status of the application pod:

```
[student@workstation ~]$ oc get pods
NAME        READY   STATUS    RESTARTS   AGE
...output omitted...
greet-2-build  0/1     Completed   0          2m14s
greet-2-deploy 1/1     Running    0          44s
greet-2-rpwb4  1/1     Running    0          34s
```

- 11.4. Test the application using the route URL you obtained from Step 9.5:

```
[student@workstation ~]$ curl \
> http://greet-${RHT_OCP4_DEV_USER}-custom-s2i.${RHT_OCP4_WILDCARD_DOMAIN}/user1
Hola user1!. Bienvenido!
```

12. Grade your work.

Run the following command on the **workstation** VM to verify that all tasks were accomplished:

```
[student@workstation ~]$ lab custom-s2i grade
```

13. Clean up. Perform the following steps:

- 13.1. Delete the **youruser-custom-s2i** project in OpenShift.

```
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-custom-s2i
```

- 13.2. Delete all test containers created earlier to test the application locally.

```
[student@workstation ~]$ sudo podman rm go-test
```

- 13.3. Delete all the container images built during this lab on the **workstation** VM.

```
[student@workstation ~]$ sudo podman rmi -f \
> localhost/s2i-go-app \
> localhost/s2i-do288-go \
> registry.access.redhat.com/ubi8/ubi:8.0
...output omitted...
```

- 13.4. Delete the **s2i-do288-go** image from the external registry:

```
[student@workstation ~]$ sudo skopeo delete \
> docker://quay.io/${RHT_OCP4_QUAY_USER}/s2i-do288-go:latest
```

- 13.5. Log in to Quay.io using your personal free account.

Navigate to <http://quay.io> and click **Sign In** to provide your user credentials. Click **Sign in to Quay Container Registry** to log in to Quay.io.

- 13.6. On the Quay.io main menu, click **Repositories** and look for **s2i-do288-go**. The lock icon indicates it is a private repository that requires authentication for both pulls and pushes. Click **s2i-do288-go** to display the **Repository Activity** page.
- 13.7. On the **Repository Activity** page for the **s2i-do288-go** repository, scroll down and click the gear icon to display the **Settings** tab. Scroll down and click **Delete Repository**.
- 13.8. In the **Delete** dialog box, click **Delete** to confirm you want to delete the **s2i-do288-go** repository. After a few moments you are returned to the **Repositories** page. You can now sign out of Quay.io.

Finish

On the **workstation** VM, run the **lab custom-s2i finish** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab custom-s2i finish
```

This concludes the lab.

Summary

In this chapter, you learned:

- An S2I builder image is a specialized container image that developers use to produce application container images. Builder images include base operating system libraries, language runtimes, frameworks, and application dependencies, as well as Source-to-Image tools and utilities.
- An S2I builder image provides S2I scripts by default. These S2I scripts can be overridden in the application source code by adding S2I scripts to the **.s2i/bin** directory.
- The **s2i** command-line tool is used to build and test S2I builder images outside of OpenShift.
- In RHEL 8 or OpenShift 4 environments, where Docker is not available by default, use the **--as-dockerfile** option for the **s2i build** command. This causes the command to produce a Dockerfile and supporting directories that you can build with Podman to test your S2I builder image.

Chapter 6

Creating Applications from OpenShift Templates

Goal

Describe the elements of a template and create a multicontainer application template.

Objectives

- Describe the elements of an OpenShift template.
- Build a multicontainer application from a custom template.

Sections

- Describing the Elements of an OpenShift Template (and Quiz)
- Creating a Multicontainer Template (and Guided Exercise)

Lab

Creating Applications from OpenShift Templates

Describing the Elements of an OpenShift Template

Objectives

After completing this section, you should be able to describe the elements of an OpenShift template.

Describing a Template

An OpenShift *template* is a YAML or JSON file consisting of a set of OpenShift resources. Templates define *parameters* that are used to customize the resource configuration. OpenShift processes templates by replacing parameter references with values and creating a customized set of resources.

OpenShift templates offer similar functionality to Kubernetes Helm charts. You can use both Helm charts or OpenShift templates to manage your multi-container applications similarly. However, Helm charts are not in the scope of this course.

A template is useful when you want to deploy a set of resources as a single unit, rather than deploying them individually. Example use cases for when to use a template include:

- An independent software vendor (ISV) provides a template to deploy their product on OpenShift. The template contains configuration details of the containers that make up the product, along with a deployment configuration such as the number of replicas, services and routes, persistent storage configuration, health checks, and resource limits for compute resources such as CPU, memory, and I/O.
- Your multtier application consists of a number of separate components, such as a web server, an application server, and a database, and you would like to deploy these components together as a single unit on OpenShift to simplify the process of deploying the application in a staging environment. This will allow your QA and acceptance testing teams to rapidly provision applications for testing.

Template Syntax

The syntax of an OpenShift template follows the general syntax of an OpenShift resource, but with the **objects** attribute in place of the **spec** attribute. A template usually includes **parameters** and **labels** attributes, as well as annotations in its metadata.

The following listing shows a sample template definition in YAML format and illustrates the main syntax elements. As with any OpenShift resource, you can also define templates in JSON syntax:

```
apiVersion: template.openshift.io/v1
kind: Template 1
metadata:
  name: mytemplate
  annotations:
    description: "Description" 2
objects: 3
- apiVersion: v1
  kind: Pod
```

```

metadata:
  name: myapp
spec:
  containers:
    - env:
        - name: MYAPP_CONFIGURATION
          value: ${MYPARAMETER} ④
      image: myorganization/myapplication
      name: myapp
      ports:
        - containerPort: 80
          protocol: TCP
  parameters: ⑤
    - description: Myapp configuration data
      name: MYPARAMETER
      required: true
  labels: ⑥
    mylabel: myapp

```

- ① Template Resource type
- ② Optional annotations for use by OpenShift tools
- ③ Resource list
- ④ Reference to a template parameter
- ⑤ Parameter list
- ⑥ Label list

The resource list of a template usually includes other resources like build configurations, deployment configurations, persistent volume claims (PVC), services, and routes.

Any attribute in the resource list can reference the value of any parameter.

You can define custom labels for a template. OpenShift adds these labels to all resources created by the template.

When a template defines multiple resources, it is essential to consider the order in which these resources are defined to accommodate dependencies between resources. OpenShift does not report an error if a resource references a dependent resource that does not exist.

A process that is triggered by a resource might fail if it starts before a dependent resource. One example is a build configuration that references an image stream as the output image, and your template defines that image stream after the build configuration. In this scenario, it is possible that the build configuration starts a build before the image stream exists.

Defining Template Parameters

The previous example template included the definition of a required parameter. Both optional and required parameters can provide default values. For example:

```

parameters:
  - description: Myapp configuration data
    name: MYPARAMETER
    value: /etc/myapp/config.ini

```

OpenShift can generate random default values for parameters. This is useful for secrets and passwords:

```
parameters:
- description: ACME cloud provider API key
  name: APIKEY
  generate: expression
  from:"[a-zA-Z0-9]{12}"
```

The syntax for generated values is a subset of the Perl regular expression syntax. See the references at the end of this section for the full template parameter expression syntax.

Adding a Template to OpenShift

To add a template to OpenShift, use either the **oc create** command or the web console (using the import YAML page) to create the template resource from the template definition file, the same way you would for any other kind of resource.

It is a common practice to create projects that hold only templates because OpenShift allows users to easily share templates between multiple users and projects. These projects usually include other potentially shared resources, such as image streams.

A default installation of Red Hat OpenShift Container Platform provides several templates in the **openshift** project. All OpenShift cluster users have read access to the **openshift** project, but only cluster administrators have permission to create or delete templates in this project.

Browsing Templates with the OpenShift CLI

OpenShift provides a number of templates in the **openshift** namespace by default. You can create applications from these templates, as well as customize the provided templates to suit the needs of your application.

To view the list of provided templates, use the **oc get templates** command:

```
[user@host ~]$ oc get templates -n openshift
```

To view the list of parameters provided by the template as well as a brief description, use the **oc process** command. For example, to view the parameters for a Node.js and MongoDB template provided by OpenShift:

```
[user@host ~]$ oc process --parameters -n openshift nodejs-mongodb-example
```

To export the template as a YAML file to use as a base for your own custom template, use the **oc get** command. For example, to export the **nodejs-mongodb-example** template provided by OpenShift in YAML format:

```
[user@host ~]$ oc get template nodejs-mongodb-example -o yaml -n openshift
```

You can also use the **oc describe template** command to get a more detailed description of a template. For example, to describe the **nodejs-mongodb-example** template provided by OpenShift:

```
[user@host ~]$ oc describe template nodejs-mongodb-example -n openshift
Name: nodejs-mongodb-example ①
Namespace: openshift
```

```

Created: 6 days ago
Labels: samples.operator.openshift.io/managed=true
...output omitted...
Annotations: iconClass=icon-nodejs
    openshift.io/display-name=Node.js + MongoDB (Ephemeral)
    openshift.io/documentation-url=https://github.com/sclorg/nodejs-ex
...output omitted...
Parameters: ②
  Name: NAME
  Display Name: Name
  Description: The name assigned to all of the frontend objects defined in
this template.
  Required: true
  Value: nodejs-mongodb-example

  Name: NAMESPACE
  Display Name: Namespace
  Description: The OpenShift Namespace where the ImageStream resides.
  Required: true
  Value: openshift

  Name: NODEJS_VERSION
  Display Name: Version of NodeJS Image
  Description: Version of NodeJS image to be used (6, 8, or latest).
  Required: true
  Value: 8
...output omitted...
Object Labels: app=nodejs-mongodb-example,template=nodejs-mongodb-example ③

Message: The following service(s) have been created in your project: ${NAME},
${DATABASE_SERVICE_NAME}.

For more information about using this template, including OpenShift
considerations, see https://github.com/sclorg/nodejs-ex/blob/master/README.md.

Objects: ④
  Secret ${NAME}
  Service ${NAME}
  Route ${NAME}
  ImageStream ${NAME}
  BuildConfig ${NAME}
  DeploymentConfig ${NAME}
  Service ${DATABASE_SERVICE_NAME}
  DeploymentConfig ${DATABASE_SERVICE_NAME}

```

- ① The name of the template.
- ② A list of parameters the template offers, including whether they are required and any default values.
- ③ A list of labels that OpenShift applies to all resources it creates from the template.
- ④ A summary of all the resources that OpenShift creates from the template.



References

Further information is available in the *Using Templates* chapter in the documentation for Red Hat OpenShift Container Platform 4.5; at https://access.redhat.com/documentation/en-us/openshift_container_platform/4.5/html/images/using-templates

► Quiz

Describing the Elements of an OpenShift Template

Choose the correct answers to the following questions:

► 1. What is the purpose of an OpenShift template?

- a. To describe the resource configuration for a single container.
- b. To encapsulate a set of OpenShift resources for reuse.
- c. To provide custom configurations for an OpenShift cluster.
- d. To customize the appearance of the web console.

► 2. Which two of the following approaches define a template parameter as optional?

(Choose two.)

- a. Set the **required** attribute to **false**.
- b. Set the **required** attribute to **not**.
- c. Omit the **required** attribute.
- d. Set the **required** attribute to **expression**.
- e. Set the **from** attribute to a regular expression.

► 3. Which of the following commands returns the number of parameters and objects provided by a template?

- a. **oc export**.
- b. **oc describe**.
- c. **oc create**.
- d. **oc get**.
- e. None of the above.

► 4. Which three statements describe valid locations to insert template parameter references? (Choose three.)

- a. As the name of an environment variable inside a pod resource.
- b. As the key for a label in all resources created by the template.
- c. As the value for the **host** attribute in a route resource.
- d. As the value for an annotation in the template.
- e. As the value of an environment variable inside a pod resource.

► Solution

Describing the Elements of an OpenShift Template

Choose the correct answers to the following questions:

► 1. What is the purpose of an OpenShift template?

- a. To describe the resource configuration for a single container.
- b. To encapsulate a set of OpenShift resources for reuse.
- c. To provide custom configurations for an OpenShift cluster.
- d. To customize the appearance of the web console.

► 2. Which two of the following approaches define a template parameter as optional?

(Choose two.)

- a. Set the **required** attribute to **false**.
- b. Set the **required** attribute to **not**.
- c. Omit the **required** attribute.
- d. Set the **required** attribute to **expression**.
- e. Set the **from** attribute to a regular expression.

► 3. Which of the following commands returns the number of parameters and objects provided by a template?

- a. **oc export**.
- b. **oc describe**.
- c. **oc create**.
- d. **oc get**.
- e. None of the above.

► 4. Which three statements describe valid locations to insert template parameter references? (Choose three.)

- a. As the name of an environment variable inside a pod resource.
- b. As the key for a label in all resources created by the template.
- c. As the value for the **host** attribute in a route resource.
- d. As the value for an annotation in the template.
- e. As the value of an environment variable inside a pod resource.

Creating a Multicontainer Template

Objectives

After completing this section, you should be able to build a multicontainer application from a custom template.

Creating a Template from Application Resources

A typical scenario for using OpenShift templates is to deploy an application that requires multiple sets of pods, each one with a deployment configuration, services, and other auxiliary resources such as secrets and persistent volume claims.

Consider an application that connects to a database to store and manage data. A template for this application would include:

- A deployment configuration and a service for the application, as well as another deployment configuration and service for the database.
- Two image streams to point to the container images: one for the application, and another for the database.
- A secret for database access credentials.
- A persistent volume claim for storing the database data.
- A route for external access to the application.

You can create the required template in several different ways. The most common approaches are discussed in the next section.

Manually Creating Resource Files

If you are comfortable with the syntax of YAML or JSON files, and you are familiar with the attributes for each of the OpenShift resource types, you can create templates by hand.

The general process for this approach is described below.

1. Create a basic template skeleton with basic attributes such as a name, tags, and other metadata information and declare it as an OpenShift **Template** resource.
2. Look at existing templates that come installed with OpenShift by default, and customize them to your needs by cutting and pasting relevant resources for your application.
3. To further customize the template, use the **oc explain** and **oc api-resources** commands to view the attributes for each resource type that you want to customize and then add it to your template.

Concatenating Resource Files

The **oc new-app** command can create a resource definition file, instead of creating resources in the current project, by using the **-o** option. You can concatenate multiple resource definition files inside the resources list of a skeleton template definition file.

The general process for this approach is described below.

1. Create a basic template skeleton with basic attributes such as a name, tags, and other metadata information and declare it as an OpenShift **Template** resource.
2. Use the **oc new-app** command with the **-o** option to export the resource configuration to a file.
3. Clean the runtime attributes from the generated resource file.
4. Cut and paste the resource list into a skeleton template file.

Exporting Existing Resources

The **oc get** command can create a resource definition file by using the **-o** and **--export** options together. The **-o** option takes either **yaml** or **json** as a parameter, and exports the resource definition in YAML or JSON format respectively. After you export a set of resources to a template file, you can add annotations and parameters as desired.

The general process for this approach is described below.

1. Create a basic template skeleton with basic attributes such as a name, tags, and other metadata information and declare it as an OpenShift **Template** resource.
2. Use the **oc get** command with the **-o --export** options to export the resource configuration to a file.
3. Clean the runtime attributes from the generated resource file.
4. Cut and paste the resource list into a skeleton template file.

When you use the **oc get** command to export resources, select only the necessary resource types. Include only resources that you would need to deploy an application. Do not include derived resources such as builds, deployments, replication controllers, or pods.

The order in which you list the resources in the **oc get** command is important. You need to export any dependent resources first, and then the resources that depend on them. For example, you need to export image streams before the build configurations and deployment configurations that reference those image streams.

The following example creates a YAML file containing different types of resources in the current project:

```
[user@host ~]$ oc get -o yaml --export is,bc,dc,svc,route > mytemplate.yaml
```

Depending on your needs, add more resource types to the previous command. For example, add **secret** before **bc** and **dc**. It is safe to add **pvc** to the end of the list of resource types because a deployment waits for a persistent volume claim to bind.

Cleaning and Editing Templates

The **oc get** and **oc new-app** commands do not generate resource definitions that are ready for use in a template. These resource definitions contain runtime information that a template does not need, and some of it could prevent the template from working at all. Examples of runtime information are attributes such as **status**, **creationTimeStamp**, **image**, and **uid**, as well as most annotations that start with the **openshift.io/generated-by** prefix.

Some resource types, such as secrets, require special handling. It is not possible to initialize key values inside the **data** attribute using template parameters. The **data** attribute from a secret resource needs to be replaced by the **stringData** attribute, and all key values need to be unencoded.

Another example is image stream resources. OpenShift caches container images from external registries using the internal registry, and the exported image stream points to the internal registry. If you use this image stream resource in a template, it fails because the container image does not exist in the current project. You need to change the image stream to point to the external registry.

Describing OpenShift Resource Types

To create a template from scratch, or to clean a resource file generated by the **oc get** command, you need to know which attributes contain runtime information. This is explained in the Red Hat OpenShift Container Platform product documentation. See the references at the end of this section. Another way is to use the **oc explain** command.

With a resource type as an argument, the **oc explain** command lists the top-level attributes for that resource type:

```
[user@host ~]$ oc explain routes
DESCRIPTION:
A route allows developers to expose services through an HTTP(S) aware load
balancing and proxy layer via a public DNS entry.
...output omitted...
FIELDS:
  apiVersion <string>
    ...output omitted...

  kind <string>
    ...output omitted...

  metadata <Object>
    Standard object metadata.

  spec <Object> -required-
    spec is the desired state of the route
  ...output omitted...
```

Most resource types have many levels of attributes. Use dot notation to request attributes from lower levels:

```
[user@host ~]$ oc explain routes.spec
RESOURCE: spec <Object>

DESCRIPTION:
spec is the desired state of the route
...output omitted...
FIELDS:
  to <Object> -required-
    to is an object the route should use as the primary backend.
    ...output omitted...

  wildcardPolicy <string>
```

```
Wildcard policy if any for the route. Currently only 'Subdomain' or 'None'
is allowed.
...output omitted...
```

Use the **oc api-resources** command for a complete list of supported resources in the OpenShift instance.

Creating an Application from a Template

You can deploy an application directly from a template resource definition file. The **oc new-app** command and the **oc process** command use the template file as input and process it to apply parameters and create resources.

You can also pass a template file to the **oc create** command to create a template resource in OpenShift. If the template is meant to be reused by multiple developers, it is better to create the template resource in a shared project. If the template is meant to be used by a single deployment, it is better to keep it in a file.

Whereas the **oc new-app** command creates resources from the template, the **oc process** command creates a resource list from the template. You need to either save the resource list generated by the **oc process** command to a file or pass it as input to the **oc create** command to create the resources from the list.

For both the **oc new-app** command and the **oc process** command, each parameter value has to be provided using a different **-p** option. Another option that both commands accept is the **-l** option, which adds a label to all resources created from the template.

The following is an example **oc new-app** command that deploys an application from a template file:

```
[user@host ~]$ oc new-app --file mytemplate.yaml -p PARAM1=value1 \
> -p PARAM2=value2
```

The following is an example **oc process** command that deploys an application from a template file:

```
[user@host ~]$ oc process -f mytemplate.yaml -p PARAM1=value1 \
> -p PARAM2=value2 > myresourcelist.yaml
```

The file generated by the previous example would then be given to the **oc create** command:

```
[user@host ~]$ oc create -f myresourcelist.yaml
```

You can combine the previous two examples using a pipe:

```
[user@host ~]$ oc process -f mytemplate.yaml -p PARAM1=value1 \
> -p PARAM2=value2 | oc create -f -
```

Red Hat recommends using the **oc new-app** command rather than the **oc process** command. You can use the **oc process** command with the **-f** option to list only those parameters defined by the specified template:

```
[user@host ~]$ oc process -f mytemplate.yaml --parameters
```



Note

There is no section in the OpenShift 4.5 web console to list or view templates. Nevertheless, you can list available templates in **Administrator** → **Home** → **Search** and **Developer** → **Search** pages.

You cannot create applications from custom templates in the OpenShift 4.5 web console by default. Cluster administrators can install additional templates into the Developer Catalog. Templates in the Developer Catalog can be used to create new applications.



References

Further information about creating and using templates is available in the *Using Templates* chapter of the Red Hat OpenShift Container Platform documentation at https://access.redhat.com/documentation/en-us/openshift_container_platform/4.5/html/images/using-templates

► Guided Exercise

Creating a Multicontainer Template

In this exercise, you will create an OpenShift template that deploys a multicontainer application. The template includes an HTTP API, a database, and persistent storage.

Outcomes

You should be able to:

- Create a template from a running application using the **oc get** command.
- Clean the template to remove runtime information.
- Add parameters to the template.
- Create a new application from the template using the OpenShift command-line tool (**oc**).

Before You Begin

To perform this exercise, ensure you have access to:

- A running OpenShift cluster.
- The PHP 7.2 S2I builder image, and the MySQL 5.7 database image required by the template.
- The sample application (quotes) in the Git repository.

Run the following command on the **workstation** VM to validate the prerequisites, provision persistent storage, deploy the multicontainer application to the **youruser-quotes-dev** project, and download the required files to complete this exercise:

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

To review how the application is deployed, refer to the **new-app-db.sh**, **add-vol.sh**, **new-app-php.sh**, and **add-route.sh** scripts in the **~/DO288/labs/create-template** folder.

► 1. Inspect the quotes application deployed to the **youruser-quotes-dev** project.

1.1. Load your classroom environment configuration.

Run the following command to load the environment variables created in the first guided exercise:

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

1.2. Log in to OpenShift using your developer user account:

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful.
...output omitted...
```

- 1.3. Set **youruser-quotes-dev** as your current active project:

```
[student@workstation ~]$ oc project ${RHT_OCP4_DEV_USER}-quotes-dev
```

- 1.4. The database and application are already deployed. Wait for the build to complete and for both the database and application pods to be ready and running:

```
[student@workstation ~]$ oc status
In project youruser-quotes-dev on server
https://api.cluster.domain.example.com:6443

http://quotesapi-youruser-quotes-dev.apps.cluster.domain.example.com to pod port
  8080-tcp (svc/quotesapi)
dc/quotesapi deploys istag/quotesapi:latest <-
bc/quotesapi source builds https://github.com/youruser/D0288-apps on openshift/
php:7.2
  deployment #1 deployed 28 seconds ago - 1 pod

svc/quotesdb - 172.30.239.22:3306
dc/quotesdb deploys openshift/mysql:5.7
  deployment #2 deployed about a minute ago - 1 pod
  deployment #1 failed about a minute ago: newer deployment was found running
...output omitted...
```

The previous output shows that the database is deployed from a container image, and the application is deployed from source code.

- 1.5. Verify that the project includes a persistent volume claim:

```
[student@workstation ~]$ oc get pvc
NAME          STATUS  VOLUME      CAPACITY  ACCESS MODES  ...
quotesdb-claim  Bound   pvc-df1c...  1Gi       RWO        ...
```

- 1.6. Verify that the project includes a route:

```
[student@workstation ~]$ oc get route/quotesapi -o jsonpath='{.spec.host}{"\n"}'
quotesapi-youruser-quotes-dev.apps.cluster.domain.example.com
```

Do not try to test the application. It will fail because the database is not initialized. The only use of the **youruser-quotes-dev** project is to export its resources to a template.

- 2. Create the template definition file. Start with a basic template definition file and then export the required resources one by one, clean out the runtime information, and then copy the cleaned resource configuration to the template.

Create a new file called **quotes-template-clean.yaml** in the **/home/student** directory. Add the following YAML snippet to declare this resource definition file as an OpenShift template:

```
apiVersion: template.openshift.io/v1
kind: Template
metadata:
  name: quotes
  annotations:
    openshift.io/display-name: Quotes Application
    description: The Quotes application provides an HTTP API that returns a
      random, funny quote.
    iconClass: icon-php
    tags: php,mysql
objects:
```

You can also copy the YAML snippet from the [~/D0288/solutions/create-template/new-template.yaml](#) file.



Warning

Ensure that the **description** attribute value is in a single continuous line with no line breaks. You will get errors for improperly formatted YAML files when the template is parsed later in the exercise.

- 3. Export the resources in the project one by one, starting with the image stream resources.

- 3.1. Export the image stream:

```
[student@workstation ~]$ oc get -o yaml --export is > /tmp/is.yaml
```

- 3.2. Make a copy of the exported file and clean the runtime attributes from it:

```
[student@workstation ~]$ cp /tmp/is.yaml /tmp/is-clean.yaml
```

- 3.3. Clean the runtime attributes from the **/tmp/is-clean.yaml** file.

A copy of the cleaned file is available at [~/D0288/labs/create-template/is-clean.yaml](#). Compare your cleaned file against this version and make the necessary edits to make them the same.

Remove the first two lines from the file:

```
apiVersion: v1
items:
```

- 3.4. There are two image stream resources in the exported file. The first is for the quotesapi application, and the second for the quotesdb database. Remove the entire image stream resource for the quotesdb database. The deployment configuration for the database will create the image stream automatically.

Starting with the second occurrence of the image stream resource, that is:

```
- apiVersion: image.openshift.io/v1
kind: ImageStream
metadata:
  annotations:
    openshift.io/image.dockerRepositoryCheck: "2019-06-21T02:54:22Z"
  creationTimestamp: "2019-06-21T02:54:22Z"
  generation: 2
  name: quotesdb

...output omitted...
```

Delete all lines beginning from the above listed snippet to the end of the file.

- 3.5. The following steps involve cleaning the image stream resource for the quotesapi application only.
Remove the **openshift.io/generated-by**, **creationTimestamp**, **generation**, **namespace**, **resourceVersion**, **selfLink**, and **uid** attributes.
Remove the **managedFields**, and **status** attributes including their child attributes.

▶ 4. Export and clean the build configuration.

- 4.1. Export the build configuration:

```
[student@workstation ~]$ oc get -o yaml --export bc > /tmp/bc.yaml
```

- 4.2. Make a copy of the exported file and clean the runtime attributes from it:

```
[student@workstation ~]$ cp /tmp/bc.yaml /tmp/bc-clean.yaml
```

Clean the runtime attributes from the **/tmp/bc-clean.yaml** file.

A copy of the cleaned file is available at **~/DO288/labs/create-template/bc-clean.yaml**. Compare your cleaned file against this version and make the necessary edits to make them the same.

Ignore differences in the **secret** and **uri** attributes. These values will be different for you.

- 4.3. Remove the first two lines of the file:

```
apiVersion: v1
items:
```

- 4.4. Remove the **openshift.io/generated-by**, **creationTimestamp**, **generation**, **namespace**, **resourceVersion**, **selfLink**, and **uid** attributes.
- 4.5. Remove the **managedFields** attribute and all its child attributes.
- 4.6. Remove the **namespace** attribute that refers to the **youruser-quotes-dev** project. Do not remove the **namespace: openshift** reference under the **sourceStrategy** attribute lower down in the file.
- 4.7. Remove the **lastTriggeredImageID** attribute under **imageChange**.
- 4.8. Remove the **status** attribute and all its child attributes at the bottom of the file.

- 4.9. Remove the **kind: List** attribute at the bottom of the file, and all other attributes under it.
- ▶ 5. Export and clean the deployment configuration for the quotesapi application and the quotesdb database.
 - 5.1. Export the deployment configuration:

```
[student@workstation ~]$ oc get -o yaml --export dc > /tmp/dc.yaml
```
 - 5.2. Make a copy of the exported file and clean the runtime attributes from it:

```
[student@workstation ~]$ cp /tmp/dc.yaml /tmp/dc-clean.yaml
```
 - 5.3. Clean the runtime attributes from the **/tmp/dc-clean.yaml** file.
A copy of the cleaned file is available at **~/D0288/labs/create-template/dc-clean.yaml**. Compare your cleaned file against this version and make the necessary edits to make them the same.
 - 5.4. Remove the first two lines of the file:

```
apiVersion: v1
items:
```
 - 5.5. Remove all references to **openshift.io/generated-by**, **creationTimestamp**, **generation**, **resourceVersion**, **selfLink**, and **uid** attributes in the file.
 - 5.6. Remove the **namespace** attribute that refers to the **youruser-quotes-dev** project. Do not remove the **namespace: openshift** reference under the **imageChangeParams** attribute lower down in the file.
 - 5.7. Remove the **managedFields** attribute and all its child attributes.
 - 5.8. Remove all references to **image** and **lastTriggeredImage** attributes in the file.
 - 5.9. Remove the **status** attribute and all its child attributes for both deployment configuration resources.
 - 5.10. Remove the **kind: List** attribute at the bottom of the file, and all other attributes under it.
- ▶ 6. Export and clean the service configuration for the quotesapi application and the quotesdb database.
 - 6.1. Export the service configuration:

```
[student@workstation ~]$ oc get -o yaml --export svc > /tmp/svc.yaml
```
 - 6.2. Make a copy of the exported file and clean the runtime attributes from it:

```
[student@workstation ~]$ cp /tmp/svc.yaml /tmp/svc-clean.yaml
```
 - 6.3. Clean the runtime attributes from the **/tmp/svc-clean.yaml** file.

A copy of the cleaned file is available at `~/D0288/labs/create-template/svc-clean.yaml`. Compare your cleaned file against this version and make the necessary edits to make them the same.

- 6.4. Remove the first two lines of the file:

```
apiVersion: v1
items:
```

- 6.5. Remove all references to `openshift.io/generated-by`, `creationTimestamp`, `namespace`, `resourceVersion`, `selfLink`, and `uid` attributes in the file.
- 6.6. Remove the `managedFields` attribute and all its child attributes.
- 6.7. Remove all references to the `clusterIP` attribute under the `spec` attribute in the file.
- 6.8. Remove the `status` attribute and all its child attributes for both service resources.
- 6.9. Remove the `kind: List` attribute at the bottom of the file, and all other attributes under it.

▶ 7. Export and clean the route configuration for the quotesapi application.

- 7.1. Export the route configuration:

```
[student@workstation ~]$ oc get -o yaml --export route > /tmp/route.yaml
```

- 7.2. Make a copy of the exported file and clean the runtime attributes from it:

```
[student@workstation ~]$ cp /tmp/route.yaml /tmp/route-clean.yaml
```

- 7.3. Clean the runtime attributes from the `/tmp/route-clean.yaml` file.
A copy of the cleaned file is available at `~/D0288/labs/create-template/route-clean.yaml`. Compare your cleaned file against this version and make the necessary edits to make them the same.

- 7.4. Remove the first two lines of the file:

```
apiVersion: v1
items:
```

- 7.5. Remove all references to `openshift.io/generated-by`, `creationTimestamp`, `namespace`, `resourceVersion`, `selfLink`, and `uid` attributes in the file.
- 7.6. Remove the `managedFields` attribute and all its child attributes.
- 7.7. Remove the `host` and `subdomain` attributes under the `spec` attribute.
- 7.8. Remove the `status` attribute and all its child attributes.
- 7.9. Remove the `kind: List` attribute at the bottom of the file, and all other attributes under it.

▶ 8. Export and clean the persistent volume configuration (PVC) for the quotesdb database.

- 8.1. Export the persistent volume configuration:

```
[student@workstation ~]$ oc get -o yaml --export pvc > /tmp/pvc.yaml
```

- 8.2. Make a copy of the exported file and clean the runtime attributes from it:

```
[student@workstation ~]$ cp /tmp/pvc.yaml /tmp/pvc-clean.yaml
```

- 8.3. Clean the runtime attributes from the `/tmp/pvc-clean.yaml` file.

A copy of the cleaned file is available at `~/D0288/labs/create-template/pvc-clean.yaml`. Compare your cleaned file against this version and make the necessary edits to make them the same.

- 8.4. Remove the first two lines of the file:

```
apiVersion: v1
items:
```

- 8.5. Remove all attributes under the `metadata.annotations` attribute.
- 8.6. Remove all references to `creationTimestamp`, the `finalizers` attribute and its children, as well as the `namespace`, `resourceVersion`, `selfLink`, and `uid` attributes in the file.
- 8.7. Remove the `managedFields` attribute and all its child attributes.
- 8.8. Remove the `dataSource` attribute under the `spec` attribute.
- 8.9. Remove the `storageClassName`, `volumeMode`, and `volumeName` attributes under the `spec` attribute.
- 8.10. Remove the `status` attribute and all its child attributes.
- 8.11. Remove the `kind: List` attribute at the bottom of the file, and all other attributes under it.

- 9. Copy the individual resource configuration YAML snippets from the cleaned files into the `/home/student/quotes-template-clean.yaml` file under the `objects` attribute in the following order:

```
[student@workstation ~]$ cat /tmp/is-clean.yaml >> ~/quotes-template-clean.yaml
[student@workstation ~]$ cat /tmp/bc-clean.yaml >> ~/quotes-template-clean.yaml
[student@workstation ~]$ cat /tmp/dc-clean.yaml >> ~/quotes-template-clean.yaml
[student@workstation ~]$ cat /tmp/svc-clean.yaml >> ~/quotes-template-clean.yaml
[student@workstation ~]$ cat /tmp/route-clean.yaml >> ~/quotes-template-clean.yaml
[student@workstation ~]$ cat /tmp/pvc-clean.yaml >> ~/quotes-template-clean.yaml
```

A copy of the final cleaned template is available at `~/D0288/solutions/create-template/quotes-template-clean.yaml`. Compare your final cleaned template against this file.

Ignore differences in the `secret` and `uri` attributes. These values will be different for you. You will parameterize these values in the next step.

- 10. Add parameters to make the template reusable.

To keep this guided exercise short, you will add only three parameters: one for the Git URL where the application source code is stored, and two for passwords and secrets.

- 10.1. Inspect the file that contains the parameter definition. Notice that only the **APP_GIT_URL** parameter is required. The **PASSWORD** and **SECRET** parameters have random default values:

```
[student@workstation ~]$ cat ~/DO288/labs/create-template/parameters.yaml
parameters:
- name: APP_GIT_URL
  displayName: Application Source Git URL
  description: The Git URL of the application source code
  required: true
- name: PASSWORD
  displayName: Database Password
  description: Password to access the database
  generate: expression
  from: '[a-zA-Z0-9]{16}'
- name: SECRET
  displayName: Webhook Secret
  description: Secret for webhooks
  generate: expression
  from: '[a-zA-Z0-9]{40}'
```

- 10.2. Make a copy of the final cleaned template YAML file before adding the parameters:

```
[student@workstation ~]$ cp ~/quotes-template-clean.yaml ~/quotes-template.yaml
```

- 10.3. Open the **quotes-template.yaml** file in a text editor. Copy the contents of the **parameters.yaml** file to the end of the **quotes-template.yaml** file.

- 10.4. Change the **secret**, **password**, and **uri** attributes to reference template parameters.

Use the following listing as a guide to make the changes:

```
...output omitted...
kind: BuildConfig
...output omitted...
name: quotesapi
...output omitted...
source:
  contextDir: quotes
  git:
    uri: ${APP_GIT_URL}
  type: Git
  ...output omitted...
triggers:
- github:
    secret: ${SECRET}
  type: GitHub
- generic:
    secret: ${SECRET}
  ...output omitted...
```

```
kind: DeploymentConfig
...output omitted...
name: quotesapi
...output omitted...
- name: DATABASE_PASSWORD
  value: ${PASSWORD}
...output omitted...
kind: DeploymentConfig
...output omitted...
name: quotesdb
...output omitted...
- name: MYSQL_PASSWORD
  value: ${PASSWORD}
...output omitted...
```

- 10.5. To make the MySQL database container persistent, you need to add a **volumeMounts** attribute to the database deployment configuration, and reference the persistent volume claim resource declared in the template.

Add the following lines to the quotesdb deployment configuration as follows:

```
...output omitted...
terminationMessagePath: /dev/termination-log
terminationMessagePolicy: File
volumeMounts:
- mountPath: /var/lib/mysql/data
  name: quotesdb-volume-1
dnsPolicy: ClusterFirst
restartPolicy: Always
...output omitted...
```

- 10.6. The template file is now complete. Save your edits.

To verify the changes you made during this step, review the **quotes-template.yaml** file in the **~/DO288/solutions/create-template** folder. If you are uncertain about your edits, you can copy the solution file and continue to the next step.

► 11. Create a new application from the template.

- 11.1. Create the **youruser-myquotes** project:

```
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-myquotes
Now using project "youruser-myquotes" on server
"https://api.cluster.domain.example.com:6443"
...output omitted...
```

► 12. Create a new instance of the quotes application from the template.

- 12.1. Use the **oc new-app** command to create a new instance of the quotes application from the template:

```
[student@workstation ~]$ oc new-app --file=quotes-template.yaml \
> -p APP_GIT_URL=https://github.com/${RHT_OCP4_GITHUB_USER}/DO288-apps \
> -p PASSWORD=mypass
```

```
--> Deploying template "youruser-myquotes/quotes" to project youruser-myquotes

Quotes Application
-----
The Quotes application provides an HTTP API that returns a random, funny quote.

* With parameters:
* Application Source Git URL=https://github.com/youruser/D0288-apps
* Database Password=mypass
* Webhook Secret=D6xlih2F3KIyYwsIXs3nLysGHJbi6JLhtaul6I10 # generated

--> Creating resources ...
imagestream.image.openshift.io "quotesapi" created
buildconfig.build.openshift.io "quotesapi" created
deploymentconfig.apps.openshift.io "quotesapi" created
deploymentconfig.apps.openshift.io "quotesdb" created
service "quotesapi" created
service "quotesdb" created
route.route.openshift.io "quotesapi" created
persistentvolumeclaim "quotesdb-claim" created
--> Success
...output omitted...
```



Note

If you see an error at this step, ensure that there are no extra line breaks, and that your template YAML file is properly indented. OpenShift displays the line number in the template where processing failed. Use this information to identify and fix the issues.

- 12.2. Wait for the build to complete and for the quotesdb and quotesapi applications to have one pod each ready and running.

```
[student@workstation ~]$ oc get pods
NAME          READY   STATUS    RESTARTS   AGE
...output omitted...
quotesapi-1-tphzw   1/1     Running   0          3m12s
quotesdb-1-rk8mw   1/1     Running   0          4m28s
```

- 13. Populate the database and test the application.

- 13.1. Inspect the script that populates the database.

The **populate-db.sh** script in the **~/D0288/labs/create-template** directory uses a port-forwarding tunnel to connect to the database pod and a local MySQL client to run an SQL script:

```
[student@workstation ~]$ cat ~/D0288/labs/create-template/populate-db.sh
...output omitted...
echo 'Creating tunnel...'
pod=$(oc get pod -l deploymentconfig=quotesdb -o name)
oc port-forward $(basename ${pod}) 30306:3306 &
tunnel=$!
sleep 3
```

```
echo 'Initializing the database...'
mysql -h127.0.0.1 -P30306 -uquoteapp -pmypass quotesdb \
< ~/DO288/labs/create-template/quote.sql
sleep 3
echo 'Terminating tunnel...'
kill ${tunnel}
```

13.2. Run the **populate-db.sh** script:

```
[student@workstation ~]$ ~/DO288/labs/create-template/populate-db.sh
Creating tunnel...
Forwarding from 127.0.0.1:30306 -> 3306
Forwarding from [::1]:30306 -> 3306
Initializing the database...
Handling connection for 30306
Terminating tunnel...
```

13.3. Get the route host name for the application:

```
[student@workstation ~]$ oc get route/quotesapi -o jsonpath='{.spec.host}{"\n"}'
quotesapi-youruser-quotes-dev.apps.cluster.domain.example.com
```

13.4. Use the **curl** command and the host name from the previous step to access the application. Your output might be different from this example:

```
[student@workstation ~]$ curl \
> quotesapi-${RHT_OCP4_DEV_USER}-myquotes.${RHT_OCP4_WILDCARD_DOMAIN}/get.php
Veni, vidi, vici...
```

- 14. Clean up. Delete the projects from OpenShift and release the persistent storage allocated for this exercise.

Open a terminal window on the **workstation** VM and run the following command to perform the cleanup tasks:

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

This concludes the guided exercise.

► Lab

Creating Applications from OpenShift Templates

Performance Checklist

In this lab, you will deploy the To Do List application from a template that you will complete for reuse.

Outcomes

You should be able to complete a template that deploys a database pod and a web application pod, use the template to deploy the application, and verify that the application works.

Before You Begin

To perform this exercise, ensure you have access to:

- A running OpenShift cluster.
- The sample application (`todo-single`) in the Git repository.
- The npm dependencies required by the application (`restify`, `sequelize`, and `mysql`).
- The Node.js 10 S2I builder image, and the MySQL 5.7 database image required by the template.

Run the following command on the **workstation** VM to validate the prerequisites and to download the starter project files and solution files:

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

Requirements

The application is a To Do List application written in JavaScript. It consists of a web front end, based on the AngularJS web framework, and an HTTP API back end, based on Node.js. The back end uses the Restify and Sequelize frameworks.

Both the application front end and back end run on the same container. The application is deployed from source code stored in a Git repository. A MySQL database is used as the data store. The application initializes the database on startup.

The template takes the following parameters:

- **APP_GIT_URL**: The Git URL where the source code for the To Do List application is stored.
- **HOSTNAME**: The host name used to access the To Do List application.
- **NPM_PROXY**: The URL of the NPM repository server.
- **SECRET**: The secret for OpenShift webhooks.

- **PASSWORD:** The database connection password. The user name is fixed in the template.
- **CLEAN_DATABASE:** A flag that indicates whether the application initializes the database on startup.

The template parameters have the following restrictions:

- The **APP_GIT_URL**, **HOSTNAME**, **NPM_PROXY**, and **CLEAN_DATABASE** parameters are required.
- The **SECRET** and **PASSWORD** parameters have random default values.
- The **CLEAN_DATABASE** parameter has the default value of **true**.

A starter template file called **todo-template.yaml** is provided in the **~/D0288/labs/review-template** folder. It contains the resources required to deploy the application. The resources are in the correct order, and already cleaned up. You are required to add the missing parameters and add references to the parameters in the resource list.

Deploy the application according to the following requirements:

- The application project is called **youruser-review-template**.
- Use the **~/D0288/labs/review-template/oc-new-app.sh** script to deploy the application. Edit the script as needed to pass any required parameters.
- The application initializes the database on startup.
- The password to access the database is **mypass**.
- Npm modules required to build the application are available from:
http://nexus-common.apps.cluster.domain.example.com/repository/nodejs
- If you prefer to test the application with a web browser, use the web front end to add some entries to the To Do List application interactively. The application user interface is accessible from the following URL:
http://youruser-todo.apps.cluster.domain.example.com/todo/index.html
- If you prefer to test the application using the command line, use the following URL to submit an HTTP GET request to the application back end:
http://youruser-todo.apps.cluster.domain.example.com/todo/api/items-count

The reply includes a **count** attribute, even if there are no entries in the application. An error reply means the database was not initialized correctly.

Steps

1. Review the **todo-template.yaml** starter template file in the **~/D0288/labs/review-template** folder. Make a copy of this file in the **/home/student/** directory, and use this copy to make your changes throughout the lab. Identify and add any missing parameters at the end of the file. Identify any parameter references missing from the template resource list and add them as needed. Use the existing parameter definitions and parameter references as a guide to add the missing ones.

You are not required to, but you can inspect the **todo-template-clean.yaml** file in the same folder. This file contains the application resources exported by the **oc get** command, with the runtime attributes already removed. Do not make any changes to this file.

You are not required to, but you can also inspect the **new-app-db.sh**, **new-app-node.sh**, and **add-route.sh** files in the same folder that contain the commands used to generate the resources that were exported to the **todo-template-clean.yaml** file. Do not make any changes to these files and do not run any of them.

2. Create a new project and deploy the To Do List application using the template definition file you completed during the previous step.

Review the **oc-new-app.sh** starter script file in the **~/DO288/labs/review-template** folder. Make a copy of this file in the **/home/student/** directory and use this copy to make your changes throughout the lab. Identify and add any missing parameters to the **oc new-app** command line. Run the finalized **oc-new-app.sh** script to deploy the application.

Do not perform any manual steps to initialize the database. The To Do List application creates the required database tables if you pass the correct set of parameters to the template. The database tables do not need an initial data set.

3. Test the To Do List application using either a web browser or the command line. Do not forget to source the variables from the **/usr/local/etc/ocp4.config** before logging in to OpenShift.
4. Grade your work.

Run the following command on the **workstation** VM to verify that all tasks were accomplished:

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

5. Clean up. Delete the project from OpenShift.

Open a terminal window on the **workstation** VM and run the following command to perform the cleanup tasks:

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

This concludes the lab.

► Solution

Creating Applications from OpenShift Templates

Performance Checklist

In this lab, you will deploy the To Do List application from a template that you will complete for reuse.

Outcomes

You should be able to complete a template that deploys a database pod and a web application pod, use the template to deploy the application, and verify that the application works.

Before You Begin

To perform this exercise, ensure you have access to:

- A running OpenShift cluster.
- The sample application (`todo-single`) in the Git repository.
- The npm dependencies required by the application (`restify`, `sequelize`, and `mysql`).
- The Node.js 10 S2I builder image, and the MySQL 5.7 database image required by the template.

Run the following command on the **workstation** VM to validate the prerequisites and to download the starter project files and solution files:

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

Requirements

The application is a To Do List application written in JavaScript. It consists of a web front end, based on the AngularJS web framework, and an HTTP API back end, based on Node.js. The back end uses the Restify and Sequelize frameworks.

Both the application front end and back end run on the same container. The application is deployed from source code stored in a Git repository. A MySQL database is used as the data store. The application initializes the database on startup.

The template takes the following parameters:

- **APP_GIT_URL**: The Git URL where the source code for the To Do List application is stored.
- **HOSTNAME**: The host name used to access the To Do List application.
- **NPM_PROXY**: The URL of the NPM repository server.
- **SECRET**: The secret for OpenShift webhooks.

- **PASSWORD:** The database connection password. The user name is fixed in the template.
- **CLEAN_DATABASE:** A flag that indicates whether the application initializes the database on startup.

The template parameters have the following restrictions:

- The **APP_GIT_URL**, **HOSTNAME**, **NPM_PROXY**, and **CLEAN_DATABASE** parameters are required.
- The **SECRET** and **PASSWORD** parameters have random default values.
- The **CLEAN_DATABASE** parameter has the default value of **true**.

A starter template file called **todo-template.yaml** is provided in the **~/D0288/labs/review-template** folder. It contains the resources required to deploy the application. The resources are in the correct order, and already cleaned up. You are required to add the missing parameters and add references to the parameters in the resource list.

Deploy the application according to the following requirements:

- The application project is called **youruser-review-template**.
- Use the **~/D0288/labs/review-template/oc-new-app.sh** script to deploy the application. Edit the script as needed to pass any required parameters.
- The application initializes the database on startup.
- The password to access the database is **mypass**.
- Npm modules required to build the application are available from:
<http://nexus-common.apps.cluster.domain.example.com/repository/nodejs>
- If you prefer to test the application with a web browser, use the web front end to add some entries to the To Do List application interactively. The application user interface is accessible from the following URL:
<http://youruser-todo.apps.cluster.domain.example.com/todo/index.html>
- If you prefer to test the application using the command line, use the following URL to submit an HTTP GET request to the application back end:
<http://youruser-todo.apps.cluster.domain.example.com/todo/api/items-count>

The reply includes a **count** attribute, even if there are no entries in the application. An error reply means the database was not initialized correctly.

Steps

1. Review the **todo-template.yaml** starter template file in the **~/D0288/labs/review-template** folder. Make a copy of this file in the **/home/student/** directory, and use this copy to make your changes throughout the lab. Identify and add any missing parameters at the end of the file. Identify any parameter references missing from the template resource list and add them as needed. Use the existing parameter definitions and parameter references as a guide to add the missing ones.

You are not required to, but you can inspect the **todo-template-clean.yaml** file in the same folder. This file contains the application resources exported by the **oc get** command, with the runtime attributes already removed. Do not make any changes to this file.

You are not required to, but you can also inspect the **new-app-db.sh**, **new-app-node.sh**, and **add-route.sh** files in the same folder that contain the commands used to generate the resources that were exported to the **todo-template-clean.yaml** file. Do not make any changes to these files and do not run any of them.

- 1.1. Create a copy of the starter template to make edits:

```
[student@workstation ~]$ cp ~/D0288/labs/review-template/todo-template.yaml \
> ~/todo-template.yaml
```

- 1.2. Identify any missing parameters in the template file.

Open the **~/todo-template.yaml** file with a text editor.

Scroll down to the end of the file. The file contains four parameters called **APP_GIT_URL**, **HOSTNAME**, **NPM_PROXY**, and **SECRET**. These parameter definitions are complete and do not need any changes.

The **PASSWORD** and **CLEAN_DATABASE** parameters are missing and need to be added.

- 1.3. Add the missing parameters to the **~/todo-template.yaml** template file.

Append the following lines to the file. You can copy these lines from the **add-parameters.yaml** file in the **~/D0288/solutions/review-template** folder:

```
- name: PASSWORD
  displayName: Database Password
  description: Password to access the database
  generate: expression
  from: '[a-zA-Z0-9]{16}'
- name: CLEAN_DATABASE
  displayName: Initialize the database
  description: If 'true', the database is cleaned when the application starts.
  required: true
  value: "true"
```

- 1.4. Identify the missing references to parameter values in the template resource list.

Some of the resources already reference the parameters correctly. The following listing highlights the parameter references that are already in the template file. You do not need to change any of them:

```
...output omitted...
kind: BuildConfig
...output omitted...
  name: todoapp
  ...output omitted...
  triggers:
  - github:
      secret: ${SECRET}
    type: GitHub
  - generic:
      secret: ${SECRET}
  ...output omitted...
kind: DeploymentConfig
  ...output omitted...
  name: todoapp
```

```
...output omitted...
- env:
  - name: DATABASE_NAME
    value: tododb
  - name: DATABASE_PASSWORD
    value: ${PASSWORD}
  - name: DATABASE_SVC
    value: tododb
  - name: DATABASE_USER
    value: todoapp
  - name: DATABASE_INIT
    value: ${CLEAN_DATABASE}
...output omitted...
```

A few resources are missing references to parameters. The following listing highlights the attributes that are set to hard-coded values and need to be replaced by parameter references:

```
...output omitted...
kind: BuildConfig
...output omitted...
  name: todoapp
...output omitted...
  strategy:
    sourceStrategy:
      env:
        - name: npm_config_registry
          value: http://INVALIDHOST.NODOMAIN.NUL
...output omitted...
kind: DeploymentConfig
...output omitted...
  name: tododb
...output omitted...
  - env:
    - name: MYSQL_DATABASE
      value: tododb
    - name: MYSQL_PASSWORD
      value: FIXEDPASSWD
...output omitted...
kind: Route
...output omitted...
  name: todoapp
...output omitted...
spec:
  host: http://INVALIDHOST.NODOMAIN.NUL
...output omitted...
```

15. Add the missing references to parameters to the `~/todo-template.yaml` file. The following listing shows the changes you need to make:

```
...output omitted...
kind: BuildConfig
...output omitted...
  name: todoapp
```

```

...output omitted...
strategy:
sourceStrategy:
env:
- name: npm_config_registry
  value: ${NPM_PROXY}
...output omitted...
kind: DeploymentConfig
...output omitted...
name: tododb
...output omitted...
- env:
  - name: MYSQL_DATABASE
    value: tododb
  - name: MYSQL_PASSWORD
    value: ${PASSWORD}
...output omitted...
kind: Route
...output omitted...
name: todoapp
...output omitted...
spec:
host: ${HOSTNAME}
...output omitted...

```

- 1.6. Review your edits and save the changes to the `~/todo-template.yaml` template file. You can compare your edits with the solution file in the `~/D0288/solutions/review-template` folder.
2. Create a new project and deploy the To Do List application using the template definition file you completed during the previous step.

Review the `oc-new-app.sh` starter script file in the `~/D0288/labs/review-template` folder. Make a copy of this file in the `/home/student/` directory and use this copy to make your changes throughout the lab. Identify and add any missing parameters to the `oc new-app` command line. Run the finalized `oc-new-app.sh` script to deploy the application.

Do not perform any manual steps to initialize the database. The To Do List application creates the required database tables if you pass the correct set of parameters to the template. The database tables do not need an initial data set.

- 2.1. Create a copy of the starter script to make edits:

```
[student@workstation ~]$ cp ~/D0288/labs/review-template/oc-new-app.sh \
> ~/oc-new-app.sh
```

- 2.2. Identify any missing parameters in the `~/oc-new-app.sh` script file.

Open the `~/oc-new-app.sh` file with a text editor.

The script passes values for the `APP_GIT_URL`, `NPM_PROXY`, `PASSWORD`, and `CLEAN_DATABASE` parameters.

The values passed for `APP_GIT_URL`, `NPM_PROXY`, and `PASSWORD` are correct; you do not need to change them. You need to change the value for the `CLEAN_DATABASE` parameter to `true`.

You also need to add a command-line option to pass a value for the `HOSTNAME` parameter.

2.3. Make changes to the `~/oc-new-app.sh` script.

Use the following listing as a guide to make the changes:

```
oc new-app --as-deployment-config --name todo --file ~/todo-template.yaml \
> -p APP_GIT_URL=https://github.com/${RHT_OCP4_GITHUB_USER}/D0288-apps \
> -p NPM_PROXY=http://${RHT_OCP4_NEXUS_SERVER}/repository/nodejs \
> -p PASSWORD=mypass \
> -p CLEAN_DATABASE=true \
> -p HOSTNAME=${RHT_OCP4_DEV_USER}-todo.${RHT_OCP4_WILDCARD_DOMAIN}
```

Do not add line breaks to any parameter value. The value for **NPM_PROXY** needs to be on a single line. Recall that you do not need to change it from the starter file.

- 2.4. Review your edits and save the changes to the `~/oc-new-app.sh` file. You can compare your edits with the solution file in the `~/D0288/solutions/review-template` folder.
3. Test the To Do List application using either a web browser or the command line. Do not forget to source the variables from the `/usr/local/etc/ocp4.config` before logging in to OpenShift.

3.1. Load your classroom environment configuration.

Run the following command to load the environment variables created in the first guided exercise:

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

3.2. Log in to OpenShift using your developer user account:

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful.
...output omitted...
```

3.3. Create the `youruser-review-template` project:

```
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-review-template
Now using project "youruser-review-template" on server
"https://api.cluster.domain.example.com:6443"
...output omitted...
```

3.4. Run the `oc-new-app.sh` script to create all of the resources from the `todo-template.yaml` template file:

```
[student@workstation ~]$ ~/oc-new-app.sh
...output omitted...
--> Creating resources...
imagestream "todoapp" created
buildconfig "todoapp" created
deploymentconfig "todoapp" created
deploymentconfig "tododb" created
service "todoapp" created
service "tododb" created
```

```
route "todoapp" created
--> Success
...output omitted...
```

- 3.5. Wait for the To Do List application build to finish:

```
[student@workstation ~]$ oc logs bc/todoapp -f
...output omitted...
Push successful
```

- 3.6. Wait for the application and the database pod to be ready and running:

```
[student@workstation ~]$ oc get pods
NAME        READY   STATUS    RESTARTS   AGE
...output omitted...
todoapp-1-nch6c  1/1     Running   0          1m
tododb-1-lpk63  1/1     Running   0          2m
```

- 3.7. Get the route host name for the application:

```
[student@workstation ~]$ oc get route/todoapp -o jsonpath='{.spec.host}{"\n"}'
youruser-todo.apps.cluster.domain.example.com
```

- 3.8. Test the application using the route host name obtained from the previous step.

If you prefer to test the To Do List application using a web browser, access the following URL and add some entries to the list:

<http://youruser-todo.apps.cluster.domain.example.com/todo/index.html>

- 3.9. If you prefer to test the To Do List application using the command line, open a terminal window and run the following command:

```
[student@workstation ~]$ curl -siw "\n" \
> http://${RHT_OCP4_DEV_USER}-todo.${RHT_OCP4_WILDCARD_DOMAIN}\ \
> /todo/api/items-count
HTTP/1.1 200 OK
...output omitted...
{"count":0}
```

4. Grade your work.

Run the following command on the **workstation** VM to verify that all tasks were accomplished:

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

5. Clean up. Delete the project from OpenShift.

Open a terminal window on the **workstation** VM and run the following command to perform the cleanup tasks:

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

This concludes the lab.

Summary

In this chapter, you learned:

- A template is a parameterized list of OpenShift resources. A typical use case for templates is deploying applications composed of multiple pods.
- Template parameters can be required or optional, can define default values, and can define randomly generated default values that are useful for secrets and passwords.
- Typical methods of creating a template are concatenating the output of multiple **oc new-app -o** commands and exporting existing resources using the **oc get -o yaml --export** command.
- The **oc describe** and **oc explain** commands help developers to find information about all valid attributes for a given kind of resource.

Chapter 7

Managing Application Deployments

Goal

Monitor application health and implement various deployment methods for cloud-native applications.

Objectives

- Implement liveness and readiness probes.
- Select the appropriate deployment strategy for a cloud-native application.
- Manage the deployment of an application with CLI commands.

Sections

- Monitoring Application Health (and Guided Exercise)
- Selecting an Appropriate Deployment Strategy (and Guided Exercise)
- Managing Application Deployments with CLI Commands (and Guided Exercise)

Lab

Managing Application Deployments

Monitoring Application Health

Objectives

After completing this section, you should be able to implement readiness and liveness probes to monitor application readiness and health.

OpenShift Readiness and Liveness Probes

Applications can become unreliable for a variety of reasons, including temporary connectivity loss, configuration errors, or application errors. Developers can use *probes* to monitor their applications and be made aware of events that can vary from status to resource usage and errors. This monitoring is useful for fixing problems, but can also help with resource planning and managing. A probe is an OpenShift action that periodically performs diagnostics on a running container. Probes can be configured using either the `oc` command-line client, defined as part of an OpenShift template, or by using the OpenShift web console. There are currently two types of probes in OpenShift:

Readiness Probe

Readiness probes determine whether or not a container is ready to serve requests. If the readiness probe returns a failed state, OpenShift removes the IP address for the container from the endpoints of all services. Developers can use readiness probes to signal to OpenShift that even though a container is running, it should not receive any traffic from a proxy. The readiness probe is configured in the `spec.containers.readinessprobe` attribute of the pod configuration.

Liveness Probe

Liveness probes determine whether or not an application running in a container is in a **healthy** state. If the liveness probe detects an unhealthy state, OpenShift kills the container and tries to redeploy it. The liveness probe is configured in the `spec.containers.livenessprobe` attribute of the pod configuration.

OpenShift provides five options that control these two probes:

Name	Mandatory	Description	Default Value
<code>initialDelaySeconds</code>	Yes	Determines how long to wait after the container starts before beginning the probe.	0
<code>timeoutSeconds</code>	Yes	Determines how long to wait for the probe to finish. If this time is exceeded, OpenShift assumes that the probe failed.	1
<code>periodSeconds</code>	No	Specifies the frequency of the checks.	1

Name	Mandatory	Description	Default Value
successThreshold	No	Specifies the minimum consecutive successes for the probe to be considered successful after it has failed.	1
failureThreshold	No	Specifies the minimum consecutive failures for the probe to be considered failed after it has succeeded.	3

Methods of Checking Application Health

Readiness and liveness probes can check the health of applications in three ways:

HTTP Checks

An HTTP check is ideal for applications that return HTTP status codes, such as REST APIs.

OpenShift uses HTTP GET requests to check the status code of responses to determine the health of a container. The check is deemed successful if the HTTP response code is in the range 200-399. The following example demonstrates how to implement a readiness probe with the HTTP check method:

```
...
readinessProbe:
  httpGet:
    path: /health①
    port: 8080
  initialDelaySeconds: 15②
  timeoutSeconds: 1③
...

```

- ① The URL to query.
- ② How long to wait after the container starts before checking its health.
- ③ How long to wait for the probe to finish.

Container Execution Checks

Container execution checks are ideal in scenarios where you must determine the readiness and liveness status of the container based on the exit code of a process or shell script running in the container.

When using container execution checks, OpenShift executes a command inside the container. Exiting the check with a status of **0** is considered a success. All other status codes are considered a failure. The following example demonstrates how to implement a container execution check:

```
...
livenessProbe:
  exec:
    command: ①
    - cat

```

```

    - /tmp/health
initialDelaySeconds: 15
timeoutSeconds: 1
...

```

- ➊ The command to run and its arguments, as an YAML array.

TCP Socket Checks

A TCP socket check is ideal for applications that run as daemons, and open TCP ports, such as database servers, file servers, web servers, and application servers.

When using TCP socket checks, OpenShift attempts to open a socket to the container. The container is considered healthy if the check can establish a successful connection. The following example demonstrates how to implement a liveness probe using the TCP socket check method:

```

...
livenessProbe:
  tcpSocket:
    port: 8080❶
  initialDelaySeconds: 15
  timeoutSeconds: 1
...

```

- ❶ The TCP port to check.

Manage Probes Using the Web Console

Developers can create both readiness and liveness probes by editing the Deployment Configuration YAML file using either `oc edit` or the OpenShift Web Console. You can directly edit the Deployment Configuration YAML file from the **Workloads** → **Deployment Configs** → **<deployment name>** page in the web console. Click the **Actions** drop down, and select **Edit Deployment Config**.

The screenshot shows the 'Deployment Configs' section of the OpenShift Web Console. A deployment config named 'hello' is selected. The 'Actions' dropdown menu is open, showing options like 'Start Rollout', 'Edit Pod Count', and 'Edit Deployment Config'. The 'Edit Deployment Config' option is highlighted with a red box.

Deployment Config Details	
Name	hello
Namespace	NS your-project
Latest Version	1
Message	config change

Figure 7.1: Adding probes using the web console

The following example shows the YAML editor in the web console for a deployment configuration.

```

191      terminationMessagePath: /dev/termination-log
192      name: httpd-example
193      livenessProbe:
194        httpGet:
195          path: /healthz
196          port: 8080
197          scheme: HTTP
198        initialDelaySeconds: 30
199        timeoutSeconds: 3
200        periodSeconds: 10
201        successThreshold: 1
202        failureThreshold: 3
203      ports:
204        - containerPort: 8080
205          protocol: TCP
imagePullPolicy: IfNotPresent
  
```

Figure 7.2: Adding probes using the web console YAML editor

Creating Probes Using CLI

The **oc set probe** command provides an alternative approach to editing the deployment configuration YAML definition directly. When you are creating probes for running applications, using the **oc set probe** is the recommended approach as it limits your ability to make mistakes. This command provides a number of options that allow you to specify the type of probe, either readiness or liveness, as well as other necessary attributes such as the port, URL, timeout, period, and more.

The following examples demonstrate using the **oc set probe** command with a variety of options:

```
[user@host ~]$ oc set probe dc/myapp --readiness \
> --get-url=http://:8080/healthz --period=20
```

```
[user@host ~]$ oc set probe dc/myapp --liveness \
> --open-tcp=3306 --period=20 \
> --timeout-seconds=1
```

```
[user@host ~]$ oc set probe dc/myapp --liveness \
> --get-url=http://:8080/healthz --initial-delay-seconds=30 \
> --success-threshold=1 --failure-threshold=3
```

Use the **oc set probe --help** command to view all of the available options for this command.



References

Further information is available in the *Monitoring application health* section of the *Applications* chapter of the *Official Documentation* for OpenShift Container Platform 4.5; at

https://access.redhat.com/documentation/en-us/openshift_container_platform/4.5/html/applications/application-health

Further information is available in the *Configure Liveness and Readiness Probes* page of the *Kubernetes* website at

<https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-probes/>

► Guided Exercise

Activating Probes

In this exercise, you will activate liveness and readiness probes to monitor the health of an application deployed to an OpenShift cluster.

Outcomes

You should be able to:

- Activate readiness and liveness probes for an application from the command line.
- Locate probe failure messages in the event log.

Before You Begin

To perform this exercise, ensure you have access to:

- A running OpenShift cluster.
- The Node.js S2I builder image.
- The sample application in the Git repository (probes).

Run the following command on **workstation** to validate the exercise prerequisites, and to download the lab files:

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

The application you deploy in this exercise exposes two HTTP **GET** endpoints at `/healthz` and `/ready`. The `/healthz` endpoint will be used by the liveness probe, and the `/ready` endpoint will be used by the readiness probe. The endpoints respond with an HTTP status code of 200 if the pod is ready and in a healthy state, or a 503 status code otherwise.

- 1. Create a new project, and then deploy the sample application in the **probes** subdirectory of the Git repository to an OpenShift cluster.

- 1.1. Load your classroom environment configuration.

Run the following command to load the environment variables created in the first guided exercise:

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

- 1.2. Log in to OpenShift using your developer user account:

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
```

- 1.3. Create a new project:

```
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-probes
```

- 1.4. Create a new application called probes from sources in the **probes** subdirectory of the Git repository.

You can copy or execute the command from the **oc-new-app.sh** script in the **/home/student/D0288/labs/probes** folder:

```
[student@workstation ~]$ oc new-app --as-deployment-config \
> --name probes --build-env \
> npm_config_registry=http://${RHT_OCP4_NEXUS_SERVER}/repository/nodejs \
> nodejs:12~https://github.com/${RHT_OCP4_GITHUB_USER}/D0288-apps \
> --context-dir probes
...output omitted...
--> Creating resources ...
...output omitted...
--> Success
...output omitted...
```

Note that there is no space before or after the equals sign (=) that follows **npm_config_registry**.

- 1.5. View the build logs. Wait until the build finishes and the application container image is pushed to the OpenShift registry:

```
[student@workstation ~]$ oc logs -f bc/probes
...output omitted...
STEP 8: RUN /usr/libexec/s2i/assemble
--> Installing application source ...
--> Building your Node application from source
...output omitted...
Push successful
```

- 1.6. Wait until the application is deployed. View the status of the application pod. The application pod should be in a *Running* state:

[student@workstation ~]\$ oc get pods				
NAME	READY	STATUS	RESTARTS	AGE
probes-1-build	0/1	Completed	0	3m48s
probes-1-deploy	0/1	Completed	0	2m21s
probes-1-vlzh2	1/1	Running	0	2m13s

► 2. Manually test the application's **/ready** and **/healthz** endpoints.

- 2.1. Use a route to expose the application to external access:

```
[student@workstation ~]$ oc expose svc probes
route.route.openshift.io/probes exposed
```

- 2.2. Test the **/ready** endpoint of the application:

```
[student@workstation ~]$ curl \
> -i probes-${RHT_OCP4_DEV_USER}-probes.${RHT_OCP4_WILDCARD_DOMAIN}/ready
```

The /ready endpoint simulates a slow startup of the application, and so for the first 30 seconds after the application starts, it returns an HTTP status code of 503, and the following response:

```
HTTP/1.1 503 Service Unavailable
...output omitted...
Error! Service not ready for requests...
```

After the application has been running for 30 seconds, it returns:

```
HTTP/1.1 200 OK
...output omitted...
Ready for service requests...
```

2.3. Test healthz endpoint of the application:

```
[student@workstation ~]$ curl \
> -i probes-${RHT_OCP4_DEV_USER}-probes.${RHT_OCP4_WILDCARD_DOMAIN}/healthz
HTTP/1.1 200 OK
...output omitted...
OK
```

2.4. Test the application response:

```
[student@workstation ~]$ curl \
> probes-${RHT_OCP4_DEV_USER}-probes.${RHT_OCP4_WILDCARD_DOMAIN}
Hello! This is the index page for the app.
```

▶ 3. Activate readiness and liveness probes for the application.

3.1. Use the **oc set** command to add liveness and readiness probes to the Deployment Config.

```
[student@workstation ~]$ oc set probe dc/probes --liveness \
> --get-url=http://:8080/healthz \
> --initial-delay-seconds=2 --timeout-seconds=2
deploymentconfig.apps.openshift.io/probes probes updated
[student@workstation ~]$ oc set probe dc/probes --readiness \
> --get-url=http://:8080/ready \
> --initial-delay-seconds=2 --timeout-seconds=2
deploymentconfig.apps.openshift.io/probes probes updated
```

3.2. Verify the value in the **livenessProbe** and **readinessProbe** entries:

```
[student@workstation D0288-apps]$ oc describe dc/probes
Name: probes
...output omitted...
Liveness: http-get http://:8080/healthz delay=2s timeout=2s...
Readiness: http-get http://:8080/ready delay=2s timeout=2s...
...output omitted...
```

- 3.3. Wait for the application pod to redeploy and appear in a **Running** state:

```
[student@workstation D0288-apps]$ oc get pods
NAME READY STATUS RESTARTS AGE
...output omitted...
probes-3-hppqxi 0/1 Running 0 6s
```

The **READY** status will show **0/1** if the **AGE** value is less than approximately 30 seconds. After that, the **READY** status is **1/1**:

```
[student@workstation D0288-apps]$ oc get pods
NAME READY STATUS RESTARTS AGE
...output omitted...
probes-3-hppqxi 1/1 Running 0 62s
```

- 3.4. Use the **oc logs** command to see the results of the liveness and readiness probes:

```
[student@workstation ~]$ oc logs -f dc/probes
...output omitted...
nodejs server running on http://0.0.0.0:8080
ping /healthz => pong [healthy]
ping /ready => pong [notready]
ping /healthz => pong [healthy]
ping /ready => pong [notready]
ping /healthz => pong [healthy]
ping /ready => pong [ready]
...output omitted...
```

Observe that the readiness probe fails for about 30 seconds after redeployment, and then succeeds. Recall that the application simulates a slow initialization of the application by forcibly setting a 30-second delay before it responds with a status of ready.

Do not terminate this command. You will continue to monitor the output of this command in the next step.

► 4. Simulate a failure of the liveness probe.

- 4.1. In a different terminal window or tab, execute the **~/D0288/labs/probes/kill.sh** script to simulate a liveness probe failure:

```
[student@workstation ~]$ ~/D0288/labs/probes/kill.sh
Switched app state to unhealthy...
```

- 4.2. Return to the terminal where you are monitoring the application deployment:

```
[student@workstation ~]$ oc logs -f dc/probes
...output omitted...
Received kill request. Changing app state to unhealthy...
ping /ready => pong [ready]
ping /healthz => pong [unhealthy]
ping /ready => pong [ready]
ping /healthz => pong [unhealthy]
ping /ready => pong [ready]
npm info lifecycle probes@1.0.0~poststart: probes@1.0.0
npm timing npm Completed in 150591ms
npm info ok
```

Notice that OpenShift restarts the pod when the liveness probe fails a few times (the default is three times). You will see this log output only when you immediately check the application logs after you issue the kill request. If you check the logs after OpenShift restarts the pod, the logs are cleared and you only see the output shown in the next step.

- Verify that OpenShift restarts the unhealthy pod. Keep checking the output of the **oc get pods** command. Observe the **RESTARTS** column and verify that the count is greater than zero:

```
[student@workstation ~]$ oc get pods
NAME      READY   STATUS    RESTARTS   AGE
...output omitted...
probes-3-ltkkp   1/1     Running   1          21m
```

- Check the application logs again and note that the liveness probe succeeds and the application reports a healthy state:

```
[student@workstation ~]$ oc logs -f dc/probes
...output omitted...
ping /ready => pong [ready]
ping /healthz => pong [healthy]
...output omitted...
```

- ▶ 5. Verify that the failure of the liveness probe is seen in the event log using the **oc describe** command on the pod from the previous step.

```
[student@workstation ~]$ oc describe pod/probes-3-ltkkp
Events:
  Type    Reason     Age   From     Message
  ----  -----  ----  ----
  ...output omitted...
  Warning  Unhealthy  ...   ...   Liveness probe failed: ... statuscode: 503
  Normal   Killing    ...   ...   Container ... will be restarted
  Normal   Created    ...   ...   Created container probes
  Warning  Unhealthy  ...   ...   Readiness probe failed: ... statuscode: 503
```

Notice that OpenShift records the liveness probe failure and restarts the pod after the probe fails three times. Because the application has a 30 second delay before the readiness

probe succeeds, you will also see a readiness probe failure reported in the event log after OpenShift restarts the pod.

- 6. Clean up. Delete the **probes** project in OpenShift:

```
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-probes
```

Finish

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

```
[student@workstation ~]$ lab probes finish
```

This concludes the guided exercise.

Selecting the Appropriate Deployment Strategy

Objectives

After completing this section, you should be able to select the appropriate deployment strategy for a cloud-native application.

Deployment Strategies in OpenShift

A *deployment strategy* is a method of changing or upgrading an application. The objective is to make changes or upgrades with minimal downtime, and with reduced impact on end users.

OpenShift provides several deployment strategies. These strategies can be organized into two primary categories. One approach uses the deployment configuration for the application to define the deployment strategy, and the other approach uses certain features from the OpenShift router to route traffic to the application pods.

Strategies that change the deployment configuration impact all routes that use the application. Strategies that use router features affect individual routes.

Strategies that involve changing the deployment configuration are listed below:

Rolling

The rolling strategy is the default strategy used if you do not specify a strategy in the deployment configuration for an application.

This strategy progressively replaces instances of the previous version of an application with instances of the new version of the application. This strategy runs readiness probes to determine when new pods are ready before scaling down older pods. If a significant issue occurs, the rolling deployment is aborted. Deployment can also be manually aborted by using the `oc rollout cancel` command.

Rolling deployments in OpenShift are *canary deployments*; OpenShift tests a new version (the canary) before replacing all of the old instances. If the readiness probe never succeeds, OpenShift removes the canary instance and automatically rolls back the deployment configuration.

Use a rolling deployment strategy when:

- You want no downtime during an application update.
- Your application supports running an older version and a newer version at the same time.

Recreate

In this strategy, OpenShift first stops all the pods that are currently running and only then starts up pods with the new version of the application. This strategy incurs downtime because, for a brief period, no instances of your application are running.

Use a recreate deployment strategy when:

- Your application does not support running an older version and a newer version at the same time.

- Your application uses a persistent volume with RWO (ReadWriteOnce) access mode, which does not allow writes from multiple pods.

Custom

If neither the rolling nor the recreate deployment strategies suit your needs, you can use the custom deployment strategy to deploy your applications. There are times when the command to be executed needs more fine tuning for the system (e.g., memory for the Java Virtual Machine), or you need to use a custom image with in-house developed libraries that are not available to the general public. For these types of use cases, use the Custom strategy. You can provide your own custom container image in which you define the deployment behavior. This custom image is defined in the `spec.strategy.customParams` attribute of the application deployment configuration. You can also customize environment variables and the command to execute for the deployment.

Integrating OpenShift Deployments with Life-cycle Hooks

The Recreate and Rolling strategies support *life-cycle hooks*. You can use these hooks to trigger events at predefined points in the deployment process. OpenShift deployments contain three life-cycle hooks:

Pre-Lifecycle Hook

OpenShift executes the pre-life-cycle hook before any new pods for a deployment start, and also before any older pods shut down.

Mid-Lifecycle Hook

The mid-life-cycle hook is executed after all the old pods in a deployment have been shut down, but before any new pods are started. Mid-Lifecycle Hooks are only available for the **Recreate** strategy.

Post-Lifecycle Hook

The post-life-cycle hook is executed after all new pods for a deployment have started, and after all the older pods have shut down.

Life-cycle hooks run in separate containers, which are short-lived. OpenShift automatically cleans them up after they finish executing. Automatic database initialization and database migrations are good use cases for life-cycle hooks.

Each hook has a `failurePolicy` attribute, which defines the action to take when a hook failure is encountered. There are three policies:

- Abort: The deployment process is considered a failure if the hook fails.
- Retry: Retry the hook execution until it succeeds.
- Ignore: Ignore any hook failure and allow the deployment to proceed.

Implementing Advanced Deployment Strategies Using the OpenShift Router

Advanced Deployment strategies that use the OpenShift router features are listed below:

Blue-Green Deployment

In Blue-Green deployments, you have two identical environments running concurrently, where each environment runs a different version of the application. The OpenShift router is used to direct traffic from the current in-production version (Green) to the newer updated version

(Blue). You can implement this strategy using a route and two services. Define a service for each specific version of the application.

The route points to one of the services at any given time, and can be changed to point to a different service when ready, or to facilitate a rollback. As a developer, you can test the new version of your application by connecting to the new service before routing your production traffic to it. When your new application version is ready for production, change the production router to point to the new service defined for your updated application.

A/B Deployment

The A/B deployment strategy allows you to deploy a new version of the application for a limited set of users in the production environment. You can configure OpenShift so that it routes the majority of requests to the currently deployed version in a production environment, while a limited number of requests go to the new version.

By controlling the portion of requests sent to each version as testing progresses, you can gradually increase the number of requests sent to the new version. Eventually, you can stop routing traffic to the previous version. As you adjust the request load on each version, the number of pods in each service may need to be scaled to provide the expected performance.

Additional things to consider when you are planning the deployment strategy for your application include *N-1 compatibility* and *graceful termination*:

N-1 Compatibility

Many deployment strategies require two versions of the application to be running at the same time. When running two versions of an application simultaneously, ensure that data written by the new code can be read and handled (or gracefully ignored) by the old version of the code; this is called N-1 Compatibility.

The data managed by the application can take many forms: data stored on disk, in a database, or in a temporary cache. Most well designed stateless web applications can support rolling deployments, but it is important to test and design your application to handle N-1 compatibility.

Graceful Termination

OpenShift allows application instances to shut down cleanly before removing them from the router's load-balancing roster. Applications must ensure they cleanly terminate user connections before they exit.

OpenShift sends a **TERM** signal to the processes in the container when it wants to shut it down. Application code, on receiving a **SIGTERM** signal, should stop accepting new connections. Stopping new connections ensures that the router can route traffic to other active instances. The application code should then wait until all open connections are closed (or gracefully terminate individual connections at the next opportunity) before exiting.

After the graceful termination period has expired, OpenShift sends a **KILL** signal to any process that has not exited. This immediately ends the process. The **terminationGracePeriodSeconds** attribute of a pod or pod template controls the graceful termination period (the default is 30 seconds), and can be customized per application.



References

Further information about deployment strategies is available in the *Deployments* chapter of the *Applications* guide for Red Hat OpenShift Container Platform 4.5; at https://access.redhat.com/documentation/en-us/openshift_container_platform/4.5/html-single/applications/index#deployments

An introduction to Blue-Green, canary, and rolling deployments

<https://opensource.com/article/17/5/colorful-deployments>

OpenShift Release Deployment Scenarios

<https://keithtenzer.com/2016/08/11/openshift-v3-basic-release-deployment-scenarios/>

Application Release Strategies with OpenShift

<https://access.redhat.com/articles/2897391>

Demonstration of custom deployment strategy with OpenShift

<https://www.youtube.com/watch?v=Plv17cvkEgQ>

Using Pipelines in OpenShift for CI/CD

<https://developers.redhat.com/blog/2017/01/09/using-pipelines-in-openshift-3-3-for-cicd/>

OpenShift Jenkins plugin

<https://github.com/openshift/jenkins-plugin>

► Guided Exercise

Implementing a Deployment Strategy

In this exercise, you will initialize a database using a deployment life-cycle hook.

Outcomes

You should be able to:

- Change the deployment strategy of a MySQL database deployment to **Recreate**.
- Add a post-deployment life-cycle hook to the deployment configuration to initialize the MySQL database with data from an SQL file.
- Troubleshoot and fix post-deployment life-cycle execution issues.

Before You Begin

To perform this exercise, ensure you have access to:

- A running OpenShift cluster.
- The MySQL 5.7 container image (**rhscl/mysql-57-rhel7**).

Run the following command on **workstation** to validate the exercise prerequisites, and to download the lab and solution files:

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

- 1. Create a new project, and deploy an application based on the **rhscl/mysql-57-rhel7** container image to the OpenShift cluster.

- 1.1. Load the configuration of your classroom environment.

Run the following command to load the environment variables created in the first guided exercise:

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

- 1.2. Log in to OpenShift using your developer user name:

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful.
...output omitted...
```

- 1.3. Create a new project for the application. Prefix the project name with your developer user name.

```
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-strategy
Now using project "youruser-strategy" on server "https://
api.cluster.domain.example.com:6443".
...output omitted...
```

- 1.4. Use the **oc new-app** command to create a new application called **mysql**.

Copy or execute the command from the **oc-new-app.sh** script in the **~/DO288/labs/strategy** folder:

```
[student@workstation ~]$ oc new-app --as-deployment-config --name mysql \
> -e MYSQL_USER=test -e MYSQL_PASSWORD=redhat -e MYSQL_DATABASE=testdb \
> --docker-image registry.access.redhat.com/rhscl/mysql-57-rhel7
--> Found Docker image ... "registry.access.redhat.com/rhscl/mysql-57-rhel7"
...output omitted...
--> Creating resources ...
...output omitted...
--> Success
...output omitted...
```

- 1.5. Wait until the MySQL pod is deployed. View the status of the pod. The pod should be in a *Running* state:

```
[student@workstation ~]$ oc get pods
NAME        READY     STATUS    RESTARTS   ...
mysql-1-deploy  0/1      Completed  0          ...
mysql-1-cx5hq   1/1      Running   0          ...
```

- 2. Patch the deployment configuration to change the default deployment strategy to **Recreate**.

- 2.1. Verify that the default deployment strategy for the MySQL application is **Rolling**:

```
[student@workstation ~]$ oc get dc/mysql -o jsonpath='{.spec.strategy.type}{"\n"}'
Rolling
```

- 2.2. In the following steps, you will make several changes to the deployment configuration. To prevent redeployment after every change, disable the configuration change triggers for the deployment:

```
[student@workstation ~]$ oc set triggers dc/mysql --from-config --remove
deploymentconfig.apps.openshift.io/mysql triggers updated
```

- 2.3. Change the default deployment strategy. You can copy the command from the **~/DO288/labs/strategy/recreate.sh** script, execute the script, or you can type the command, as follows:

```
[student@workstation ~]$ oc patch dc/mysql --patch \
> '{"spec":{"strategy":{"type":"Recreate"}}}'
deploymentconfig.apps.openshift.io/mysql patched
```

- 2.4. Remove the **rollingParams** attribute from the deployment configuration. You can copy or execute the command from the **~/D0288/labs/strategy/rm-rolling.sh** script:

```
[student@workstation ~]$ oc patch dc/mysql --type=json \
> -p='[{"op":"remove", "path": "/spec/strategy/rollingParams"}]' \
deploymentconfig.apps.openshift.io/mysql patched
```

- 3. Add a post life-cycle hook to initialize data for the MySQL database.
- 3.1. Briefly review the **~/D0288/labs/strategy/users.sql** file, which contains the data to insert into the MySQL database. This SQL script creates a table called **users** and inserts three rows of data:

```
CREATE TABLE IF NOT EXISTS users (
    user_id int(10) unsigned NOT NULL AUTO_INCREMENT,
    name varchar(100) NOT NULL,
    email varchar(100) NOT NULL,
    PRIMARY KEY (user_id)) ENGINE=InnoDB DEFAULT CHARSET=utf8;

insert into users(name,email) values ('user1', 'user1@example.com');
insert into users(name,email) values ('user2', 'user2@example.com');
insert into users(name,email) values ('user3', 'user3@example.com');
```

- 3.2. Briefly review the **~/D0288/labs/strategy/import.sh** script, which downloads and runs an SQL script to initialize the database. This script is executed in the post life-cycle hook:

```
#!/bin/bash
...output omitted...

echo 'Downloading SQL script that initializes the database...'
curl -s -O https://github.com/RedHatTraining/D0288-apps/releases/download/
OCP-4.1-1/users.sql

echo "Trying $HOOK_RETRIES times, sleeping $HOOK_SLEEP sec between tries:"
while [ "$HOOK_RETRIES" != 0 ]; do

    echo -n 'Checking if MySQL is up...'
    if mysqlshow -h$MYSQL_SERVICE_HOST -u$MYSQL_USER -p$MYSQL_PASSWORD -P3306
$MYSQL_DATABASE &>/dev/null
    then
        echo 'Database is up'
        break
    else
        echo 'Database is down'

        # Sleep to wait for the MySQL pod to be ready
        sleep $HOOK_SLEEP
    fi

    let HOOK_RETRIES=$HOOK_RETRIES-1
done
```

```

if [ "$HOOK_RETRIES" = 0 ]; then
    echo 'Too many tries, giving up'
    exit 1
fi

# Run the SQL script
if mysql -h$MYSQL_SERVICE_HOST -u$MYSQL_USER -p$MYSQL_PASSWORD -P3306
$MYSQL_DATABASE < /tmp/users.sql
then
    echo 'Database initialized successfully'
else
    echo 'Failed to initialize database'
    exit 2
fi

```

The script tries to connect to the database at most **HOOK_RETRIES** times, and sleeps **HOOK_SLEEP** seconds between attempts. The script must retry and sleep because the pod for the hook is created at the same time the database server pod is creating its database and user.

- 3.3. Briefly review the **~/D0288/labs/strategy/post-hook.sh** script, which adds a new post life-cycle hook to the deployment configuration:

```
[student@workstation ~]$ cat ~/D0288/labs/strategy/post-hook.sh
...output omitted...
oc patch dc/mysql --patch \
'{"spec": {"strategy": {"recreateParams": {"post": {"failurePolicy": "Abort", "execNewPod": {"containerName": "mysql", "command": ["/bin/sh", "-c", "curl -L -s https://github.com/RedHatTraining/D0288-apps/releases/download/OCP-4.1-1/import.sh -o /tmp/import.sh&&chmod 755 /tmp/import.sh&&/tmp/import.sh"]}}}}}'
```

Local copies of the **import.sh** and **users.sql** are provided for your reference. These files are downloaded from the classroom content server during redeployment because the post life-cycle hook executes in a separate pod.

- 3.4. Run the **~/D0288/labs/strategy/post-hook.sh** script:

```
[student@workstation ~]$ ~/D0288/labs/strategy/post-hook.sh
deploymentconfig.apps.openshift.io/mysql patched
```

- 4. Verify the patched deployment configuration and roll out the new deployment configuration.

- 4.1. Verify that the deployment strategy is now **Recreate**, and that there is a post life-cycle hook that executes the **import.sh** script:

```
[student@workstation ~]$ oc describe dc/mysql | grep -A 3 'Strategy:'
Strategy: Recreate
Post-deployment hook (pod type, failure policy: Abort):
  Container: mysql
  Command: /bin/sh -c curl -L -s ...
```

- 4.2. Force a new deployment to test changes to the strategy and the new post life-cycle hook:

```
[student@workstation ~]$ oc rollout latest dc/mysql
deploymentconfig.apps.openshift.io/mysql rolled out
```

- 4.3. Verify that a new MySQL pod comes up in a *Running* state, followed by the execution of the second deployment pod and also the post life-cycle hook in a separate pod:

```
[student@workstation ~]$ watch -n 2 oc get pods
NAME          READY   STATUS    RESTARTS   ...
mysql-2-deploy 1/1     Running   0          ...
mysql-2-hook-post 1/1     Running   0          ...
mysql-2-kbnpr   1/1     Running   0          ...
```

- 4.4. After a few seconds, the post life-cycle hook pod and the second deployment pod fail:

```
[student@workstation ~]$ watch -n 2 oc get pods
NAME          READY   STATUS    RESTARTS   ...
mysql-1-rcw95  1/1     Running   0          ...
mysql-2-deploy 0/1     Error    0          ...
mysql-2-hook-post 0/1     Error    0          ...
```

When both the hook and the deployment pods are in the error state, press **Ctrl+C** to exit the **watch** command.

Notice that the **mysql-1-rcw95** pod is a new pod. The second deployment terminated the original pod from the first deployment. After the second deployment failed, a new pod was created using the original deployment configuration from the first deployment.

► 5. Troubleshoot and fix the post life-cycle hook.

- 5.1. Display the logs from the failed pod. The logs show that the script tries to connect to the database only one time, and then returns an error status:

```
[student@workstation ~]$ oc logs mysql-2-hook-post
Downloading SQL script that initializes the database...
Trying 0 times, sleeping 2 sec between tries:
Too many tries, giving up
```

- 5.2. Notice the script incorrectly has a value of 0 for the **HOOK_RETRIES** variable, causing it to never connect to the database. Increase the number of times the script attempts to connect to the database server. Set the **HOOK_RETRIES** environment variable in the deployment configuration to a value of **5**.

```
[student@workstation ~]$ oc set env dc/mysql HOOK_RETRIES=5
deploymentconfig.apps.openshift.io/mysql updated
```

- 5.3. Start a third deployment to run the hook a second time, using the new values for the environment variables:

```
[student@workstation ~]$ oc rollout latest dc/mysql
deploymentconfig.apps.openshift.io/mysql rolled out
```

- 5.4. Wait until the new post life-cycle hook pod is in a status of **Completed**:

```
[student@workstation ~]$ watch -n 2 oc get pods
NAME          READY   STATUS    RESTARTS   ...
mysql-1-deploy 0/1     Completed  0          5m26s
mysql-2-deploy 0/1     Error     0          3m30s
mysql-2-hook-post 0/1     Error     0          3m8s
mysql-3-29jwt  1/1     Running   0          57s
mysql-3-deploy  0/1     Completed  0          79s
mysql-3-hook-post 0/1     Completed  0          48s
```

- 5.5. Open a new terminal to display the logs from the running post life-cycle hook pod. They show that the script can connect to the database a few times, and then returns a success status:

```
[student@workstation ~]$ oc logs -f mysql-3-hook-post
Downloading SQL script that initializes the database...
Trying 5 times, sleeping 2 sec between tries:
Checking if MySQL is up...Database is up
mysql: [Warning] Using a password on the command line interface can be insecure.
Database initialized successfully
```

The number of tries depends on your hardware and the nodes each pod is scheduled to run by OpenShift.

If the hook connected the first time, then the pod is terminated when you try to see its logs. You can move to the next step.

- 5.6. After a few seconds, the new database pod is the one created using the latest changes to the deployment configuration:

NAME	READY	STATUS	RESTARTS	...
mysql-1-deploy	0/1	Completed	0	7m46s
mysql-2-deploy	0/1	Error	0	5m50s
mysql-2-hook-post	0/1	Error	0	5m28s
mysql-3-29jwt	1/1	Running	0	3m17s
mysql-3-deploy	0/1	Completed	0	3m39s
mysql-3-hook-post	0/1	Completed	0	3m8s

After the **mysql-3-hook-post** pod runs and exits, press **Ctrl+C** to exit the **watch** command.

Notice that OpenShift retains the failed pods that were created during the previous deployment attempts, so you can display their logs for troubleshooting.

- ▶ 6. Verify that the new MySQL database pod contains the data from the SQL file.

- 6.1. Open a shell session to the MySQL container pod. Use the **oc get pods** command to get the name of the current MySQL pod:

```
[student@workstation ~]$ oc get pods
NAME          READY   STATUS    RESTARTS   AGE
...output omitted...
mysql-3-3p4m1  1/1     Running   0          8m
[student@workstation ~]$ oc rsh mysql-3-3p4m1
sh-4.2$
```

- 6.2. Verify that the **users** table has been created and populated with data from the SQL file:

```
sh-4.2$ mysql -u$MYSQL_USER -p$MYSQL_PASSWORD $MYSQL_DATABASE
...output omitted...
Server version: 5.7.16 MySQL Community Server (GPL)
...output omitted...
mysql> select * from users;
+-----+-----+
| user_id | name   | email           |
+-----+-----+
|       1 | user1  | user1@example.com |
|       2 | user2  | user2@example.com |
|       3 | user3  | user3@example.com |
+-----+-----+
3 rows in set (0.00 sec)
```

- 6.3. Exit the MySQL session and the container shell:

```
mysql> exit
Bye
sh-4.2$ exit
exit
[student@workstation ~]$
```

- 7. Clean up. Delete the **strategy** project in OpenShift:

```
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-strategy
```

Finish

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

```
[student@workstation ~]$ lab strategy finish
```

This concludes the guided exercise.

Managing Application Deployments with CLI Commands

Objectives

After completing this section, you should be able to manage the deployment of an application with CLI commands.

Deployment Configuration

A deployment configuration defines the template for a pod and manages the deployment of new images or configuration changes whenever the attributes are changed. Deployment configurations can support many different deployment patterns, including full restart, customizable rolling updates, as well as pre- and post-life-cycle hooks.

OpenShift automatically creates a replication controller that represents the deployment configuration pod template when you create a deployment configuration. When the deployment configuration changes, OpenShift creates a new replication controller with the latest pod template, and a deployment process runs to scale down the old replication controller and scale up the new replication controller. OpenShift also automatically adds and removes instances of the application from both service load balancers and routers as it starts or stops them.

A deployment configuration is declared within a **DeploymentConfig** attribute in a resource file, which can be in YAML or JSON format. Use the **oc** command to manage the deployment configuration like any other OpenShift resource. The following template shows a deployment configuration in YAML format:

```
kind: "DeploymentConfig"
apiVersion: "v1"
metadata:
  name: "frontend" ①
spec:
...
  replicas: 5 ②
  selector:
    name: "frontend"
  triggers:
    - type: "ConfigChange" ③
    - type: "ImageChange" ④
      imageChangeParams:
...
  strategy:
    type: "Rolling"
...

```

- ① The deployment configuration name.
- ② The number of replicas to run.
- ③ A configuration change trigger that causes a new replication controller to be created when there are changes to the deployment configuration.

- ④ An image change trigger that causes a new replication controller to be created each time a new version of the image is available.

Managing Deployments Using CLI Commands

Several command-line options are available to manage deployments. The following list describes the available options:

- To start a deployment, use the **oc rollout** command. The **latest** option indicates that the newest version of the template must be used:

```
[user@host ~]$ oc rollout latest dc/name
```

This option is often used to start a new deployment or upgrade an application to the latest version.

- To view the history of deployments for a specific deployment configuration, use the **oc rollout history** command:

```
[user@host ~]$ oc rollout history dc/name
```

- To access details about a specific deployment, append the **--revision** parameter to the **oc rollout history** command:

```
[user@host ~]$ oc rollout history dc/name --revision=1
```

- To access details about a deployment configuration and its latest revision, use the **oc describe dc** command:

```
[user@host ~]$ oc describe dc name
```

- To cancel a deployment, run the **oc rollout** command with the **cancel** option:

```
[user@host ~]$ oc rollout cancel dc/name
```

You may want to cancel a deployment if it is taking too long to start, there are inconsistencies in the log file, or the deployment is affecting the behavior of other resources in the system.



Warning

When canceled, the deployment configuration is automatically rolled back to the previous running replication controller.

- To retry a deployment that failed, run the **oc rollout** command with the **retry** option:

```
[user@host ~]$ oc rollout retry dc/name
```

You may retry a deployment after previously canceling that deployment, or finding an error that causes the deployment to fail, if you want to keep the same revision.

**Note**

When you retry a deployment, it restarts the deployment process but does not create a new deployment revision. OpenShift also restarts the replication controller with the same configuration it had when it failed.

- To use a previous version of the application you can roll back the deployment with the **oc rollback** command:

```
[user@host ~]$ oc rollback dc/name
```

If there is a problem with the latest deployment, such as users complaining about a new feature that does not work as expected, you can revert to a previous known working version of the application using the **oc rollback** command.

**Note**

If no revision is specified with the **--to-version** parameter, the last successfully deployed revision is used.

**Note**

Deployment configurations support automatically rolling back to the last successful revision of the configuration in case the latest deployment process fails. In that case, the latest template that failed to deploy stays intact and is available for review.

- To prevent accidentally starting a new deployment process after a rollback is complete, image change triggers are disabled as part of the rollback process. However, after a rollback, you can re-enable image change triggers with the **oc set triggers** command:

```
[user@host ~]$ oc set triggers dc/name --auto
```

- To view deployment logs, use the **oc logs** command:

```
[user@host ~]$ oc logs -f dc/name
```

If the latest revision is running or failed, the **oc logs** command returns the logs of the process that is responsible for deploying your pods. If it is successful, it returns the logs from a pod of your application.

You can also view logs from older failed deployment processes, provided they have not been pruned or deleted manually:

```
[user@host ~]$ oc logs --version=1 dc/name
```

- You can scale the number of pods in a deployment using the **oc scale** command:

```
[user@host ~]$ oc scale dc/name --replicas=3
```

The number of replicas eventually propagates to the desired and current state configured by the deployment configuration.

Deployment Triggers

A deployment configuration can contain *triggers*, which drive the creation of new deployments in response to events, both inside and outside of OpenShift. There are two types of events that trigger a deployment:

- Configuration change
- Image change

Configuration Change Trigger

The **ConfigChange** trigger results in a new deployment whenever changes are detected to the replication controller template of the deployment configuration. You can rely on this trigger to be activated after changing replica size, changing the image to use for the application, or other changes made to the deployment configuration.

An example of a **ConfigChange** trigger is shown below:

```
triggers:  
  - type: "ConfigChange"
```

Image Change Trigger

The **ImageChange** trigger results in a new deployment whenever the value of an image stream tag changes. This is useful in an environment where images are updated independently from the application code for security reasons or library updates.

An example of an **ImageChange** trigger is shown below:

```
triggers:  
  - type: "ImageChange"  
    imageChangeParams:  
      automatic: true①  
      containerNames:  
        - "helloworld"  
      from:  
        kind: "ImageStreamTag"  
        name: "origin-ruby-sample:latest"
```

- ① If the **automatic** attribute is set to false, the trigger is disabled.

In the previous example, when the **latest** tag value of the **origin-ruby-sample** image stream changes, a new deployment is created using the new tag value for container.

Use the **oc set triggers** command to set a deployment trigger for a deployment configuration. For example, to set the **ImageChange** trigger, run the following command:

```
[user@host ~]$ oc set triggers dc/name \  
> --from-image=myproject/origin-ruby-sample:latest -c helloworld
```

Setting Deployment Resource Limits

A deployment is completed by a pod that consumes resources (memory and CPU) on a node. By default, pods consume unlimited node resources. However, if a project specifies default resource limits, then pods only consume resources up to those limits.

You can also limit resource use by specifying resource limits as part of the deployment strategy. These resource limits apply to the application pods created by the deployment, but not to deployer pods. You can use deployment resources with the Recreate, Rolling, or Custom deployment strategies.

In the following example, resources required for the deployment are declared under the **resources** attribute of the deployment configuration:

```
type: "Recreate"
resources:
  limits:
    cpu: "100m" ①
    memory: "256Mi" ②
```

- ① CPU resource in CPU units. **100m** equals 0.1 CPU units.
- ② Memory resource in bytes. **256Mi** equals 268435456 bytes ($256 * 2^20$).



References

Additional information about deployments is available in the *Deployments* chapter of the *Applications* guide for Red Hat OpenShift Container Platform 4.5; at https://access.redhat.com/documentation/en-us/openshift_container_platform/4.5/html-single/applications/index#deployments

► Guided Exercise

Managing Application Deployments

In this exercise, you will manage the deployment of an application that runs on an OpenShift cluster.

Outcomes

You should be able to:

- Deploy a Thorntail-based application to an OpenShift cluster.
- Update the deployment configuration for the running application to include a liveness probe.
- Make changes to the application source and redeploy the application.
- Roll back the application to the previously deployed version.

Before You Begin

To perform this exercise, ensure you have access to:

- A running OpenShift cluster.
- The **redhat-openjdk-18/openjdk18-openshift** OpenJDK S2I builder image.
- The quip application in the Git repository.

Run the following command on **workstation** to validate the exercise prerequisites, and to download the lab and solution files:

```
[student@workstation ~]$ lab app-deploy start
```

► 1. Review the application source code.

1. Enter your local clone of the **DO288-apps** Git repository and check out the **master** branch of the course repository to ensure that you start this exercise from a known good state:

```
[student@workstation ~]$ cd DO288-apps
[student@workstation DO288-apps]$ git checkout master
...output omitted...
```

- 1.2. Create a new branch where you can save any changes you make during this exercise:

```
[student@workstation D0288-apps]$ git checkout -b app-deploy
Switched to a new branch 'app-deploy'
[student@workstation D0288-apps]$ git push -u origin app-deploy
...output omitted...
* [new branch]      app-deploy -> app-deploy
Branch app-deploy set up to track remote branch app-deploy from origin.
[student@workstation D0288-apps]$ cd ~
```

- 1.3. Inspect the **Quip.java** file in the **~/D0288-apps/quip/src/main/java/com/redhat/training/example** directory.

The quip application is a very basic Java JAX-RS REST service implementation, with two endpoints:

```
...output omitted...
@Path("/")
public class Quip {

    @GET
    @Produces("text/plain")
    public Response index() throws Exception {
        String host = InetAddress.getLocalHost().getHostName();
        return Response.ok("Veni, vidi, vici...\n").build();①
    }

    @GET
    @Path("/ready")
    @Produces("text/plain")
    public Response ready() throws Exception {
        return Response.ok("OK\n").build();②
    }
...output omitted...
```

- ① Requests to the root URL ('/') endpoint return a simple quote.
- ② Requests to the **ready** endpoint return an **OK** message.

▶ 2. Create an application based on the source code.

- 2.1. Load your classroom environment configuration.

Run the following command to load the environment variables created in the first guided exercise:

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

- 2.2. Log in to OpenShift using your developer user account:

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful
...output omitted...
```

- 2.3. Create a new project for the application. Prefix the project name with your developer user name:

```
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-app-deploy
```

- 2.4. Create an application using the `oc new app` command. The command is also provided for you in the `/home/student/D0288/labs/app-deploy/oc-new-app.sh` file. You can run this script directly:

```
[student@workstation ~]$ oc new-app --as-deployment-config --name quip \
> --build-env MAVEN_MIRROR_URL=http://${RHT_OCP4_NEXUS_SERVER}/repository/java \
> -i redhat-openjdk18-openshift:1.5 \
> https://github.com/${RHT_OCP4_GITHUB_USER}/D0288-apps#app-deploy \
> --context-dir quip
--> Found image c1bf724 (9 months old) in image stream "openshift/redhat-
openjdk18-...
--> Creating resources ...
  imagestream.image.openshift.io "quip" created
  buildconfig.build.openshift.io "quip" created
  deploymentconfig.apps.openshift.io "quip" created
  service "quip" created
--> Success
...output omitted...
```

- 2.5. View the application build logs. It will take some time to build the application container image and push it to the OpenShift internal registry:

```
[student@workstation ~]$ oc logs -f bc/quip
...output omitted...
[INFO] Repackaged .war: /tmp/src/target/quip.war
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
...output omitted...
Pushing image ....registry.svc:5000/youruser-app-deploy/quip:latest ...
...output omitted...
Push successful
```

- 2.6. Wait until the application is deployed. The application pod must be in a *Running* state, and marked as ready:

```
[student@workstation ~]$ oc get pods
NAME        READY   STATUS    RESTARTS   AGE
quip-1-build  0/1     Completed  0          2m17s
quip-1-deploy 1/1     Completed  0          53s
quip-1-v8gq4  1/1     Running   0          45s
```

- 3. Test the application to verify that it serves requests from clients.

- 3.1. Review the application logs to see if there were any errors during startup:

```
[student@workstation ~]$ oc logs dc/quip
...output omitted...
... INFO [org.jboss.as.server] (main) WFLYSRV0010: Deployed "quip.war" (...)
... INFO [org.wildfly.swarm] (main) WFSWARM99999: Thorntail is Ready
```

The logs show that the application started without any errors.

- 3.2. Verify that the OpenShift service for the application has a registered endpoint to route incoming requests:

```
[student@workstation ~]$ oc describe svc/quip
Name:           quip
Namespace:      youruser-app-deploy
Labels:         app=quip
Annotations:   openshift.io/generated-by: OpenShiftNewApp
Selector:       app=quip,deploymentconfig=quip
Type:          ClusterIP
IP:            172.30.37.127
Port:          8080-tcp  8080/TCP
TargetPort:    8080/TCP
Endpoints:     10.128.2.111:8080
...output omitted...
```

- 3.3. Expose the application for external access by using a route:

```
[student@workstation ~]$ oc expose svc quip
route.route.openshift.io/quip exposed
```

- 3.4. Obtain the route URL using the **oc get route** command:

```
[student@workstation ~]$ oc get route/quip \
> -o jsonpath='{.spec.host}{"\n"}'
quip-youruser-app-deploy.apps.cluster.domain.example.com
```

- 3.5. Test the application using the route URL you obtained in the previous step:

```
[student@workstation ~]$ curl \
> http://quip-${RHT_OCP4_DEV_USER}-app-deploy.${RHT_OCP4_WILDCARD_DOMAIN}
Veni, vidi, vici...
```

▶ 4. Activate readiness and liveness probes for the application.

- 4.1. Use the **oc set** command to add a liveness and readiness probe to the Deployment Config.

```
[student@workstation ~]$ oc set probe dc/quip --liveness \
> --get-url=http://:8080/ready \
> --initial-delay-seconds=30 --timeout-seconds=2
deploymentconfig.apps.openshift.io/quip probes updated
[student@workstation ~]$ oc set probe dc/quip --readiness \
> --get-url=http://:8080/ready \
> --initial-delay-seconds=30 --timeout-seconds=2
deploymentconfig.apps.openshift.io/quip probes updated
```

- 4.2. Verify the value in the **livenessProbe** and **readinessProbe** entries:

```
[student@workstation ~]$ oc describe dc/quip
Name: quip
...output omitted...
  Liveness: http-get http://:8080/ready delay=30s timeout=2s...
  Readiness: http-get http://:8080/ready delay=30s timeout=2s...
...output omitted...
```

- 4.3. Wait for the application pod to redeploy and appear in a **Running** state:

```
[student@workstation ~]$ oc get pods
NAME      READY   STATUS    RESTARTS   AGE
...output omitted...
quip-3-hppqxi  1/1     Running      0          6s
```

- 4.4. Use the **oc describe** command to inspect the running pod and ensure the probes are active:

```
[student@workstation ~]$ oc describe pod quip-3-hppqxi | grep http-get
  Liveness: http-get http://:8080/ready delay=30s timeout=2s...
  Readiness: http-get http://:8080/ready delay=30s timeout=2s...
```

The readiness and liveness probes for the application are now active.

- 4.5. Test the application using the route URL you obtained in the previous step:

```
[student@workstation ~]$ curl \
> http://quip-$RHT_OCP4_DEV_USER-app-deploy.$RHT_OCP4_WILDCARD_DOMAIN]
Veni, vidi, vici...
```

- ▶ 5. Make some changes to the application source. Verify that you can see the changes when you test the application.

- 5.1. Review the script provided at **~/D0288/labs/app-deploy/app-change.sh**. The script changes the source code to print the message in English. The script changes the message and then commits and pushes the change to the classroom Git repository. Briefly review the script:

```
[student@workstation ~$ cat ~/D0288/labs/app-deploy/app-change.sh
#!/bin/bash
```

```

echo "Changing quip to english..."
sed -i 's/Veni, vidi, vici/I came, I saw, I conquered/g' \
/home/student/D0288-apps/quip/src/main/java/com/redhat/training/example/Quip.java

echo "Committing the changes..."
cd /home/student/D0288-apps/quip
git commit -a -m "Changed quip lang to english"

echo "Pushing changes to classroom Git repository..."
git push
cd ~

```

- 5.2. Run the `~/D0288/labs/app-deploy/app-change.sh` script:

```

[student@workstation ~]$ ~/D0288/labs/app-deploy/app-change.sh
Changing quip to english...
Committing the changes...
[app-deploy afdf7c3] Changed quip lang to english
...output omitted...
To https://github.com/youruser/D0288-apps
  dfe07f7..0aa1ac1  app-deploy -> app-deploy

```

- 5.3. Start a new build of the application and follow the build logs:

```

[student@workstation ~]$ oc start-build quip -F
build.build.openshift.io/quip-2 started
...output omitted...
Push successful

```

- 5.4. Wait until a new application pod is deployed. The pod must be in a *Running* state. The pod should also be marked ready, as follows:

NAME	READY	STATUS	RESTARTS	AGE
quip-1-build	0/1	Completed	0	5m
quip-1-deploy	0/1	Completed	0	5m
quip-2-build	0/1	Completed	0	2m45s
quip-3-deploy	0/1	Completed	0	2m15s
quip-4-deploy	0/1	Completed	0	41s
quip-4-vbdr4	1/1	Running	0	32s

- 5.5. Retest the application after the change, and verify that a message in English is printed:

```

[student@workstation ~]$ curl \
> http://quip-${RHT_OCP4_DEV_USER}-app-deploy.${RHT_OCP4_WILDCARD_DOMAIN}
I came, I saw, I conquered...

```

- 6. Roll back to the previous deployment. Verify that you see the older message (the one you saw before making the change).

- 6.1. Roll back to the previous version of the deployment. You will see a warning message that image change triggers are disabled by the **oc rollback** command:

```
[student@workstation ~]$ oc rollback dc/quip
deploymentconfig.apps.openshift.io/quip deployment #5 rolled back to quip-3
Warning: the following images triggers were disabled: quip:latest
You can re-enable them with: oc set triggers dc/quip --auto
```

- 6.2. Wait for the new application pod to deploy. It must be in a *Running* state. The pod should also be marked ready, as follows:

NAME	READY	STATUS	RESTARTS	AGE
quip-1-build	0/1	Completed	0	6m
quip-1-deploy	0/1	Completed	0	6m
quip-3-build	0/1	Completed	0	4m
quip-3-deploy	0/1	Completed	0	4m
quip-4-deploy	0/1	Completed	0	2m
quip-5-deploy	0/1	Completed	0	45s
quip-5-z7lg5	1/1	Running	0	17s

- 6.3. After you have rolled back the application, retest it and verify that the Latin message is printed:

```
[student@workstation ~]$ curl \
> http://quip-${RHT_OCP4_DEV_USER}-app-deploy.${RHT_OCP4_WILDCARD_DOMAIN}
Veni, vidi, vici...
```

► 7. Clean up. Delete the **youruser-app-deploy** project in OpenShift:

```
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-app-deploy
project.project.openshift.io "youruser-app-deploy" deleted
```

Finish

On **workstation**, run the **lab app-deploy finish** command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab app-deploy finish
```

This concludes the guided exercise.

► Lab

Managing Application Deployments

Performance Checklist

In this lab, you will manage the deployment of an application, and scale it on an OpenShift cluster.

Outcomes

You should be able to:

- Deploy a PHP-based application to an OpenShift cluster.
- Scale the application to run in multiple pods.
- Modify the application source, redeploy the application, and verify that the changes are reflected.
- Roll back a change and verify that the previous version of the application is deployed.

Before You Begin

To perform this lab, ensure you have access to:

- A running OpenShift cluster.
- The `php-scale` application source code in the Git repository.

Run the following command on **workstation** to validate the prerequisites:

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

Requirements

The application is written in the PHP. The application prints a simple string message with the version of the application, and the name and IP address of the pod on which it is running. You must deploy and test the application on an OpenShift cluster according to the following instructions:

- Use the **php:7.3** image stream to deploy the application.
- Ensure that the application name for OpenShift is `scale`.
- Create the application in a project named **`youruser-manage-deploy`**.
- Ensure that the application is accessible at the URL:

`http://scale-youruser-manage-deploy.apps.cluster.domain.example.com`.

- Ensure that the application uses the Git repository at:

`https://github.com/youruser/D0288-apps`.

- The **php-scale** directory in the Git repository contains the source code for the application.

Steps

- Enter your local clone of the **D0288-apps** Git repository and check out the **master** branch of the course repository to ensure you start this exercise from a known good state:

```
[student@workstation ~]$ cd D0288-apps
[student@workstation D0288-apps]$ git checkout master
...output omitted...
```

- Create a new branch where you can save any changes you make during this exercise:

```
[student@workstation D0288-apps]$ git checkout -b manage-deploy
Switched to a new branch 'manage-deploy'
[student@workstation D0288-apps]$ git push -u origin manage-deploy
...output omitted...
* [new branch]      manage-deploy -> manage-deploy
Branch manage-deploy set up to track remote branch manage-deploy from origin.
```

- Log in to the OpenShift cluster using your personal development user name.

Create a new project named **youruser-manage-deploy** and deploy the application using the **php:7.3** image stream. Ensure that you reference the **manage-deploy** branch you created in the previous step when you deploy the application.

Expose and test the application using the generated route URL. Verify that you can see **version 1** and the pod name in the output.

- Verify that the **Rolling** strategy is the default deployment strategy.
- Log in to the OpenShift web console using your personal development user name. Scale the application to two pods. Use the **curl** command to retest the application, and verify that the requests are round-robin load-balanced across the two pods.
- Change the version number in the source to 2. Do not make any other changes to the source code. Commit the changes to the Git repository.
- Use the web console to start a new build of the application. Verify that OpenShift scales down the pods running the older version, and scales up two new pods with the latest version of the application. Retest the application using the **curl** command. Verify that **version 2** is in the output.
- Roll back to the previous deployment. Use the **curl** command to retest the application. Verify that **version 1** is seen in the output.
- Grade your work.

Run the following command on **workstation** to verify that all tasks were accomplished:

```
[student@workstation D0288-apps]$ lab manage-deploy grade
```

- Clean up and delete the project.

Finish

On **workstation**, run the **lab manage-deploy finish** script to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises. The finish action releases this project and its resources.

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

This concludes the review lab.

► Solution

Managing Application Deployments

Performance Checklist

In this lab, you will manage the deployment of an application, and scale it on an OpenShift cluster.

Outcomes

You should be able to:

- Deploy a PHP-based application to an OpenShift cluster.
- Scale the application to run in multiple pods.
- Modify the application source, redeploy the application, and verify that the changes are reflected.
- Roll back a change and verify that the previous version of the application is deployed.

Before You Begin

To perform this lab, ensure you have access to:

- A running OpenShift cluster.
- The `php-scale` application source code in the Git repository.

Run the following command on **workstation** to validate the prerequisites:

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

Requirements

The application is written in the PHP. The application prints a simple string message with the version of the application, and the name and IP address of the pod on which it is running. You must deploy and test the application on an OpenShift cluster according to the following instructions:

- Use the **php:7.3** image stream to deploy the application.
- Ensure that the application name for OpenShift is `scale`.
- Create the application in a project named **`youruser-manage-deploy`**.
- Ensure that the application is accessible at the URL:
`http://scale-youruser-manage-deploy.apps.cluster.domain.example.com`.
- Ensure that the application uses the Git repository at:
`https://github.com/youruser/D0288-apps`.
- The **php-scale** directory in the Git repository contains the source code for the application.

Steps

- Enter your local clone of the **D0288-apps** Git repository and check out the **master** branch of the course repository to ensure you start this exercise from a known good state:

```
[student@workstation ~]$ cd D0288-apps
[student@workstation D0288-apps]$ git checkout master
...output omitted...
```

- Create a new branch where you can save any changes you make during this exercise:

```
[student@workstation D0288-apps]$ git checkout -b manage-deploy
Switched to a new branch 'manage-deploy'
[student@workstation D0288-apps]$ git push -u origin manage-deploy
...output omitted...
* [new branch]      manage-deploy -> manage-deploy
Branch manage-deploy set up to track remote branch manage-deploy from origin.
```

- Log in to the OpenShift cluster using your personal development user name.

Create a new project named **youruser-manage-deploy** and deploy the application using the **php:7.3** image stream. Ensure that you reference the **manage-deploy** branch you created in the previous step when you deploy the application.

Expose and test the application using the generated route URL. Verify that you can see **version 1** and the pod name in the output.

- Load your classroom environment configuration.

Run the following command to load the environment variables created in the first guided exercise:

```
[student@workstation D0288-apps]$ source /usr/local/etc/ocp4.config
```

- Log in to OpenShift and create a new project. Prefix the project name with your developer user name.

```
[student@workstation D0288-apps]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful.
...output omitted...
[student@workstation D0288-apps]$ oc new-project \
> ${RHT_OCP4_DEV_USER}-manage-deploy
Now using project "yourusername-manage-deploy" on server "https://
api.cluster.domain.example.com:6443".
```

- Create a new application:

```
[student@workstation D0288-apps]$ oc new-app --as-deployment-config --name scale \
> php:7.3-https://github.com/${RHT_OCP4_GITHUB_USER}/D0288-apps#manage-deploy \
> --context-dir php-scale
--> Found image f10275b (3 weeks old) in image stream "openshift/php" under...
...output omitted...
--> Creating resources ...
```

Chapter 7 | Managing Application Deployments

```
imagestream.image.openshift.io "scale" created
buildconfig.build.openshift.io "scale" created
deploymentconfig.apps.openshift.io "scale" created
service "scale" created
--> Success
...output omitted...
```

- 3.4. View the build logs. Wait until the build finishes and the application container image is pushed to the OpenShift registry:

```
[student@workstation D0288-apps]$ oc logs -f bc/scale
...output omitted...
Push successful
```

- 3.5. Wait until the application is deployed. View the status of the application pod. The application pod should be in a *Running* state:

```
[student@workstation D0288-apps]$ oc get pods
NAME        READY   STATUS    RESTARTS   AGE
scale-1-build  0/1     Completed  0          5m14s
scale-1-deploy  0/1     Completed  0          82s
scale-1-w48nd   1/1     Running   0          74s
```

- 3.6. Use a route to expose the application to external access:

```
[student@workstation D0288-apps]$ oc expose svc scale
route.route.openshift.io/scale exposed
```

- 3.7. Obtain the route URL using the **oc get route** command:

```
[student@workstation D0288-apps]$ oc get route scale \
> -o jsonpath='{.spec.host}{"\n"}'
scale-youruser-manage-deploy.apps.cluster.domain.example.com
```

- 3.8. Test the application with the **curl** command:

```
[student@workstation D0288-apps]$ curl \
> http://scale-${RHT_OCP4_DEV_USER}-manage-deploy.${RHT_OCP4_WILDCARD_DOMAIN}
This is version 1 of the app. I am running on host -> scale-1-gp3w0 (10.128.1.21)
```

4. Verify that the **Rolling** strategy is the default deployment strategy.

```
[student@workstation D0288-apps]$ oc describe dc/scale | grep 'Strategy'
Strategy: Rolling
```

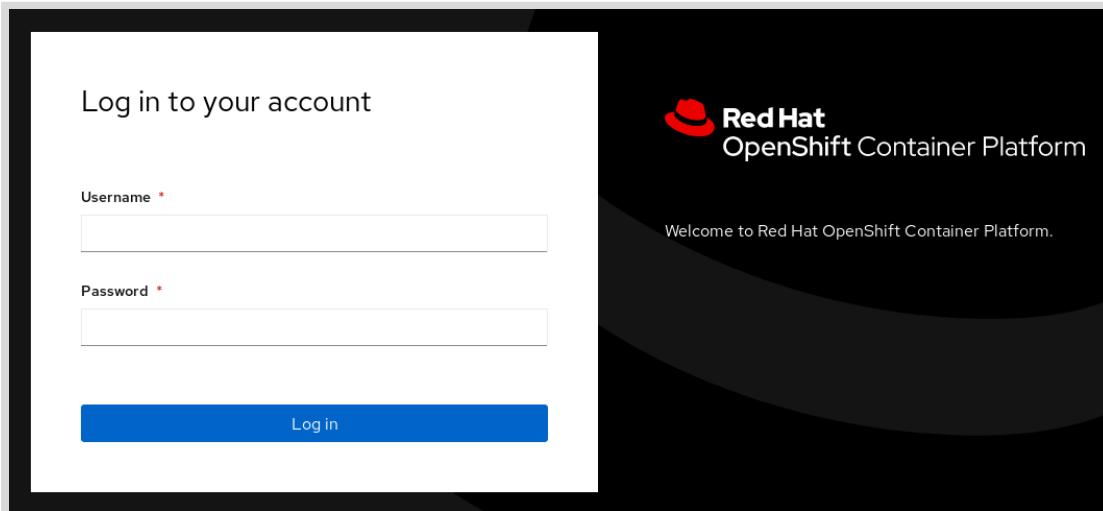
5. Log in to the OpenShift web console using your personal development user name. Scale the application to two pods. Use the **curl** command to retest the application, and verify that the requests are round-robin load-balanced across the two pods.

- 5.1. Open a web browser and navigate to the OpenShift web console.

Find the host name of your OpenShift web console inspecting the routes on the **openshift-console** project.

```
[student@workstation DO288-apps]$ oc get route console -n openshift-console \
> -o jsonpath='{.spec.host}{"\n"}'
console-openshift-console.apps.cluster.domain.example.com
```

- 5.2. Log in as the developer user. Your user name (*youruser*) is the **RHT_OCP4_DEV_USER** variable in the **/usr/local/etc/ocp4.config** classroom configuration file. Your password is the value of the **RHT_OCP4_DEV_PASSWORD** variable in the same file.



- 5.3. Click **youruser-manage-deploy** on the **Projects** page to open the **Overview** page for the project.
Click **Workloads** to show the deployments available for the project.
- 5.4. Select the **scale** deployment configuration entry to display its details. In the **Details** section, click the upper arrow on the right of the blue circle to increase the number of pods to two.

The screenshot shows the OpenShift Workloads interface for a project named 'youruser-manage-deploy'. The 'Workloads' tab is selected. A search bar at the top left says 'Group by: Application'. Below it, a filter bar says 'Filter by name...'. A list item 'scale' is expanded, showing a sub-item 'DC scale, #1' with '1 of 1 pods'. To the right, a 'Health Checks' panel indicates that the container scale does not have health checks. The main content area shows a table with one row:

Name	Latest Version
scale	1

Watch as OpenShift creates another pod. This might take several minutes.

- 5.5. Return to the terminal window, and use the **curl** command to make multiple HTTP requests to the route URL to test the application:

```
[student@workstation D0288-apps]$ curl \
> http://scale-${RHT_OCP4_DEV_USER}-manage-deploy.${RHT_OCP4_WILDCARD_DOMAIN}
This is version 1 of the app. I am running on host -> scale-1-gp3w0 (10.128.1.21)
[student@workstation D0288-apps]$ curl \
> http://scale-${RHT_OCP4_DEV_USER}-manage-deploy.${RHT_OCP4_WILDCARD_DOMAIN}
This is version 1 of the app. I am running on host -> scale-1-567x7 (10.129.1.101)
```

You should see the requests being round-robin load-balanced across the two pods.



Note

You cannot use a web browser to test the route URL, because the OpenShift router enables session stickiness by default. Therefore you cannot verify the load-balancing behavior. Test the application using the **curl** command.

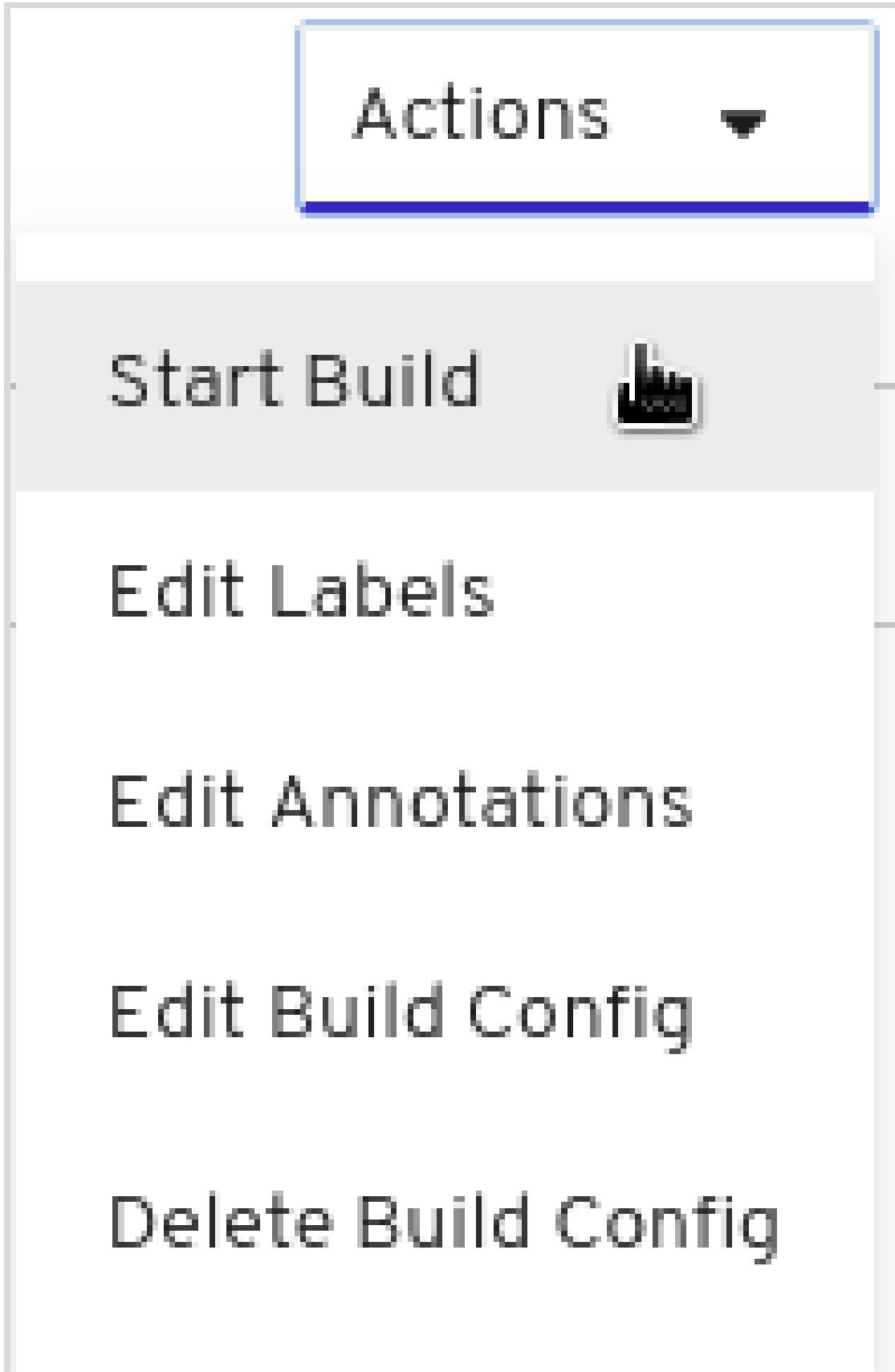
6. Change the version number in the source to 2. Do not make any other changes to the source code. Commit the changes to the Git repository.
 - 6.1. Edit the **~/D0288-apps/php-scale/index.php** file and change the version number (on the second line) to 2, as shown below. Do not change anything else in this file:

```
<?php  
    print "This is version 2 of the app. I am running on host...  
?>
```

- 6.2. Commit the changes and push to the Git repository:

```
[student@workstation DO288-apps]$ git commit -a -m "Updated app to version 2"  
[manage-deploy 3633f74] updating version  
 1 file changed, 1 insertion(+), 1 deletion(-)  
[student@workstation DO288-apps]$ git push
```

7. Use the web console to start a new build of the application. Verify that OpenShift scales down the pods running the older version, and scales up two new pods with the latest version of the application. Retest the application using the **curl** command. Verify that **version 2** is in the output.
- 7.1. In the left menu of the OpenShift web console, open the **Builds** menu, and then select the **Build Configs** entry. Select the **scale** build configuration, and then use the **Actions** drop-down in the upper-right corner and click **Start Build**:



- 7.2. The console redirects you to the summary page about the new build. Use the **Logs** button to watch the build log as the new application container image is built and pushed to the internal OpenShift registry.
- 7.3. After the build has finished, OpenShift scales up two new pods of the new version of the application, checks that the pods are ready to receive traffic, and then scales down the two older pods. You should see two pods running with the new version.
- 7.4. Return to the terminal window and use the **curl** command to make multiple HTTP requests to the route URL to test the application:

```
[student@workstation D0288-apps]$ curl \
> http://scale-${RHT_OCP4_DEV_USER}-manage-deploy.${RHT_OCP4_WILDCARD_DOMAIN}
This is version 2 of the app. I am running on host -> scale-2-w7nfz (10.128.1.27)
[student@workstation D0288-apps]$ curl \
> http://scale-${RHT_OCP4_DEV_USER}-manage-deploy.${RHT_OCP4_WILDCARD_DOMAIN}
This is version 2 of the app. I am running on host -> scale-2-dxswv (10.129.1.119)
```

You should see version 2 in the output. You should also see the requests being round-robin load-balanced across the two pods. Note that the pod name and IP addresses are different from the older deployment.

8. Roll back to the previous deployment. Use the **curl** command to retest the application. Verify that **version 1** is seen in the output.

- 8.1. Roll back to the previous version of the deployment. You will get a warning message that image change triggers are disabled by the **oc rollback** command:

```
[student@workstation D0288-apps]$ oc rollback dc/scale
deploymentconfig.apps.openshift.io/scale deployment #3 rolled back to scale-1
Warning: the following images triggers were disabled: scale:latest
You can re-enable them with: oc set triggers dc/scale --auto
```

- 8.2. Wait for the new application pod to deploy. It must be in a *Running* state. The pod should also be marked ready, as follows:

[student@workstation D0288-apps]\$ oc get pods				
NAME	READY	STATUS	RESTARTS	AGE
scale-1-build	0/1	Completed	0	40m
scale-1-deploy	0/1	Completed	0	36m
scale-2-build	0/1	Completed	0	10m
scale-2-deploy	0/1	Completed	0	6m59s
scale-3-bcxpg	1/1	Running	0	58s
scale-3-deploy	0/1	Completed	0	77s
scale-3-lnp46	1/1	Running	0	68s

- 8.3. Return to the terminal window, and use the **curl** command to make multiple HTTP requests to the route URL to test the application:

```
[student@workstation D0288-apps]$ curl \
> http://scale-${RHT_OCP4_DEV_USER}-manage-deploy.${RHT_OCP4_WILDCARD_DOMAIN}
This is version 1 of the app. I am running on host -> scale-3-bcxpg (10.128.2.133)
[student@workstation D0288-apps]$ curl \
> http://scale-${RHT_OCP4_DEV_USER}-manage-deploy.${RHT_OCP4_WILDCARD_DOMAIN}
This is version 1 of the app. I am running on host -> scale-3-lnp46 (10.131.1.206)
```

You should see version 1 in the output. You should also see the requests being round-robin load-balanced across the two pods. Note that the pod name and IP addresses are different from the previous deployment.

9. Grade your work.

Run the following command on **workstation** to verify that all tasks were accomplished:

```
[student@workstation D0288-apps]$ lab manage-deploy grade
```

10. Clean up and delete the project.

```
[student@workstation D0288-apps]$ oc delete project \
> ${RHT_OCP4_DEV_USER}-manage-deploy
```

Finish

On **workstation**, run the **lab manage-deploy finish** script to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises. The finish action releases this project and its resources.

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

This concludes the review lab.

Summary

In this chapter, you learned:

- Readiness and liveness probes monitor the health of your applications.
- Use the **Rolling** strategy when you can simultaneously run two versions of your application to upgrade with no downtime. This strategy first scales up additional pods with the new version, and then once ready, scales down the pods with the older version.
- Use the **Recreate** strategy when you cannot run two versions of your application at the same time. This strategy shuts down all pods with the previous version and then starts additional pods with the newer version.
- Use the **Custom** strategy to customize the deployment process when the two strategies OpenShift provides do not suit your needs.
- Both the **Recreate** and **Rolling** strategies support life-cycle hooks, which allow you to customize the deployment at various points within the deployment process.
- You can limit resource usage for application deployments by specifying resource limits as part of the deployment strategy.

Chapter 8

Implementing Continuous Integration and Continuous Deployment Pipelines in OpenShift

Goal

Create and deploy Jenkins pipelines to facilitate continuous integration and deployment with OpenShift.

Objectives

- Describe principles of continuous integration and continuous deployment.
- Deploy Jenkins to manage pipelines in OpenShift.
- Create and run a custom Jenkins pipeline.

Sections

- Describing CI/CD Concepts (and Quiz)
- Implementing Jenkins Pipelines on OpenShift (and Guided Exercise)
- Writing Custom Jenkins Pipelines (and Guided Exercise)

Lab

Implementing Continuous Integration and Continuous Deployment Pipelines in OpenShift

Describing CI/CD Concepts

Objectives

After completing this section, you should be able to describe the principles of continuous integration and continuous deployment.

Introducing Continuous Integration, Continuous Delivery and Continuous Release

Continuous Integration (CI), *Continuous Delivery (CD)*, *Continuous Release (CR)* are three practices put to use in DevOps teams. The combination of CI and CD, commonly called CI/CD, allows you to frequently and reliably deliver applications to customers by introducing automation into the stages of development and deployment. These two techniques are a solution to the problem of continuously integrating new code, which can be quite cumbersome and costly when not adequately automated. The CR practice is an optional final step where you automatically push changes to production following a successful CI/CD integration and deployment.

CI/CD introduces ongoing automation and continuous monitoring throughout the life cycle of applications, from integration and testing phases to delivery and deployment. The process of implementing this automation can be called a *CI/CD pipeline*. Ideally, development and operations teams work together to build and support this automation using agile methodologies.

Continuous Integration

Continuous Integration (CI) is an automation process targeting developers to improve the quality of their code and the reliability of their applications. CI works best when developers integrate their code into a shared source code repository like Git or Subversion several times a day. Each commit is then verified by an automated build, allowing teams to detect problems early.

Automating application tests allows developers to avoid spending valuable time manually validating each change that they make. CI is also a solution to the problem of having too many branches of an application in development concurrently, where functionality in some of these branches might conflict with each other, or cause unwanted side effects.

In the traditional waterfall model of software development, developers merge their source code from their branches into the main branch infrequently, and typically all at once (this approach is called "big bang integration"). The resulting process can be tedious, manual, error-prone, and time-intensive, as the code merging often has many conflicts.

A better approach is to adopt a more agile process of continuous integration, where developers regularly commit their changes so that their changes are tested and verified automatically. In a typical agile project, tasks are broken down into much smaller units of work which are more easily completed within a short time-frame, allowing team members to integrate their work with the whole project up to several times a day, producing a number of benefits to the project, such as:

- Smaller tasks are more easily reassignable, and have outcomes that are generally easier to test in an automated manner.
- If automated testing discovers a conflict between new and existing code, the CI pipeline makes it easier to identify and fix those bugs quickly.

- Small tasks contribute to transparency of the project's progress, which allows everyone to stay up-to-date with the application design and direction.
- Small tasks that are failed typically produce less damage to the overall project in the form of delays or negatively impacting the codebase by requiring lots of refactoring.

This approach has some clear requirements in order to work reliably:

- Developers must agree to use a source code management (SCM) repository - their source code is regularly merged into common code base in that repository;
- There must be sufficiently extensive unit tests for each individual component developed to ensure that a single component is working according to specification. Some teams decide to use test-driven design (TDD) to fulfill this requirement.
- There must be very basic, but representative integration tests that exercise logical functions of several modules in conjunction, such as making a purchase. These tests are usually performed at the lowest possible layer, such as simply invoking business logic components internally to the application, and mocking any external dependencies such as databases and other web services.

An example CI pipeline could be as follows:

1. The pipeline is triggered by a new commit to the source code repository.
2. The pipeline builds the entire application directly from the source code. Linting and static code analysis tools are run in this stage.
3. Unit and integration tests are run to verify that the new code does not break existing functionality, or introduce new bugs.
4. If all the stages complete successfully, the application is ready for deployment into a test or staging environment, where more end to end tests (both manual and automated) occur and, if approved, the team can automatically promote the application to production.

Continuous Delivery

The goal of Continuous Delivery (CD), is to have code in a state that is always ready for deployment to a production environment.

In the traditional waterfall process, deploying code to production is a complex, tedious, manual process with long checklists, involving multiple teams and lots of debugging after the code is deployed, to ensure it works as intended.

In a continuous delivery pipeline, every stage, from the merging of code changes to the delivery of production-ready builds, involves test automation and code release automation. At the end of the process, the operations team can confidently deploy small incremental changes to an application to production quickly and efficiently.

For continuous delivery to be effective, it is essential that your pipeline already contain a continuous integration process. Without reliable automated testing and verification of your application, you cannot confidently continuously deploy it. An example CD pipeline could be as follows:

1. Assuming that all the stages of the continuous integration pipeline are successful, deploy the application to a QA or staging environment.
2. Run automated and manual end to end tests in the staging environment. You can add an optional stage where a human operator manually checks that the tests are complete, and that the application functionality meets the expectation of end users.

3. Optionally, deploy ("promote") the application to the production environment.

Continuous Release

The goal of Continuous Release (CR), also called "Continuous Deployment" (not to be confused with Continuous Delivery), is to push all of your changes to your end users as quickly and efficiently as possible. To achieve CR means carefully replacing manual deployment steps following a production sign-off, with automated ones, such that any service level requirements are maintained. Instead of someone (or a team of people) coordinating each step of the application deployment to production environment, these steps must be implemented in an automated way.

There are a number of additional strategies that facilitate successful Continuous Release, including but not limited to:

- Blue-Green production environments, where post-deployment smoke tests are performed in the inactive environment prior to roll-over.
- A/B testing scenarios where multiple different versions of an application may be running at the same time, and only a small portion of the customers is exposed to the new version for a while, allowing for quick rollback and minimal damage in case of problems.
- Different deployment strategies (such as rolling deployments), where the exact steps of a rollout depend on the nature of the application, and even possible data structure changes, either of which can prevent us from being able to run more than one version (or even more than one instance) of the application at the same time.
- Composite applications where different backend modules serve each of the UI components (such as the catalog, item preview, suggestions, etc.), allowing for independent upgrades of individual modules.
- Defensive application coding, including the use of microservice fault tolerance patterns such as circuit breaker, and fallback responses in the event of a missing dependency.
- Clear and precise smoke tests, along with reliable rollback procedures for every step of the application deployment.
- Ongoing performance and functional monitoring for every application module, and every logical function.

It is important to remember that, while CR is a wonderful practice if implemented, it may not be suitable for everything. There are applications, particularly legacy monolithic ones, but also others, that can not easily be modularized, can not easily scale horizontally, thus providing a poor foundation for a typical rolling deployment with zero downtime.

There are even use-cases where implementing CR is more trouble than it is worth. Consider a typical offline back-office environment, where the business operates on a very predictable schedule, giving the operations team plenty of opportunity for scheduled outages, rather than risking the deployment of a new version during working hours, potentially disrupting hundreds or thousands of customers. However, even in such environments, having some (or most) of the building blocks for CR in place can immensely simplify and speed up the manual deployment process, and make it more reliable.



References

What is CI/CD?

<https://www.redhat.com/en/topics/devops/what-is-ci-cd>

CI/CD with OpenShift

<https://blog.openshift.com/cicd-with-openshift/>

► Quiz

CI/CD Concepts

Choose the correct answers to the following questions:

► 1. **Arrange the following stages in a continuous integration pipeline for a Java application in the correct order:**

1. Compile the code using Apache Maven, and then run a code analysis tool to ensure the code adheres to your organizations Java coding standards.
2. Tag the source code. Create a zip file of the code repository and store it as an artifact in a Nexus server, so that you can repeatedly build the application in the staging environment.
3. Set up webhooks to enable kicking off the pipeline when code is committed to your source code repository.
4. Run JUnit tests and integration tests.
 - a. 3, 2, 4, 1
 - b. 2, 3, 1, 4
 - c. 3, 1, 4, 2
 - d. 3, 2, 1, 4

► 2. **Which of the two following stages are good candidates for a continuous integration pipeline that rapidly delivers Node.js microservices? (Choose two)**

- a. Run ESLint tooling on the entire code base and report the issues.
- b. Run webpack tooling on the code base and deploy the microservice to a Content Delivery Network (CDN) server serving static files.
- c. Add a Node Package Manager (NPM) dependency to the project that automatically restarts the Node.js server when code changes are detected.
- d. Run **npm test** in the project to run all unit tests.
- e. Run **npm db-generate** to generate a scaffolded database access code for the project.

► **3. Arrange the following stages in the correct order for a continuous integration pipeline for a Python application:**

1. Run **pylint** tooling on the Python project and report the issues.
 2. Run unit tests using the **pytest** framework.
 3. Add Git tags to the project to enable reproducible builds.
 4. Checkout application source code from a Git repository.
- a. 4, 2, 3, 1
 - b. 4, 3, 1, 2
 - c. 4, 3, 2, 1
 - d. 4, 1, 2, 3

► **4. Which of the two following stages are good candidates for a continuous integration pipeline that rapidly delivers PHP microservices deployed on an Apache web server? (Choose two)**

- a. Automatically restart the Apache web server when source code changes are detected.
- b. Run unit tests using the **PHPUnit** framework.
- c. Automatically tune the Apache web server based on increasing or decreasing load.
- d. Automatically format and indent code to adhere to PHP 7 coding standards.
- e. Send an email when end users hit an error condition in the application resulting in an **HTTP 500 - Server Error** status.
- f. Run cron jobs at set intervals to gather statistics about the users accessing the application.

► Solution

CI/CD Concepts

Choose the correct answers to the following questions:

► 1. **Arrange the following stages in a continuous integration pipeline for a Java application in the correct order:**

1. Compile the code using Apache Maven, and then run a code analysis tool to ensure the code adheres to your organizations Java coding standards.
2. Tag the source code. Create a zip file of the code repository and store it as an artifact in a Nexus server, so that you can repeatedly build the application in the staging environment.
3. Set up webhooks to enable kicking off the pipeline when code is committed to your source code repository.
4. Run JUnit tests and integration tests.
 - a. 3, 2, 4, 1
 - b. 2, 3, 1, 4
 - c. 3, 1, 4, 2
 - d. 3, 2, 1, 4

► 2. **Which of the two following stages are good candidates for a continuous integration pipeline that rapidly delivers Node.js microservices? (Choose two)**

- a. Run ESLint tooling on the entire code base and report the issues.
- b. Run webpack tooling on the code base and deploy the microservice to a Content Delivery Network (CDN) server serving static files.
- c. Add a Node Package Manager (NPM) dependency to the project that automatically restarts the Node.js server when code changes are detected.
- d. Run **npm test** in the project to run all unit tests.
- e. Run **npm db-generate** to generate a scaffolded database access code for the project.

► **3. Arrange the following stages in the correct order for a continuous integration pipeline for a Python application:**

1. Run **pylint** tooling on the Python project and report the issues.
 2. Run unit tests using the **pytest** framework.
 3. Add Git tags to the project to enable reproducible builds.
 4. Checkout application source code from a Git repository.
- a. 4, 2, 3, 1
b. 4, 3, 1, 2
c. 4, 3, 2, 1
d. 4, 1, 2, 3

► **4. Which of the two following stages are good candidates for a continuous integration pipeline that rapidly delivers PHP microservices deployed on an Apache web server? (Choose two)**

- a. Automatically restart the Apache web server when source code changes are detected.
- b. Run unit tests using the **PHPUnit** framework.
- c. Automatically tune the Apache web server based on increasing or decreasing load.
- d. Automatically format and indent code to adhere to PHP 7 coding standards.
- e. Send an email when end users hit an error condition in the application resulting in an **HTTP 500 - Server Error** status.
- f. Run cron jobs at set intervals to gather statistics about the users accessing the application.

Implementing Jenkins Pipelines on OpenShift

Objectives

After completing this section, you should be able to deploy Jenkins to manage pipelines in OpenShift.

Introducing Jenkins on OpenShift

OpenShift provides support for creating, deploying, and managing CI/CD pipelines using Jenkins. You can create complex CI/CD pipelines that automate the workflow of rapidly deploying your application to production.

OpenShift comes with built-in support for running Jenkins pipelines. Pipelines describe the automated process for building, testing, deploying, and promoting your applications on OpenShift. Pipelines are described using a *Jenkinsfile*, which encapsulates the automation steps in your pipeline.

Jenkins provides a Pipeline plug-in, which provides a feature to describe pipeline steps using a domain specific language (DSL) based on the Groovy programming language. OpenShift comes with several other plug-ins for Jenkins that can help you create, deploy, and run pipelines for any type of application that you want to deploy.

List of Jenkins plug-ins

OpenShift Jenkins Client Plugin

The OpenShift Jenkins Client Plugin aims to provide a simple and concise domain specific language, using Jenkins Pipeline syntax for rich interactions with OpenShift. The plug-in leverages the OpenShift command line tool (**oc**), which must be available on the Jenkins slave nodes executing the pipeline. The Jenkins slave nodes are container images that can be run on OpenShift.

This plug-in is installed and enabled by default when using the inbuilt OpenShift Container Platform Jenkins image. The plug-in enables OpenShift Container Platform specific functions are available to use within the Jenkinsfile for your pipeline.

OpenShift Jenkins Sync Plugin

The OpenShift Jenkins Sync Plugin acts as a bridge between the OpenShift cluster and the Jenkins instance, and keeps build configuration and build objects in sync with Jenkins jobs and builds.

OpenShift Login Plugin

The OpenShift Login Plugin integrates the authentication and authorization of your Jenkins instance with your OpenShift cluster, providing a single sign-on functionality. You will access the Jenkins dashboard using the same credentials that you use to for the OpenShift web console.

Running Jenkins Pipelines on OpenShift

OpenShift provides the *JenkinsPipelineStrategy* as a build type that allows you to define a pipeline for execution by Jenkins. The build can be started, monitored, and managed by OpenShift in the same way as any other build type.

Pipeline workflows are defined in a Jenkinsfile, either embedded directly in the build configuration, or supplied in a Git repository and referenced by the build configuration. It is a recommended practice to keep the Jenkinsfile as a separate file at the root of your application source code tree and reference it in the build configuration.



Note

The Jenkins pipeline build strategy is deprecated in OpenShift Container Platform 4. The equivalent functionality is present in the OpenShift Pipelines based on Tekton. Despite of the deprecation, Jenkins images on OpenShift are still fully supported.

The following steps will create and run a simple Jenkins pipeline on OpenShift:

1. You first need to deploy an instance of Jenkins on your OpenShift cluster. Create a project to host the Jenkins instance:

```
[user@host ~]$ oc new-project myjenkins
```

2. OpenShift comes with ready to use templates for deploying Jenkins:

```
[user@host ~]$ oc get templates -n openshift | grep jenkins
jenkins-ephemeral      Jenkins service, without persistent storage....
jenkins-persistent      Jenkins service, with persistent storage....
```

3. Deploy the Jenkins instance:

```
[user@host ~]$ oc new-app --as-deployment-config jenkins-ephemeral
```

It will take some time for Jenkins to be deployed. Wait until the Jenkins pods are running and ready.

```
[user@host ~]$ oc get pods
NAME           READY   STATUS    RESTARTS   AGE
jenkins-1-deploy  0/1     Completed  0          13d
jenkins-1-rtnw5  1/1     Running   0          13d
```

4. Get the route URL for the Jenkins dashboard:

```
[user@host ~]$ oc get route
jenkins-myjenkins.apps.cluster.domain.example.com
```

5. Navigate to the route URL from the previous step using a web browser. You should be prompted to log in using your OpenShift credentials.

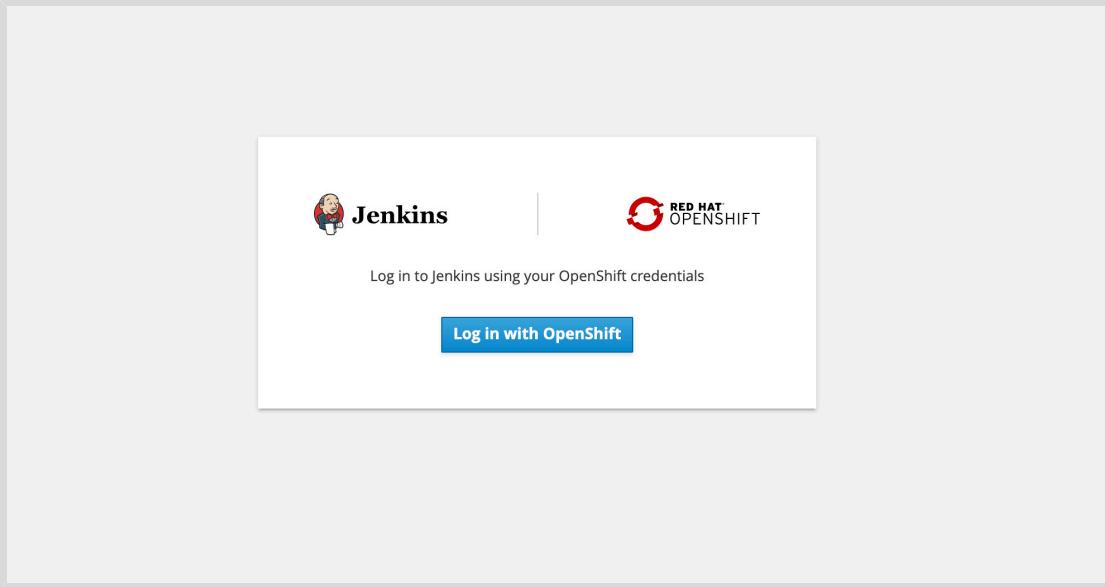


Figure 8.1: Jenkins login with OpenShift credentials

Once you log in, you will be prompted to authorize access to your account:

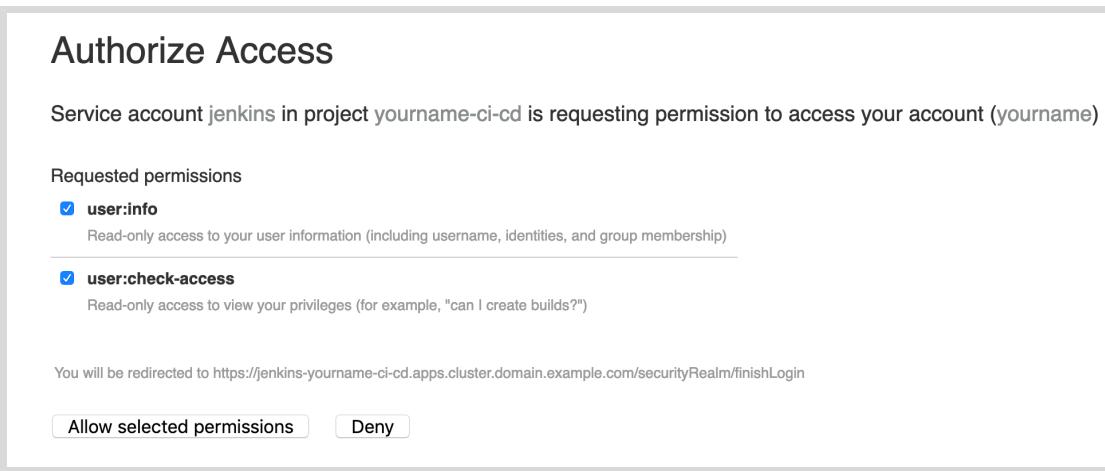


Figure 8.2: Authorize access to service account

Allow the service account to access your account details to view the Jenkins dashboard.

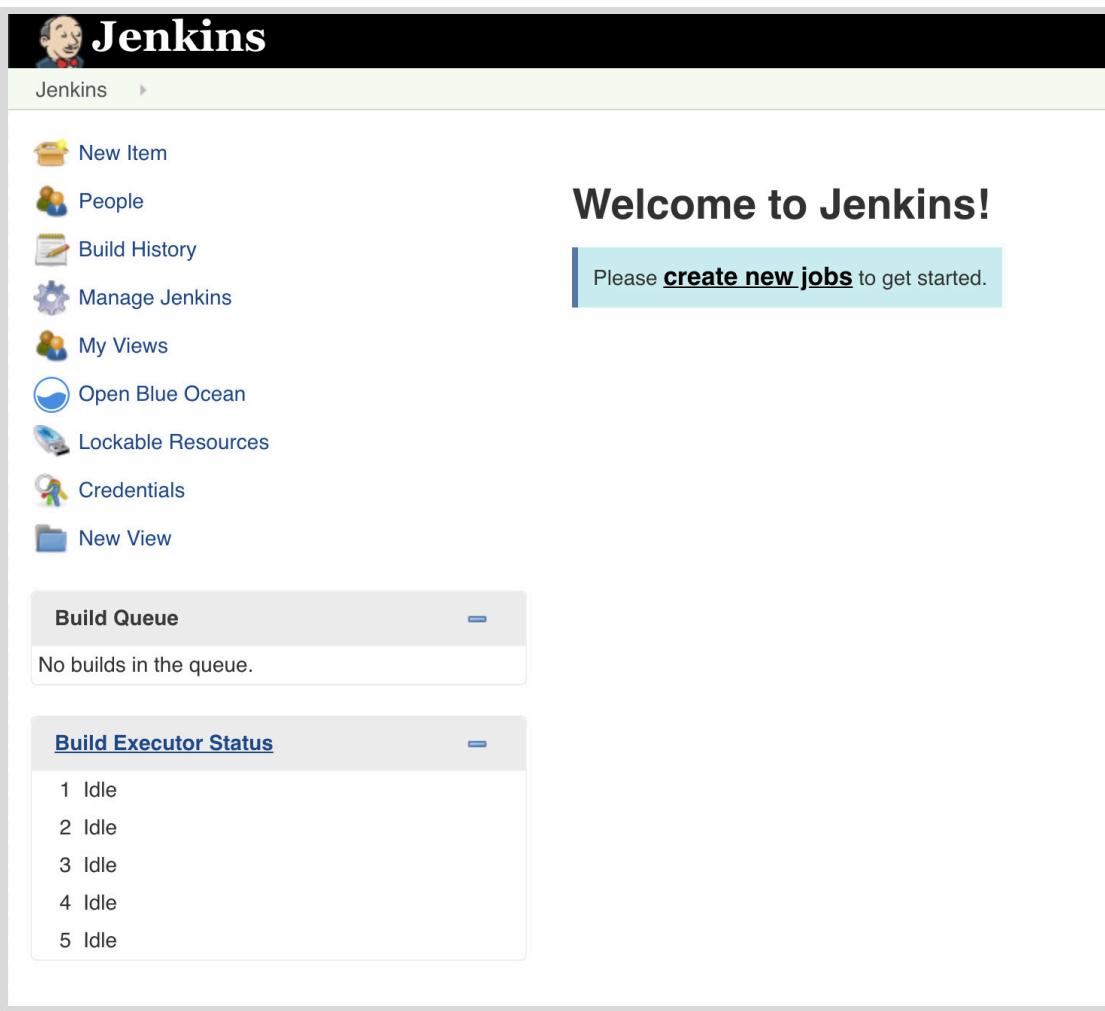
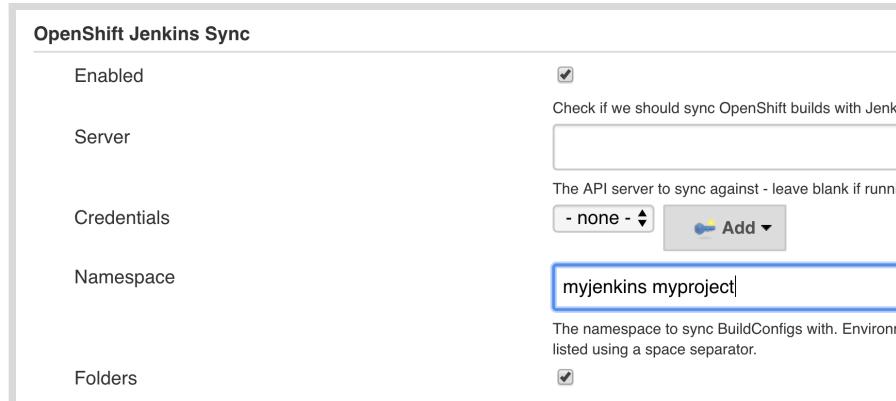


Figure 8.3: Jenkins home page

6. You need to configure the Jenkins Sync Plugin to monitor your project for pipelines, as well to allow Jenkins to interact with the OpenShift cluster to create and modify resources.

To configure the Jenkins Sync Plugin, click **Manage Jenkins** in the left menu of the Jenkins home page, and then click **Configure System** in the **Manage Jenkins** page to bring up the Jenkins global configuration page.

7. Scroll down to the **OpenShift Jenkins Sync** section, and add the project where you are deploying the pipeline to the **Namespace** field, next to the already existing **myjenkins** project. Separate the entries by a space, and do not use a comma. The project does not need to exist in OpenShift when you perform this step. You will create the project in subsequent steps.

**Figure 8.4: Jenkins Sync Plugin Configuration**

Click **Save** to apply the changes to the Jenkins Sync Plugin configuration.

8. Create a new project to deploy the application. The name of the project should be in the list of projects monitored by the Jenkins Sync Plugin client.

```
[user@host ~]$ oc new-project myproject
```

9. The service account associated with the Jenkins deployment must have the **edit** role for each project monitored by Jenkins. Add the **edit** role to the service account associated with the Jenkins deployment for your project.

```
[user@host ~]$ oc policy add-role-to-user \
> edit system:serviceaccount:myjenkins:jenkins \
> -n myproject
```

10. Create a build configuration for the pipeline. An example build configuration looks like the following:

```
kind: "BuildConfig"
apiVersion: "build.openshift.io/v1"
metadata:
  name: "mypipeline" ①
spec:
  source:
    contextDir: jenkins ②
    git:
      uri: "https://mygit/myapp" ③
      ref: "master"
  strategy:
    jenkinsPipelineStrategy: ④
    type: JenkinsPipeline ⑤
```

- ① The name for the build configuration.
- ② The directory relative to the root of the Git repository where the Jenkinsfile is stored. This attribute is optional. OpenShift looks for a Jenkinsfile at the root of the Git repository by default.
- ③ The URL of the Git repository where the application source code is stored. The Jenkinsfile is usually kept in the same Git repository.

④ ⑤ The strategy and type for this build configuration, indicating that it is a Jenkins pipeline.

Create the build configuration:

```
[user@host ~]$ oc create -f mypipeline-bc.yaml
```

11. Start a new build:

```
[user@host ~]$ oc start-build mypipeline
```

12. Once the build is running, you can monitor the progress of the pipeline execution from the **Builds** → **Builds** page in the OpenShift web console.
13. To view details about the pipeline execution, you can use the Jenkins dashboard to view the build log for each build. OpenShift creates a folder in the Jenkins dashboard for each project where pipelines are run.



References

OpenShift Jenkins Container Image

<https://github.com/openshift/jenkins>

Jenkins OpenShift Login Plugin

<https://github.com/openshift/jenkins-openshift-login-plugin>

OpenShift Jenkins Sync Plugin

<https://github.com/openshift/jenkins-sync-plugin>

Further information about the pipeline build strategy is in the *Using build strategies* chapter of the *Builds* guide for Red Hat OpenShift Container Platform 4.5; at https://docs.openshift.com/container-platform/4.5/builds/build-strategies.html#builds-strategy-pipeline-build_build-strategies

► Guided Exercise

Run a Simple Jenkins Pipeline

In this exercise, you will deploy Jenkins to manage pipelines in OpenShift, and then create a simple Jenkins pipeline.

Outcomes

You should be able to:

- Deploy a Jenkins server instance on OpenShift.
- Create and run a simple Jenkins pipeline using the **Pipeline** build strategy.

Before You Begin

To perform this exercise, ensure that you have access to:

- A running OpenShift cluster.
- The sample Jenkins pipeline in the Git repository (**simple-pipeline**).

Run the following command on **workstation** to validate the prerequisites and download solution files:

```
[student@workstation ~]$ lab simple-pipeline start
```

► 1. Deploy a Jenkins instance on OpenShift.

1.1. Load your classroom environment configuration.

Run the following command to load the environment variables created in the first guided exercise:

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

1.2. Log in to OpenShift using your developer user account.

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful.
...output omitted...
```

1.3. Inspect the Jenkins templates that are available in OpenShift.

```
[student@workstation ~]$ oc get templates -n openshift | grep jenkins
jenkins-ephemeral    Jenkins service, without persistent storage....
jenkins-persistent   Jenkins service, with persistent storage....
```

For this exercise, you will use the **jenkins-ephemeral** template.

- 1.4. Create a new project to host the Jenkins instance. Prefix the project name with your developer user name:

```
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-ci-cd
Now using project "youruser-ci-cd" on server
"https://api.cluster.domain.example.com:6443".
...output omitted...
```

- 1.5. Deploy an instance of Jenkins.

```
[student@workstation ~]$ oc new-app --as-deployment-config \
> jenkins-ephemeral -p MEMORY_LIMIT=2048Mi
--> Deploying template "openshift/jenkins-ephemeral" to project youruser-ci-cd
...output omitted...
--> Creating resources ...
...output omitted...
--> Success
...output omitted...
```

The Jenkins instance will take some time to deploy. Move on to the next steps to inspect and edit the build configuration while Jenkins is deploying.



Note

The **MEMORY_LIMIT** parameter increases the amount of memory allocated to the jenkins container to improve performance. The JVM maximum heap size is set to 50% of the value of **MEMORY_LIMIT** by default. The default value of **MEMORY_LIMIT** is 1GB.

- 2. Edit the build configuration for the pipeline.

- 2.1. Inspect the build configuration resource file at **~/D0288/labs/simple-pipeline/simple-pipeline.yaml** using a text editor:

```
kind: "BuildConfig"
apiVersion: "build.openshift.io/v1"
metadata:
  name: "simple-pipeline" ①
spec:
  source:
    contextDir: simple-pipeline ②
    git:
      uri: "https://github.com/yourgituser/D0288-apps" ③
      ref: "simple-pipeline"
  strategy:
    jenkinsPipelineStrategy: ④
  type: JenkinsPipeline ⑤
```

- ① The name for the build configuration.
- ② The directory relative to the root of the Git repository where the Jenkinsfile is stored.
- ③ The URL of the Git repository where the application source code is stored.

- ④ ⑤ The strategy and type for this build configuration, indicating that it is a Jenkins pipeline.

- 2.2. Edit the Git URL where the application source is stored.

Edit the build configuration and change the value of the `spec.source.git.uri` attribute to point to your Git repository:

```
git:
  uri: "https://github.com/yourgituser/D0288-apps"
  ref: "simple-pipeline"
```

► 3. Inspect the Jenkins pipeline.

- 3.1. Enter your local clone of the **D0288-apps** Git repository and checkout the **master** branch of the course repository to ensure you start this exercise from a known good state:

```
[student@workstation ~]$ cd D0288-apps
[student@workstation D0288-apps]$ git checkout master
...output omitted...
```

- 3.2. Create a new branch to save any changes you make during this exercise:

```
[student@workstation D0288-apps]$ git checkout -b simple-pipeline
Switched to a new branch 'simple-pipeline'
[student@workstation D0288-apps]$ git push -u origin simple-pipeline
...output omitted...
* [new branch]      simple-pipeline -> simple-pipeline
Branch simple-pipeline set up to track remote branch simple-pipeline from origin.
```

- 3.3. Inspect the Jenkinsfile.

Edit the `~/D0288-apps/simple-pipeline/Jenkinsfile` file in a text editor. The Jenkinsfile defines a very basic 4 stage pipeline. The first (**stage 1**), and the last (**stage 3**) stages are complete.

In the subsequent steps, you will complete the pipeline by implementing the lines marked as **TODO**: in the file.

► 4. Complete the Jenkins pipeline.

- 4.1. Edit the `~/D0288-apps/simple-pipeline/Jenkinsfile` file and perform the following steps. You can also copy these instructions from the provided `~/D0288/solutions/simple-pipeline/Jenkinsfile` file.
- 4.2. Add a label to the **node** section to enable the pipeline to run on a node labeled **master**. Locate the line:

```
// TODO: run this simple pipeline on jenkins 'master' node
```

Replace it with the following:

```
...output omitted...
agent {
  node {
    label 'master'
  }
}
...output omitted...
```

- 4.3. Add a new stage called **stage 2** after **stage 1**. The new stage simply prints a hello message. Add the following lines:

```
...output omitted...
stage('stage 2') {
  steps {
    sh 'echo hello from stage 2!'
  }
}
...output omitted...
```

- 4.4. Add a stage after **stage 2** that asks for manual approval before executing **stage 3**. Add the following lines:

```
...output omitted...
stage('manual approval') {
  steps {
    timeout(time: 60, unit: 'MINUTES') {
      input message: "Move to stage 3?"
    }
  }
}
...output omitted...
```

- 4.5. Save the Jenkinsfile and commit the changes to the Git repository from the `~/D0288-apps/simple-pipeline` folder:

```
[student@workstation D0288-apps]$ cd simple-pipeline
[student@workstation simple-pipeline]$ git commit -a -m "Completed Jenkinsfile"
...output omitted...
[student@workstation simple-pipeline]$ git push
...output omitted...
[student@workstation simple-pipeline]$ cd ~
```

- 5. Configure the Jenkins Sync Plugin, and allow Jenkins to monitor your project for pipelines.

- 5.1. Verify that the Jenkins instance you deployed earlier is ready and running:

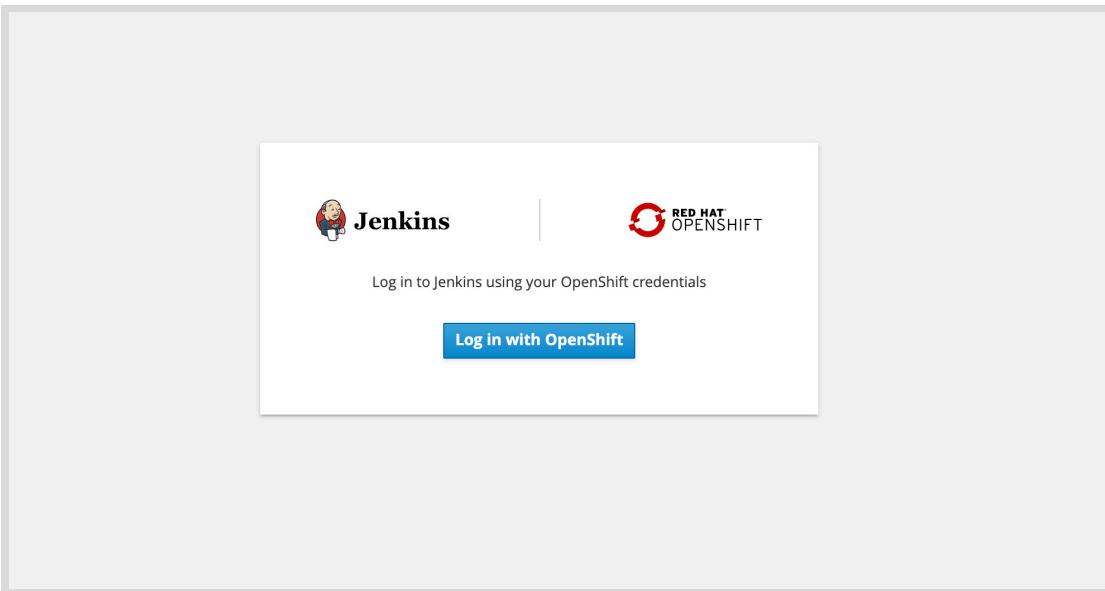
```
[student@workstation ~]$ oc get pods
NAME           READY   STATUS    RESTARTS   AGE
jenkins-1-deploy  0/1     Completed   0          9m42s
jenkins-1-q5msj  1/1     Running    0          9m34s
```

- 5.2. Get the route URL for the Jenkins instance:

```
[student@workstation ~]$ oc get route/jenkins -o jsonpath='{.spec.host}{"\n"}'  
jenkins-youruser-ci-cd.apps.cluster.domain.example.com
```

- 5.3. Navigate to the route URL from the previous step using a web browser.

You will be presented with a screen to log in using your OpenShift credentials.



Click **Log in with OpenShift** to bring up the OpenShift web console login page.

- 5.4. Log in using your developer user account. Your user name is the **RHT_OCP4_DEV_USER** variable in the **/usr/local/etc/ocp4.config** classroom configuration file. Your password is the **RHT_OCP4_DEV_PASSWORD** variable in the same file.
- 5.5. You will be shown a screen that asks you to authorize service account access to your account.

Authorize Access

Service account jenkins in project yourname-ci-cd is requesting permission to access your account (yourname)

Requested permissions

user:info

Read-only access to your user information (including username, identities, and group membership)

user:check-access

Read-only access to view your privileges (for example, "can I create builds?")

You will be redirected to <https://jenkins-yourname-ci-cd.apps.cluster.domain.example.com/securityRealm/finishLogin>

Allow selected permissions Deny

Click **Allow selected permissions** to bring up the Jenkins home page.

The screenshot shows the Jenkins home page. At the top, there's a navigation bar with a Jenkins logo and the word "Jenkins". Below it is a sidebar with icons and links: "New Item", "People", "Build History", "Manage Jenkins", "My Views", "Open Blue Ocean", "Lockable Resources", "Credentials", and "New View". The main content area has a large "Welcome to Jenkins!" message with a call to action: "Please [create new jobs](#) to get started." Below this, there are two sections: "Build Queue" (which says "No builds in the queue.") and "Build Executor Status" (which lists 1, 2, 3, 4, and 5 idle executors).

- 5.6. Click **Manage Jenkins** in the left menu of the Jenkins home page, and then click on **Configure System** in the **Manage Jenkins** page to bring up the Jenkins global configuration page.
- 5.7. Scroll down to the **OpenShift Jenkins Sync** section, and add the namespace ***youruser-simple-pipeline*** to the **Namespace** field, next to the already existing ***youruser-ci-cd*** namespace. Separate the entries by a space, and do not use a comma.
Do not change any other parameter on this page. You will create the ***youruser-simple-pipeline*** project in the next step.
Click **Save** to apply the changes to the Jenkins global configuration.

OpenShift Jenkins Sync

Enabled	<input checked="" type="checkbox"/>	Check if we should sync OpenShift builds with Jenkins jobs
Server	<input type="text"/>	
Credentials	- none -	<input type="button" value="Add"/>
Namespace	youruser-ci-cd youruser-simple-pipeline	
Folders	<input checked="" type="checkbox"/> Check if we should create folders for each OpenShift namespace	
Sync Job Name Pattern	<input type="text"/> The regular expression to match pipeline job names which should be sync'd to BuildConfigs in OpenShift sync Jenkins Jobs to OpenShift BuildConfigs.	

▶ 6. Deploy the build configuration to your project, and start a new build.

- 6.1. Create a new project for running the pipeline. Prefix the project name with your developer user name.

```
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-simple-pipeline
...output omitted...
```

Ensure that the project name matches the namespace you added to the **OpenShift Jenkins Sync** section in the previous step.

- 6.2. The service account associated with the Jenkins deployment must have the **edit** role for each project monitored by Jenkins. Add the **edit** role to the service account associated with the Jenkins deployment for the **youruser-simple-pipeline** project.

```
[student@workstation ~]$ oc policy add-role-to-user \
> edit system:serviceaccount:${RHT_OCP4_DEV_USER}-ci-cd:jenkins \
> -n ${RHT_OCP4_DEV_USER}-simple-pipeline
clusterrole.rbac.authorization.k8s.io/edit added:
"system:serviceaccount:youruser-ci-cd:jenkins"
```

- 6.3. Create the build configuration:

```
[student@workstation ~]$ oc create \
> -f ~/D0288/labs/simple-pipeline/simple-pipeline.yaml
buildconfig.build.openshift.io/simple-pipeline created
```

- 6.4. Open a web browser and navigate the OpenShift web console.

Find the host name of your OpenShift web console inspecting the routes on the **openshift-console** project.

```
[student@workstation ~]$ oc get route console -n openshift-console \
> -o jsonpath='{.spec.host}{"\n"}'
console-openshift-console.apps.cluster.domain.example.com
```

Log in using your developer user account.

6.5. Select the **youruser-simple-pipeline** project.

In the left navigation bar, click **Builds** → **Build Configs**, and verify that a build configuration named **simple-pipeline** is visible.

6.6. Start a new build from the command line to initiate the pipeline:

```
[student@workstation ~]$ oc start-build simple-pipeline  
build.build.openshift.io/simple-pipeline-1 started
```

You can also use the OpenShift web console to start a build. Click the **simple-pipeline** build configuration, and then click **Actions** → **Start Build** in the top right corner to start a build.

► 7. View pipeline execution progress in the OpenShift web console.



Note

The Jenkins pipeline build strategy is deprecated in OpenShift Container Platform 4. The equivalent functionality is present in the OpenShift Pipelines based on Tekton.

7.1. Once the build is started, view the progress of the pipeline.

In the left navigation bar, click **Builds** → **Builds**, and verify that a build named **simple-pipeline-1** is visible.

7.2. Click **simple-pipeline-1** to view the build details page.

After some time, you should see the pipeline stages executing. Execution is paused at the **manual approval** stage.

Builds > Build Details

B simple-pipeline-1 Running Actions ▾

Details YAML Environment Logs Events

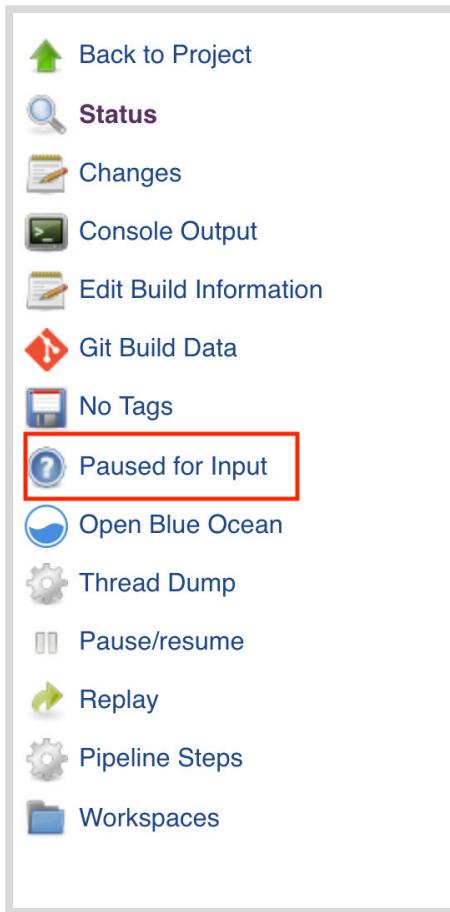
Pipeline build strategy deprecation
With the release of [OpenShift Pipelines based on Tekton](#), the pipelines build strategy has been deprecated. Users should either use Jenkins files directly on Jenkins or use cloud-native CI/CD with Openshift Pipelines.
[Try the OpenShift Pipelines tutorial](#)

Build Details

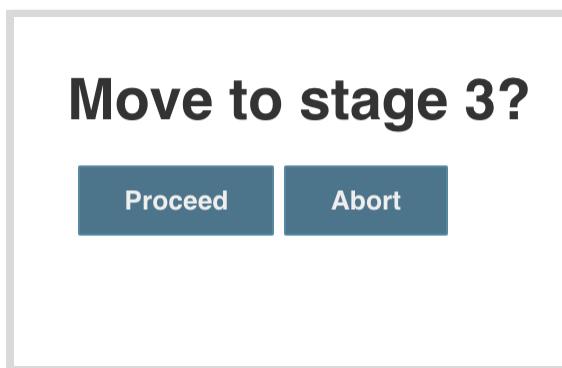
Build 1 3 minutes ago View logs	Declarative:... 2 minutes ago	→	stage 1 2 minutes ago	→	stage 2 2 minutes ago	→	manual app... Input Required
--	----------------------------------	---	--------------------------	---	--------------------------	---	---------------------------------

- 7.3. Click **Input Required** below the yellow pause icon to open the Jenkins build details page in a new browser tab.

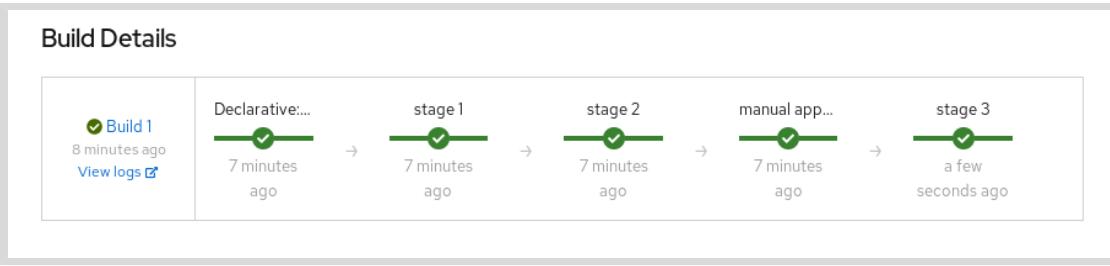
Click **Paused for Input** in the Jenkins left menu near the blue question mark icon.



- 7.4. Click **Proceed** in the **Move to stage 3?** prompt.



- 7.5. The pipeline execution will now continue to **stage 3** and the pipeline will complete successfully.



- ▶ 8. View the details of the executed pipeline in the Jenkins web console.
 - 8.1. Click the **Jenkins** icon in the top left corner of the Jenkins console to view the home page.
A new folder named **youruser-simple-pipeline** is now listed. All pipeline execution logs for this project are stored inside this folder. Click **youruser-simple-pipeline** to view the pipelines in this project.
 - 8.2. Click **youruser-simple-pipeline/simple-pipeline** to view the build history for the pipeline. Click #1 (blue ball icon) in the **Build History** section on the left menu.



Note

Clicking on the build number opens the build details, while clicking on the blue ball icon besides it opens the console output page for that build.

Click **Console Output** to view the detailed execution log for the pipeline. You will see detailed execution information about each stage in the pipeline. The output is as follows:

```
...output omitted...
OpenShift Build youruser-simple-pipeline/simple-pipeline-1 from https://github.com/yourgituser/D0288-apps
...output omitted...
[Pipeline] Start of Pipeline
[Pipeline] node
...output omitted...
stage 1: using project: youruser-ci-cd in cluster https://172.30.0.1:443
...output omitted...
[Pipeline] sh
[workspace] Running shell script
+ echo hello from stage '2!'
...output omitted...
[Pipeline] input
Move to stage 3?
Proceed or Abort
Approved by youruser
...output omitted...
[Pipeline] { (stage 3)
[Pipeline] sh
[workspace] Running shell script
+ echo hello from stage '3!.' This is the last stage...
hello from stage 3!. This is the last stage...
```

```
...output omitted...
[Pipeline] End of Pipeline
Finished: SUCCESS
```

- 9. Cleanup: Delete the **youruser-simple-pipeline** project.

```
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-simple-pipeline
```



Warning

Do NOT delete the **youruser-ci-cd** project. You will reuse the Jenkins instance in the next exercise.

Finish

On **workstation**, run the **lab simple-pipeline finish** command to complete this exercise. This is an important step to ensure that resources from previous exercises do not impact upcoming exercises. The finish action releases this project and its resources.

```
[student@workstation ~]$ lab simple-pipeline finish
```

This concludes the guided exercise.

Writing Custom Jenkins Pipelines

Objectives

After completing this section, you should be able to create and run a custom Jenkins pipeline.

Creating a Custom Jenkins Pipeline

You can create a custom Jenkins pipeline for your application by writing scripts in the OpenShift Pipeline Domain Specific Language (DSL) in a Jenkinsfile. The OpenShift Pipeline DSL aims to provide a readable, concise, comprehensive, and fluent Jenkins Pipeline syntax interacting with an OpenShift cluster. The plug-in provides wrapper DSL functions for the OpenShift command line tool (**oc**) which must be available on the nodes executing the script.

Elements of a Jenkinsfile

The following section describes some of the most important and frequently used elements of a Jenkinsfile.

pipeline

A Jenkinsfile begins with the pipeline element declared as follows:

```
pipeline {  
    ...output omitted...  
}
```

options

You can optionally declare global options that apply to the entire pipeline in the **options** element:

```
pipeline {  
    options {  
        // set a timeout of 45 minutes for this pipeline  
        timeout(time: 45, unit: 'MINUTES')  
    }  
    ...output omitted...  
}
```

The options can be overridden on a per stage basis.

agent

The **agent** element specifies the execution context (where the pipeline will run) for the entire pipeline. You can optionally override the agent at a per stage level. Use a *label* element within the agent element to indicate image streams with specific labels that should be used to run the pipeline.

```
pipeline {
    options {
        ...output omitted...
    }
    agent {
        node {
            label 'master'
        }
    }
}
```

A node with the **master** label is available by default in Jenkins. It contains a minimal Linux runtime, and can be used to run basic pipelines using shell scripts.

OpenShift provides an Apache Maven and OpenJDK based runtime for Java applications with the **maven** label. OpenShift also comes with a Node.js and NPM runtime with the label **nodejs**.

You can create your own custom container images with appropriate labels and have Jenkins use it for running the pipeline. Creating custom containers is out of scope for this course. Look at the resources provided in the references section for more details.

environment

Use the **environment** element to add variables and other constants that will be used throughout the pipeline. This element is intended to capture variables in a single place to avoid duplicating values like URLs, project names, application names, and other variables used in the pipeline code.

```
environment {
    DEV_PROJECT = "myapp-dev"
    STAGE_PROJECT = "myapp-stage"
    APP_GIT_URL = "https://mygitserver/myapp"
    NEXUS_SERVER = "http://mynexusserver/repository/java"
    ...output omitted...
}
```

stages and steps

A pipeline consists of one or more **stages**. Each stage represents a unique set of tasks that you want to execute in the pipeline, such as smoke and unit tests. A stage consists of a number of **steps** declared within the **steps** element:

```
pipeline {
    options {
        ...output omitted...
    }
    ...output omitted...

    stages {
        stage('stage 1') {
            steps {
                ...output omitted...
            }
        }
    }
}
```

```

        stage('stage 2') {
            steps {
                ...output omitted...
            }
        }

        stage('stage 3') {
            steps {
                ...output omitted...
            }
        }

    }
}

```

script

Pipeline DSL code (Groovy language code) must be embedded within `script` elements. OpenShift specific functions are provided in the `openshift.*` namespace and provide wrapper functions around the `oc` command line client.

An example script that prints the current OpenShift namespace and cluster follows:

```

...output omitted...
stage('stage 1') {
    steps {
        script {
            openshift.withCluster() {
                openshift.withProject() {
                    echo "stage 1: using project:
${openshift.project()} in cluster ${openshift.cluster()}"
                }
            }
        }
    }
...output omitted...

```

A more complex example that starts a new build is as follows:

```

...output omitted...
script {
    openshift.withCluster() {
        openshift.withProject(env.DEV_PROJECT) {
            openshift.selector("bc", "${APP_NAME}").startBuild("--wait=true", "--follow=true")
        }
    }
...output omitted...

```

sh (shell scripting)

Apart from scripting pipelines using the Groovy based pipeline DSL within **script** elements, you can embed shell scripts to manipulate the pipeline runtime. The **oc** command line tool is available at runtime when the pipeline is running.

A simple one line shell script can be embedded as follows:

```
...output omitted...
stage('stage A') {
    steps {
        sh 'echo Hello World!'
    }
}
```

A multiline shell script can be embedded as follows:

```
...output omitted...
stage('stage A') {
    steps {
        sh'''
            echo 'deleting all resources...'
            oc project ${DEV_PROJECT}
            oc delete all -l app=${APP_NAME}
            sleep 5
        '''
    }
}
```



Warning

When scripting pipelines using DSL and shell scripts, make sure you are in the correct project in each stage before executing commands. The current project context switches to the default project between stages.

When using shell scripts, it is always a good idea to use the **-n** option to explicitly define the project when running **oc** commands. Similarly, for DSL code, make effective use of the **openshift.withCluster()**, and **openshift.withProject()** functions before manipulating resources in OpenShift.

input

Use the *input* element to add manual interaction to the pipeline. This is most often used for scenarios where you are promoting the application between environments, and you need a manual approval for the pipeline to continue.

For example, to ensure that promotions from development to staging always require manual approval, use the following:

```
...output omitted...
stage('Promote to Staging Env') {
    steps {
```

```
        timeout(time: 60, unit: 'MINUTES') {  
            input message: "Promote to Staging?"  
        }  
        script {  
            // code to promote app  
        }  
    }  
}  
...output omitted...
```

The `timeout` element ensures that if manual approval is not given after 60 minutes, the pipeline execution continues.



References

Jenkins Client Plugin Pipeline DSL

<https://github.com/openshift/jenkins-client-plugin>

Jenkins Pipeline DSL syntax

<https://jenkins.io/doc/book/pipeline/syntax/>

► Guided Exercise

Create and Run a Jenkins Pipeline

In this exercise, you will create and run a Jenkins pipeline for deploying a Node.js microservice.

Outcomes

You should be able to create and run a multi stage Jenkins pipeline on OpenShift for deploying a Node.js microservice.

Before You Begin



Important

You must complete Steps 1.1 to 1.5 from the previous exercise (*Run a Simple Jenkins Pipeline*) before attempting this exercise.

To perform this exercise, you must have access to:

- A running OpenShift cluster.
- A Jenkins instance deployed and running in the **youruser-ci-cd** namespace.
- The sample application in the Git repository (**books**).

Run the following command on **workstation** to validate the prerequisites and download solution files:

```
[student@workstation ~]$ lab custom-pipeline start
```

- 1. Verify that the Jenkins instance you deployed in the previous exercise is ready and running.

1. Load your classroom environment configuration.

Run the following command to load the environment variables created in the first guided exercise:

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

- 1.2. Log in to OpenShift using your developer user account.

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful.
...output omitted...
```

- 1.3. Verify that the Jenkins pod is running:

```
[student@workstation ~]$ oc get pods -n ${RHT_OCP4_DEV_USER}-ci-cd
NAME          READY   STATUS    RESTARTS   AGE
...output omitted...
jenkins-1-q5msj   1/1     Running   0          9m34s
```

- ▶ 2. Create two projects for the development and staging environments for your microservice. The pipeline will first deploy the books microservice to the development environment, and then promote it to the staging environment after running unit tests and linting tools on the code.

- 2.1. Create a new project for the staging environment. Prefix the project name with your developer user name:

```
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-books-stage
Now using project "youruser-books-stage" on server
"https://api.cluster.domain.example.com:6443".
...output omitted...
```

- 2.2. Create a new project for the development environment. Prefix the project name with your developer user name:

```
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-books-dev
Now using project "youruser-books-dev" on server
"https://api.cluster.domain.example.com:6443".
...output omitted...
```

- ▶ 3. Edit the build configuration for the pipeline.

- 3.1. Inspect the pipeline build configuration resource file at **~/D0288/labs/custom-pipeline/custom-pipeline-bc.json** using a text editor:

```
{
  "kind": "BuildConfig",
  "apiVersion": "build.openshift.io/v1",
  "metadata": {
    "name": "custom-pipeline" 1
  },
  "spec": {
    "source": {
      "type": "Git",
      "git": {
        "uri": "https://github.com/youruser/D0288-apps.git", 2
        "ref": "custom-pipeline" 3
      },
      "contextDir": "books" 4
    },
    "strategy": {
      "type": "JenkinsPipeline",
      "jenkinsPipelineStrategy": {
        "jenkinsfilePath": "jenkins/Jenkinsfile" 5
      }
    }
}
```

```

    }
}
}
```

- ➊ The name for the build configuration.
- ➋ The URL of the Git repository where the Jenkinsfile is stored.
- ➌ The branch name (in the Git repository) to be used for the build.
- ➍ The directory name under which the application source is kept.
- ➎ The directory name relative to the **contextDir** where the Jenkinsfile is kept.

**Note**

In this exercise, we store the application source for the books application, and the Jenkinsfile in the same Git repository. It is not mandatory to do this, however it is a recommended practice.

- 3.2. Edit the Git URL where the application source is stored.

Edit the build configuration and change the value of the **spec.source.git.uri** attribute to point to your Git repository:

```

...output omitted...
git: {
  uri: "https://github.com/youruser/D0288-apps",
  ref: "custom-pipeline"
},
...output omitted...
```

▶ **4.** Inspect the Jenkins pipeline.

- 4.1. Enter your local clone of the **D0288-apps** Git repository, and then checkout the **master** branch of the course repository to ensure you start this exercise from a known good state:

```

[student@workstation ~]$ cd D0288-apps
[student@workstation D0288-apps]$ git checkout master
...output omitted...
```

- 4.2. Create a new branch to save any changes you make during this exercise:

```

[student@workstation D0288-apps]$ git checkout -b custom-pipeline
Switched to a new branch 'custom-pipeline'
[student@workstation D0288-apps]$ git push -u origin custom-pipeline
...output omitted...
* [new branch]      custom-pipeline -> custom-pipeline
Branch custom-pipeline set up to track remote branch custom-pipeline from origin.
```

- 4.3. Inspect the Jenkinsfile at **~/D0288-apps/books/jenkins/Jenkinsfile**. The Jenkinsfile defines a multistage pipeline for deploying the microservice. Some of the stages are complete, while others are incomplete.

In the subsequent steps, review and complete the pipeline by implementing the lines marked as **TODO**: in the file.

► 5. Review the pipeline stages, and complete the Jenkins pipeline.

- 5.1. Edit the `~/DO288-apps/books/jenkins/Jenkinsfile` file and perform the following steps. You can also copy the completed stages from the provided `~/DO288/solutions/custom-pipeline/Jenkinsfile` file.
- 5.2. The pipeline should run on a Jenkins agent node that supports Node.js. Add a label to the `node` section to enable the pipeline to run on a node labeled `nodejs`. Locate the line:

```
//TODO: Add label for Node.js jenkins agent
```

Replace it with the following:

```
...output omitted...
agent {
  node {
    label 'nodejs'
  }
}
...output omitted...
```

- 5.3. Customize the `environment` section and add values as per your environment.

Replace the value of `youruser` with your developer user account. Replace the value of the `NEXUS_SERVER` attribute with the value of the `RHT_OCP4_NEXUS_SERVER` variable from the `/usr/local/etc/ocp4.config` file.

```
...output omitted...
environment {
  //TODO: Edit these vars as per your env
  DEV_PROJECT = "youruser-books-dev"
  STAGE_PROJECT = "youruser-books-stage"
  APP_GIT_URL = "https://github.com/youruser/DO288-apps"
  NEXUS_SERVER = "http://nexus-common.apps.cluster.domain.example.com/repository/
nodejs"

  // DO NOT CHANGE THE GLOBAL VARS BELOW THIS LINE
  APP_NAME = "books"
}
...output omitted...
```

- 5.4. The **NPM Install, Run Unit Tests, and Run Linting Tools** stages install the dependencies for the microservice, runs unit tests, and lints the code respectively.

These stages are designed to detect errors early in the development phase, and provides rapid feedback to the developers following a "fail fast" approach. Do not make any changes to these stages.

- 5.5. Edit the **Launch new app in DEV env** stage, and launch a new OpenShift application in the development environment.

Locate the line:

```
//TODO: Create a new app and expose the service
```

Replace it with the following lines:

```
...output omitted...
sh ''
  oc project ${DEV_PROJECT}
  oc new-app --as-deployment-config --name books nodejs:12~${APP_GIT_URL} \
  --build-env npm_config_registry=${NEXUS_SERVER} \
  --context-dir ${APP_NAME}

  oc expose svc/${APP_NAME}
  ...
...output omitted...
```

- 5.6. The **Wait for S2I build to complete** and **Wait for deployment in DEV env** stages wait for the build and deployment of the microservice to complete successfully.

Note the use of the OpenShift Pipeline DSL Plugin in these stages, instead of raw shell script commands, to interact with the OpenShift master API. Do not make any changes to these stages.

- 5.7. Edit the **Promote to Staging Env** stage, and tag the latest image stream of the microservice.

Locate the line:

```
//TODO: Tag the books:latest image stream as books:stage
```

Replace it with the following lines:

```
...output omitted...
script {
  openshift.withCluster() {
    openshift.tag("${DEV_PROJECT}/books:latest", "${STAGE_PROJECT}/books:stage")
  }
}...output omitted...
```

- 5.8. Edit the **Deploy to Staging Env** stage, and add code to create a new application using the tagged image stream from the previous stage.

Locate the line:

```
//TODO: Create a new app in stage using the books:stage image stream and expose
the service
```

Replace it with the following lines:

```
...output omitted...
sh '''
  oc project ${STAGE_PROJECT}
  oc new-app --as-deployment-config --name books -i books:stage
  oc expose svc/${APP_NAME}
'''
...output omitted...
```

- 5.9. Edit the **Wait for deployment in Staging** stage, and add code to verify that the application pods are running. Use the code in the **Wait for deployment in DEV env** stage as a reference to complete this stage.

Locate the line:

```
//TODO: Watch deployment until pod is in 'Running' state
```

Replace it with the following lines:

```
...output omitted...
openshift.withProject( "${STAGE_PROJECT}" ) {
  def deployment = openshift.selector("dc", "${APP_NAME}").rollout()
  openshift.selector("dc", "${APP_NAME}").related('pods').untilEach(1) {
    return (it.object().status.phase == "Running")
  }
}
...output omitted...
```

The Jenkinsfile is now complete. Save your changes.

- 5.10. Commit the changes to the Git repository from the `~/D0288-apps/books` folder:

```
[student@workstation D0288-apps]$ cd ~/D0288-apps/books
[student@workstation books]$ git commit -a \
> -m "Completed Jenkinsfile for books microservice"
...output omitted...
[student@workstation books]$ git push
...output omitted...
[student@workstation books]$ cd ~
```

- 6. Configure the Jenkins Sync Plugin, and allow Jenkins to monitor your project for pipelines.

- 6.1. Get the route URL for the Jenkins instance:

```
[student@workstation ~]$ oc get route/jenkins -n ${RHT_OCP4_DEV_USER}-ci-cd \
> -o jsonpath='{.spec.host}{"\n"}'
jenkins-youruser-ci-cd.apps.cluster.domain.example.com
```

- 6.2. Navigate to the route URL from the previous step using a web browser.

You will be presented with a screen to log in using your OpenShift credentials. Click **Log in with OpenShift** to bring up the OpenShift web console log in page.

- 6.3. Log in using your developer user account. Your user name is the **RHT_OCP4_DEV_USER** variable in the **/usr/local/etc/ocp4.config** classroom

configuration file. Your password is the **RHT_OCP4_DEV_PASSWORD** variable in the same file.

- 6.4. If you are accessing the Jenkins console for the first time, a screen that asks you to authorize service account access to your account is shown. Click **Allow selected permissions** to bring up the Jenkins home page.
- 6.5. Click **Manage Jenkins** on the left menu of the Jenkins home page, and then click **Configure System** on the **Manage Jenkins** page to bring up the Jenkins global configuration page.
- 6.6. Scroll down to the **OpenShift Jenkins Sync** section, and add the namespace **youruser-books-dev** to the **Namespace** field, next to the already existing **youruser-ci-cd** and **youruser-simple-pipeline** namespaces. Separate the entries by a space and do not use a comma.
Do not change any other parameter on this page. Click **Save** to apply the changes to the Jenkins global configuration.

The screenshot shows the Jenkins global configuration interface. In the 'OpenShift Jenkins Sync' section, there is a 'Namespace' input field which contains the text 'youruser-ci-cd youruser-simple-pipeline youruser-books-dev'. This field is highlighted with a blue border, indicating it is the current focus or has been modified. Other fields in the section include 'Enabled' (checked), 'Server' (empty), 'Credentials' (dropdown set to 'none'), and 'Folders' (checked).

► 7. Deploy the build configuration to your project, and start a new build.

- 7.1. Ensure that you create the build configuration in the **youruser-books-dev** namespace.

```
[student@workstation ~]$ oc project ${RHT_OCP4_DEV_USER}-books-dev
Now using project "youruser-books-dev" on server
...output omitted...
```

- 7.2. The service account associated with the Jenkins deployment must have the **edit** role for each project where Jenkins performs some operation. Add the **edit** role to the service account associated with the Jenkins deployment for the **youruser-books-dev** and **youruser-books-stage** projects.

```
[student@workstation ~]$ oc policy add-role-to-user \
> edit system:serviceaccount:${RHT_OCP4_DEV_USER}-ci-cd:jenkins \
> -n ${RHT_OCP4_DEV_USER}-books-dev
clusterrole.rbac.authorization.k8s.io/edit added:
"system:serviceaccount:youruser-ci-cd:jenkins"

[student@workstation ~]$ oc policy add-role-to-user \
> edit system:serviceaccount:${RHT_OCP4_DEV_USER}-ci-cd:jenkins \
```

```
> -n ${RHT_OCP4_DEV_USER}-books-stage
clusterrole.rbac.authorization.k8s.io/edit added:
"system:serviceaccount:youruser-ci-cd:jenkins"
```

- 7.3. Create the build configuration:

```
[student@workstation ~]$ oc create \
> -f ~/D0288/labs/custom-pipeline/custom-pipeline-bc.json
buildconfig.build.openshift.io/custom-pipeline created
```

- 7.4. Open a web browser and navigate to the OpenShift web console.

Find the host name of your OpenShift web console inspecting the routes on the **openshift-console** project.

```
[student@workstation ~]$ oc get route console -n openshift-console \
> -o jsonpath='{.spec.host}{"\n"}'
console-openshift-console.apps.cluster.domain.example.com
```

Log in using your developer user account.



Note

The Jenkins pipeline build strategy is deprecated in OpenShift Container Platform 4. The equivalent functionality is present in the OpenShift Pipelines based on Tekton.

- 7.5. Select the **youruser-books-dev** project.

In the left navigation bar, click **Builds** → **Build Configs**, and verify that a build configuration named **custom-pipeline** is visible.

- 7.6. Start a new build from the command line to initiate the pipeline:

```
[student@workstation ~]$ oc start-build custom-pipeline
build.build.openshift.io/custom-pipeline-1 started
```

You can also use the OpenShift web console to start a build. Click the **custom-pipeline** build configuration, and then click **Actions** → **Start Build** in the top right corner to start a build.

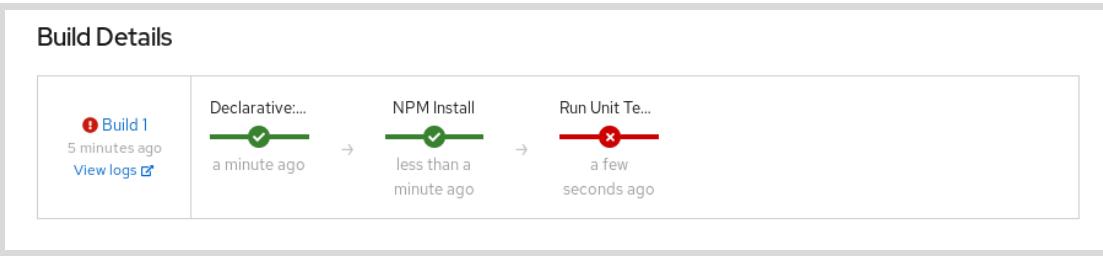
- 8. View the progress of the pipeline.

- 8.1. Once the build is started, view the progress of the pipeline.

In the left navigation bar, click **Builds** → **Builds**, and verify that a build named **custom-pipeline-1** is visible.

- 8.2. Click **custom-pipeline-1** to view the build details page.

After some time, you should see the pipeline stages executing. Execution fails in the **Run Unit Tests** stage.



▶ 9. Fix the failing unit tests and run the pipeline again.

- 9.1. On the **Build Details** page, click **View Logs** (below **Build 1** with exclamation sign in a red bubble) to view the Jenkins pipeline execution log.
- 9.2. The Jenkins pipeline execution log shows an error in the **Run Unit Tests** stage:

```
...output omitted...
+ cd books
+ npm test

> books@1.0.0 test /tmp/workspace/ ...output omitted...
> IP=0.0.0.0 PORT=3030 node_modules/.bin/mocha tests/*_test.js
...output omitted...
2 passing (138ms)
1 failing

1) Books App routes test
   GET to /authors should return 200:
     Uncaught AssertionError: expected '{"books": [{"id":1,"name":"James Joyce","dob":1882}, {"id":2,"name":"F Scott Fitzgerald","dob":1896}, {"id":3,"name":"Aldous Huxley","dob":1894}, {"id":4,"name":"Vladimir Nabokov","dob":1899}, {"id":5,"name":"William Faulkner","dob":1897}]}' to include 'James_Joyce'
...output omitted...
```

The HTTP GET call to the `/authors` endpoint returns an array of author objects. The test ascertains if the string "`James_Joyce`" is present in the output. The test failed because the response includes the string "`James Joyce`".

- 9.3. Fix the failing unit test. Open the `~/DO288-apps/books/tests/app_test.js` in a text editor.
- 9.4. Locate the following line in the '`GET to /authors should return 200`' test case at the bottom:

```
expect(res.text).to.include('James_Joyce');
```

Replace it with the following correct assertion:

```
expect(res.text).to.include('James Joyce');
```

- 9.5. Commit your changes and push the fixed test file to your `youruser-custom-pipeline` branch.

```
[student@workstation ~]$ cd ~/DO288-apps/books
[student@workstation books]$ git commit -a -m "Fixed failed unit test."
...output omitted...
[student@workstation books]$ git push
...output omitted...
[student@workstation books]$ cd ~
```

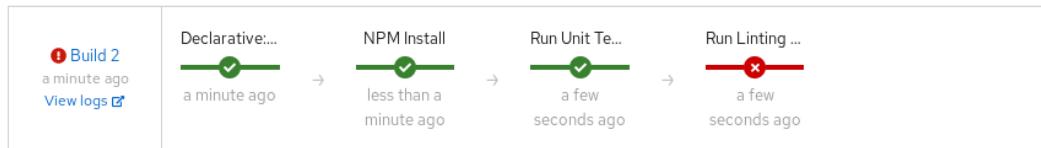
- 9.6. Start a new build to run the pipeline again.

```
[student@workstation ~]$ oc start-build custom-pipeline
build.build.openshift.io/custom-pipeline-2 started
```

- 9.7. Monitor the progress of the pipeline execution for the build **custom-pipeline-2** from the **Build Details** page.

The pipeline now fails in the **Run Linting Tools** stage.

Build Details



Check the Jenkins pipeline execution log by clicking **View Logs**. The logs shows that the linting tool is complaining about unused variables in the code, as well as usage of the **var** keyword, which is not recommended in the latest versions of JavaScript.

```
...output omitted...
+ cd books
+ npm run lint

> books@1.0.0 lint /tmp/workspace/ ...output omitted...
> eslint . --ext .js

...output omitted.../books/routes/authors.js
 7:1  error  Unexpected var, use let or const instead  no-var
 7:5  error  'user' is defined but never used          no-unused-vars
...output omitted...
```

- ▶ 10. Fix the linting errors and run the pipeline again.

- 10.1. Open the **~/DO288-apps/books/routes/authors.js** in a text editor.

- 10.2. Delete the following line of code:

```
var user;
```

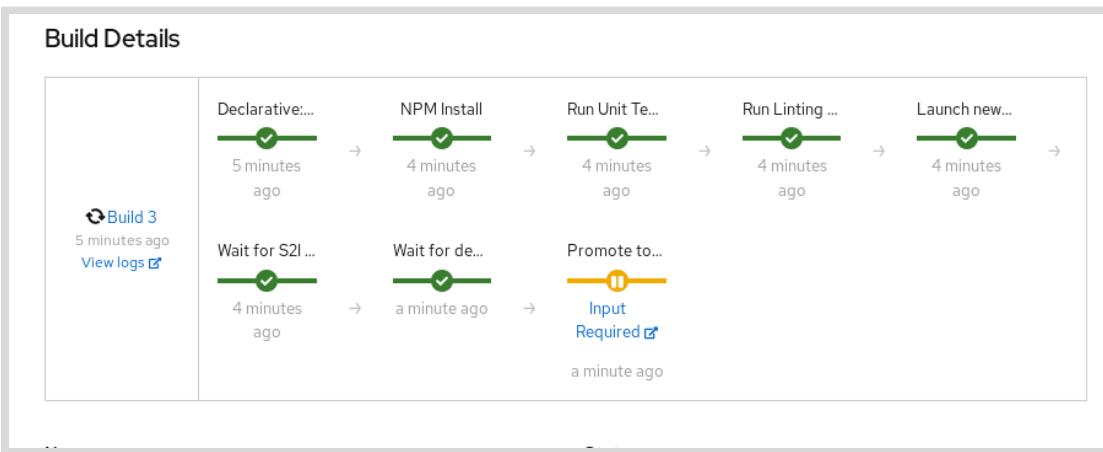
- 10.3. Commit your changes and push the fixed test file to your **youruser-custom-pipeline** branch.

```
[student@workstation ~]$ cd ~/DO288-apps/books
[student@workstation books]$ git commit -a -m "Fixed linting errors."
...output omitted...
[student@workstation books]$ git push
...output omitted...
[student@workstation books]$ cd ~
```

10.4. Start a new build to run the pipeline again.

```
[student@workstation ~]$ oc start-build custom-pipeline
build.build.openshift.io/custom-pipeline-3 started
```

Monitor the progress of the pipeline execution for the build **custom-pipeline-3** from the **Build Details** page. The pipeline should now continue with building and launching the microservice in the development environment. After these stages are complete, pipeline execution is paused at the **Promote to Staging Env** stage.



► 11. Verify that the microservice is deployed and running in the development environment.

11.1. Before promoting the microservice to the staging environment, verify that the application is running in the development environment.

Verify that the books microservice is running:

```
[student@workstation ~]$ oc get pods -n ${RHT_OCP4_DEV_USER}-books-dev
NAME      READY   STATUS    RESTARTS   AGE
...output omitted...
books-1-b6h94   1/1     Running      0          17m
```

11.2. Verify that the staging environment does not have any pods running:

```
[student@workstation ~]$ oc get pods -n ${RHT_OCP4_DEV_USER}-books-stage
No resources found.
```

11.3. Get the route URL for the books microservice:

```
[student@workstation ~]$ oc get route/books \
> -n ${RHT_OCP4_DEV_USER}-books-dev \
> -o jsonpath='{.spec.host}{"\n"}'
books-youruser-books-dev.apps.cluster.domain.example.com
```

- 11.4. Navigate to the route URL from the previous step using a web browser and verify that the books microservice is running.

The Book List

Welcome to The Book List

The List of books is [here](#).

The List of authors is [here](#).

- 12. Promote the microservice to the staging environment.

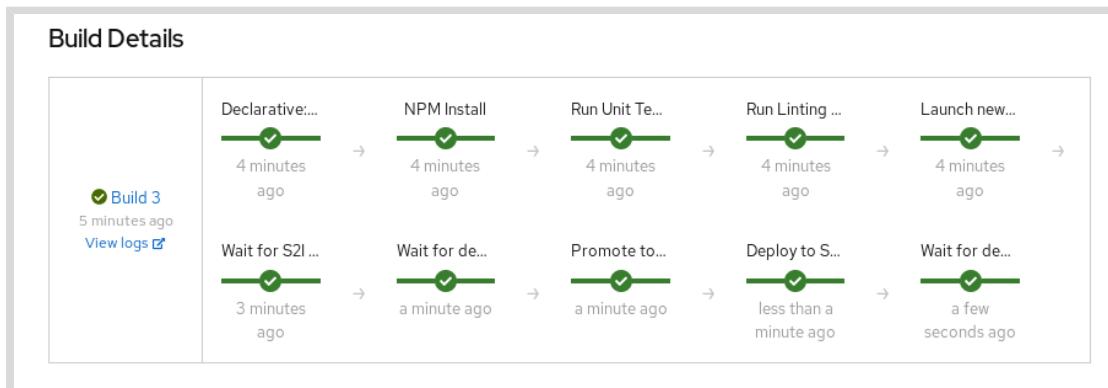
- 12.1. Once you are satisfied that the microservice is working as intended, promote the microservice to the staging environment.
- 12.2. From the **Build Details** page for the build **custom-pipeline-3**, click **Input Required** to view the Jenkins pipeline execution log.
- 12.3. From the Jenkins console left menu, click **Paused for Input**. Click **Proceed** in the **Promote to Staging?** prompt.
- 12.4. Jenkins switches to the execution log window and continues with executing the **Deploy to Staging Env** stage.

- 13. Verify that the pipeline execution is successful. Verify that the microservice is deployed and running in the staging environment.

- 13.1. Pipeline execution should complete and you should see the following in the execution log:

```
...output omitted...
Deployment to Staging env is complete.
Access the app at the URL
http://books-youruser-books-stage.apps.cluster.domain.example.com
...output omitted...
[Pipeline] End of Pipeline
Finished: SUCCESS
```

- 13.2. Click the URL for the books microservice in the staging environment. Verify that the microservice is functionally similar to the version deployed in the development environment.
- 13.3. The **Build Details** page for **custom-pipeline-3** shows the complete pipeline execution status.



- 14. Cleanup: Delete the **youruser-ci-cd**, **youruser-books-dev**, and **youruser-books-stage** projects.

```
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-books-dev
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-books-stage
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-ci-cd
```

Finish

On **workstation**, run the **lab custom-pipeline finish** command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises. The finish action releases this project and its resources.

```
[student@workstation ~]$ lab custom-pipeline finish
```

This concludes the guided exercise.

► Lab

Implementing Continuous Integration and Continuous Deployment Pipelines in OpenShift

Performance Checklist

In this lab, you will deploy Jenkins, and create a CI/CD pipeline for deploying a Java microservice on OpenShift.

Outcomes

You should be able to:

- Deploy a Jenkins ephemeral instance on OpenShift.
- Create a Jenkinsfile with multiple stages to deploy the microservice in the development environment.
- Promote the microservice from the development environment to the staging environment.

Before You Begin

To perform this exercise, ensure you have access to:

- A running OpenShift cluster.
- The sample application (movies) in the Git repository.
- The OpenJDK 8 S2I builder image required by the application.

Run the following command on the **workstation** VM to validate the prerequisites and to download the starter project files and solution files:

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

Requirements

The microservice (movies) provides a REST API that lists movies. It is written in Java using the Spring Boot framework.

The microservice is deployed from source code stored in a Git repository.

You will deploy a Jenkins ephemeral instance on OpenShift, and configure it to run pipelines from selected namespaces in OpenShift.

A starter build configuration resource file called **movies-bc.json** is provided in the **~/D0288/labs/review-cicd** folder. You are required to edit this file and customize it for your environment.

A starter Jenkinsfile containing the pipeline stages is provided in the Git repository along with the application source code. You are required to complete the Jenkinsfile and deploy it on OpenShift.

Once you deploy the pipeline and run it, you will see errors in certain stages. Fix these errors by analyzing the Jenkins pipeline execution log, and make the pipeline run all the stages successfully.

Deploy the microservice and the Jenkins pipeline according to the following requirements:

- Deploy a Jenkins ephemeral instance in a namespace called ***youruser-jenkins***. Ensure that the **MEMORY_LIMIT** parameter in the template is set to **2GB** for improved performance.
- The new project for the development environment should be called ***youruser-movies-dev***. The new project for the staging environment should be called ***youruser-movies-stage***.
- Maven dependencies required to build the application are available from:

`http://nexus-common.apps.cluster.domain.example.com/repository/java`

The **settings.xml** file in the root of the application Git repository contains the URL of the Nexus proxy server that contains all the Maven dependencies for building the application. You must edit this file and add the URL of the Nexus server.

- The microservice deployed in the development environment should be available at the following URL:

`http://movies-youruser-movies-dev.apps.cluster.domain.example.com/movies`

- The microservice deployed in the staging environment should be available at the following URL:

`http://movies-youruser-movies-stage.apps.cluster.domain.example.com/movies`

Steps

Follow the below steps to complete the lab:

1. Deploy a Jenkins ephemeral instance in the ***youruser-jenkins*** project.
2. Configure the Jenkins Sync Plugin, and allow Jenkins to monitor the ***youruser-movies-dev*** project for pipelines.
3. Create the OpenShift projects for the development and staging environments.
4. Edit the build configuration resource file and add the Git URL of the Jenkinsfile.
5. Inspect the Jenkins pipeline. The Jenkinsfile defines a multistage pipeline for deploying the microservice. Some of the stages are complete, while others are incomplete.
Enter your local clone of the **DO288-apps** Git repository, and checkout the **master** branch of the course repository to ensure you start this exercise from a known good state. Create a new branch called **review-cicd** to save any changes you make during this exercise.
6. Review and complete the pipeline by implementing the lines marked as **TODO**: in the Jenkinsfile. Edit the `~/DO288-apps/movies/Jenkinsfile` file and perform the following steps.
 - 6.1. Open the `~/DO288-apps/movies/Jenkinsfile` file using a text editor. You can also copy the completed stages from the provided `~/DO288/solutions/review-cicd/Jenkinsfile` file.
 - 6.2. The pipeline should run on a Jenkins agent node that supports Java and Apache Maven.
 - 6.3. Customize the **environment** section and add values as per your environment.

- 6.4. Inspect the **Launch new app in DEV env** stage. Add code to launch a new OpenShift application in the development environment.
- 6.5. Implement the **Wait for deployment in DEV env** stage. Add code to monitor the deployment until the pods are in the **Running** state.
- 6.6. Inspect the **Promote to Staging Env** stage. Add code to tag the **latest** version of the image stream from the development environment, with a tag in the staging environment called **stage**.
- 6.7. Inspect the **Deploy to Staging Env** stage. Add code to create a new application in the staging environment, using the tagged image stream from the previous stage.
The Jenkinsfile is now complete. Save your changes.
7. Edit the Maven proxy configuration file at `~/D0288-apps/movies/settings.xml`, and change the value of the `url` tag to point to the Nexus proxy server for your environment.
8. Commit the changes to the Git repository from the `~/D0288-apps/movies` folder:

```
[student@workstation D0288-apps]$ cd ~/D0288-apps/movies
[student@workstation movies]$ git commit -a \
> -m "Completed Jenkinsfile for movies microservice"
...output omitted...
[student@workstation movies]$ git push
...output omitted...
[student@workstation movies]$ cd ~
```

9. Deploy the build configuration to the **youruser-movies-dev** project, and start a new build. Ensure that you add the appropriate security roles for projects where Jenkins performs some operation.

Once the build is running, monitor the progress of the pipeline using the OpenShift web console. Execution fails in the **Run Unit Tests** stage.



Note

Jenkins Pipelines are deprecated in OpenShift Container Platform 4. The equivalent functionality is present in the OpenShift Pipelines based on Tekton.

10. Investigate the test failures using the Jenkins pipeline execution log. Fix the failing unit tests and run the pipeline again. The pipeline now fails in the **Static Code Analysis** stage.
11. Fix the errors pointed out by the static code analysis tools, and then run the pipeline again.
12. Verify that the microservice is deployed and running in the development environment.
13. Promote the microservice to the staging environment.
14. Verify that the pipeline execution is successful. Verify that the microservice is deployed and running in the staging environment.
15. Grade your work.

Run the following command on **workstation** to verify that all tasks were accomplished:

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

16. Cleanup: Delete the **youruser-jenkins**, **youruser-movies-dev**, and **youruser-movies-stage** projects.

```
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-movies-dev  
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-movies-stage  
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-jenkins
```

Finish

On **workstation**, run the **lab todo-migrate finish** command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

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

This concludes the lab.

► Solution

Implementing Continuous Integration and Continuous Deployment Pipelines in OpenShift

Performance Checklist

In this lab, you will deploy Jenkins, and create a CI/CD pipeline for deploying a Java microservice on OpenShift.

Outcomes

You should be able to:

- Deploy a Jenkins ephemeral instance on OpenShift.
- Create a Jenkinsfile with multiple stages to deploy the microservice in the development environment.
- Promote the microservice from the development environment to the staging environment.

Before You Begin

To perform this exercise, ensure you have access to:

- A running OpenShift cluster.
- The sample application (movies) in the Git repository.
- The OpenJDK 8 S2I builder image required by the application.

Run the following command on the **workstation** VM to validate the prerequisites and to download the starter project files and solution files:

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

Requirements

The microservice (movies) provides a REST API that lists movies. It is written in Java using the Spring Boot framework.

The microservice is deployed from source code stored in a Git repository.

You will deploy a Jenkins ephemeral instance on OpenShift, and configure it to run pipelines from selected namespaces in OpenShift.

A starter build configuration resource file called **movies-bc.json** is provided in the **~/DO288/labs/review-cicd** folder. You are required to edit this file and customize it for your environment.

A starter Jenkinsfile containing the pipeline stages is provided in the Git repository along with the application source code. You are required to complete the Jenkinsfile and deploy it on OpenShift.

Once you deploy the pipeline and run it, you will see errors in certain stages. Fix these errors by analyzing the Jenkins pipeline execution log, and make the pipeline run all the stages successfully.

Deploy the microservice and the Jenkins pipeline according to the following requirements:

- Deploy a Jenkins ephemeral instance in a namespace called ***youruser-jenkins***. Ensure that the **MEMORY_LIMIT** parameter in the template is set to **2GB** for improved performance.
- The new project for the development environment should be called ***youruser-movies-dev***. The new project for the staging environment should be called ***youruser-movies-stage***.
- Maven dependencies required to build the application are available from:

`http://nexus-common.apps.cluster.domain.example.com/repository/java`

The **settings.xml** file in the root of the application Git repository contains the URL of the Nexus proxy server that contains all the Maven dependencies for building the application. You must edit this file and add the URL of the Nexus server.

- The microservice deployed in the development environment should be available at the following URL:

`http://movies-youruser-movies-dev.apps.cluster.domain.example.com/movies`

- The microservice deployed in the staging environment should be available at the following URL:

`http://movies-youruser-movies-stage.apps.cluster.domain.example.com/movies`

Steps

Follow the below steps to complete the lab:

1. Deploy a Jenkins ephemeral instance in the ***youruser-jenkins*** project.

- 1.1. Load the configuration of your classroom environment.

Run the following command to load the environment variables created in the first guided exercise:

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

- 1.2. Log in to OpenShift using your developer user account.

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful.
...output omitted...
```

- 1.3. Create a new project to host the Jenkins instance. Prefix the project name with your developer user name:

```
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-jenkins
Now using project "youruser-jenkins" on server
"https://api.cluster.domain.example.com:6443".
...output omitted...
```

- 1.4. Deploy an instance of Jenkins.

```
[student@workstation ~]$ oc new-app --as-deployment-config \
> jenkins-ephemeral -p MEMORY_LIMIT=2048Mi
--> Deploying template "openshift/jenkins-ephemeral" to project youruser-jenkins
...output omitted...
--> Creating resources ...
...output omitted...
--> Success
...output omitted...
```

- 1.5. The deployment will take some time. Verify that the Jenkins instance is ready and running:

```
[student@workstation ~]$ oc get pods
NAME          READY   STATUS    RESTARTS   AGE
jenkins-1-deploy  0/1     Completed   0          2m42s
jenkins-1-rtnw5   1/1     Running    0          1m34s
```

- 1.6. Get the route URL for the Jenkins instance:

```
[student@workstation ~]$ oc get route/jenkins -o jsonpath='{.spec.host}{"\n"}'
jenkins-youruser-jenkins.apps.cluster.domain.example.com
```

- 1.7. Navigate to the route URL from the previous step using a web browser.

You will be presented with a screen to log in using your OpenShift credentials. Click **Log in with OpenShift** to bring up the OpenShift web console log in page.

- 1.8. Log in using your developer user account. Your user name is the **RHT_OCP4_DEV_USER** variable in the **/usr/local/etc/ocp4.config** classroom configuration file. Your password is the **RHT_OCP4_DEV_PASSWORD** variable in the same file.
- 1.9. A screen displays that asks you to authorize service account access to your account. Click **Allow selected permissions** to bring up the Jenkins home page.
2. Configure the Jenkins Sync Plugin, and allow Jenkins to monitor the **youruser-movies-dev** project for pipelines.
 - 2.1. From the Jenkins home page, click **Manage Jenkins** in the left menu of the Jenkins home page, and then click **Configure System** in the **Manage Jenkins** page to bring up the Jenkins global configuration page.
 - 2.2. Scroll down to the **OpenShift Jenkins Sync** section, and add the namespace **youruser-movies-dev** to the **Namespace** field, next to the already existing **youruser-jenkins** namespace. Separate the entries by a space, and do not use a comma.

Do not change any other parameters on this page. You will create the **youruser-movies-dev** project in the next step.

Click **Save** to apply the changes to the Jenkins global configuration.
3. Create the OpenShift projects for the development and staging environments.
 - 3.1. Create a new project for the staging environment. Prefix the project name with your developer user name:

```
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-movies-stage
Now using project "youruser-movies-stage" on server
"https://api.cluster.domain.example.com:6443".
...output omitted...
```

- 3.2. Create a new project for the development environment. Prefix the project name with your developer user name:

```
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-movies-dev
Now using project "youruser-movies-dev" on server
"https://api.cluster.domain.example.com:6443".
...output omitted...
```

- 4.** Edit the build configuration resource file and add the Git URL of the Jenkinsfile.

- 4.1. Inspect the pipeline build configuration resource file at **~/D0288/labs/review-cicd/movies-bc.json** using a text editor.

- 4.2. Edit the Git URL where the application source is stored.

Edit the build configuration and change the value of the **spec.source.git.uri** attribute to point to your Git repository:

```
...output omitted...
git: {
  uri: "https://github.com/youruser/D0288-apps",
  ref: "review-cicd"
},
...output omitted...
```

- 5.** Inspect the Jenkins pipeline. The Jenkinsfile defines a multistage pipeline for deploying the microservice. Some of the stages are complete, while others are incomplete.

Enter your local clone of the **D0288-apps** Git repository, and checkout the **master** branch of the course repository to ensure you start this exercise from a known good state. Create a new branch called **review-cicd** to save any changes you make during this exercise.

- 5.1. Check out the **master** branch:

```
[student@workstation ~]$ cd D0288-apps
[student@workstation D0288-apps]$ git checkout master
...output omitted...
```

- 5.2. Create a new branch to save any changes you make during this exercise:

```
[student@workstation D0288-apps]$ git checkout -b review-cicd
Switched to a new branch 'review-cicd'
[student@workstation D0288-apps]$ git push -u origin review-cicd
...output omitted...
* [new branch]      review-cicd -> review-cicd
Branch review-cicd set up to track remote branch review-cicd from origin.
```

- 5.3. Inspect the Jenkinsfile at `~/D0288-apps/movies/Jenkinsfile`. The Jenkinsfile defines a multistage pipeline for deploying the microservice. Some of the stages are complete, while others are incomplete.

In the subsequent steps, review and complete the pipeline by implementing the lines marked as **TODO:** in the file.

6. Review and complete the pipeline by implementing the lines marked as **TODO:** in the Jenkinsfile. Edit the `~/D0288-apps/movies/Jenkinsfile` file and perform the following steps.

- 6.1. Open the `~/D0288-apps/movies/Jenkinsfile` file using a text editor. You can also copy the completed stages from the provided `~/D0288/solutions/review-cicd/Jenkinsfile` file.

- 6.2. The pipeline should run on a Jenkins agent node that supports Java and Apache Maven.

Add a label to the **node** section to enable the pipeline to run on a node labeled **maven**. Locate the line:

```
//TODO: Add label for the Maven jenkins agent
```

Replace it with the following:

```
...output omitted...
agent {
  node {
    label 'maven'
  }
}
...output omitted...
```

- 6.3. Customize the **environment** section and add values as per your environment.

Replace the value of **youruser** with your developer user account. Replace the value of the **NEXUS_SERVER** attribute with the value of the **RHT_OCP4_NEXUS_SERVER** variable from the `/usr/local/etc/ocp4.config` file.

```
...output omitted...
environment {
  //TODO: Customize these variables for your environment
  DEV_PROJECT = "youruser-books-dev"
  STAGE_PROJECT = "youruser-books-stage"
  APP_GIT_URL = "https://github.com/youruser/D0288-apps"
  NEXUS_SERVER = "http://nexus-common.apps.cluster.domain.example.com/
repository/java"

  // DO NOT CHANGE THE GLOBAL VARS BELOW THIS LINE
  APP_NAME = "movies"
}
...output omitted...
```

- 6.4. Inspect the **Launch new app in DEV env** stage. Add code to launch a new OpenShift application in the development environment.

Locate the lines:

```
// TODO: Create a new OpenShift application based on the ${APP_NAME}:latest image stream
// TODO: Expose the ${APP_NAME} service for external access
```

Replace it with the following lines:

```
...output omitted...
sh '''
  oc new-app --as-deployment-config ${APP_NAME}:latest -n ${DEV_PROJECT}
  oc expose svc/${APP_NAME} -n ${DEV_PROJECT}
'''
...output omitted...
```

- 6.5. Implement the **Wait for deployment in DEV env** stage. Add code to monitor the deployment until the pods are in the **Running** state.

Locate the line:

```
//TODO: Watch deployment until pod is in 'Running' state
```

Replace it with the following lines:

```
...output omitted...
steps {
  script {
    openshift.withCluster() {
      openshift.withProject( "${DEV_PROJECT}" ) {
        openshift.selector("dc", "${APP_NAME}").related('pods').untilEach(1) {
          return (it.object().status.phase == "Running")
        }
      }
    }
  }
...output omitted...
```

- 6.6. Inspect the **Promote to Staging Env** stage. Add code to tag the **latest** version of the image stream from the development environment, with a tag in the staging environment called **stage**.

Locate the line:

```
// TODO: Tag the ${APP_NAME}:latest image stream in the dev env as
${APP_NAME}:stage in staging
```

Replace it with the following lines:

```
...output omitted...
openshift.tag("${DEV_PROJECT}/${APP_NAME}:latest", "${STAGE_PROJECT}/
${APP_NAME}:stage")
...output omitted...
```

- 6.7. Inspect the **Deploy to Staging Env** stage. Add code to create a new application in the staging environment, using the tagged image stream from the previous stage.

Locate the line:

```
// TODO: Create a new app in staging and expose the service
```

Replace it with the following lines:

```
...output omitted...
sh '''
  oc project ${STAGE_PROJECT}
  oc new-app --as-deployment-config --name ${APP_NAME} -i ${APP_NAME}:stage
  oc expose svc/${APP_NAME}
'''
...output omitted...
```

The Jenkinsfile is now complete. Save your changes.

7. Edit the Maven proxy configuration file at `~/D0288-apps/movies/settings.xml`, and change the value of the `url` tag to point to the Nexus proxy server for your environment.

Locate the `<url>` tag below the line:

```
<!-- // TODO: Change the url attribute to the nexus proxy server URL for your
environment. -->
```

Add the value of the `RHT_OCP4_NEXUS_SERVER` variable from the `/usr/local/etc/ocp4.config` file to the `<url>` tag:

```
...output omitted...
<url>http://nexus-common.apps.cluster.domain.example.com/repository/java</url>
...output omitted...
```

8. Commit the changes to the Git repository from the `~/D0288-apps/movies` folder:

```
[student@workstation D0288-apps]$ cd ~/D0288-apps/movies
[student@workstation movies]$ git commit -a \
> -m "Completed Jenkinsfile for movies microservice"
...output omitted...
[student@workstation movies]$ git push
...output omitted...
[student@workstation movies]$ cd ~
```

9. Deploy the build configuration to the `youruser-movies-dev` project, and start a new build. Ensure that you add the appropriate security roles for projects where Jenkins performs some operation.

Once the build is running, monitor the progress of the pipeline using the OpenShift web console. Execution fails in the **Run Unit Tests** stage.



Note

Jenkins Pipelines are deprecated in OpenShift Container Platform 4. The equivalent functionality is present in the OpenShift Pipelines based on Tekton.

- 9.1. Ensure that you create the build configuration in the **youruser-movies-dev** namespace.

```
[student@workstation ~]$ oc project ${RHT_OCP4_DEV_USER}-movies-dev
Now using project "youruser-movies-dev" on server
...output omitted...
```

- 9.2. Add the **edit** role to the service account associated with the Jenkins deployment for the **youruser-movies-dev** and **youruser-movies-stage** projects.

```
[student@workstation ~]$ oc policy add-role-to-user \
> edit system:serviceaccount:${RHT_OCP4_DEV_USER}-jenkins:jenkins \
> -n ${RHT_OCP4_DEV_USER}-movies-dev
clusterrole.rbac.authorization.k8s.io/edit added:
"system:serviceaccount:youruser-jenkins:jenkins"

[student@workstation ~]$ oc policy add-role-to-user \
> edit system:serviceaccount:${RHT_OCP4_DEV_USER}-jenkins:jenkins \
> -n ${RHT_OCP4_DEV_USER}-movies-stage
clusterrole.rbac.authorization.k8s.io/edit added:
"system:serviceaccount:youruser-jenkins:jenkins"
```

- 9.3. Create the build configuration:

```
[student@workstation ~]$ oc create \
> -f ~/D0288/labs/review-cicd/movies-bc.json
buildconfig.build.openshift.io/movies-pipeline created
```

- 9.4. Open a web browser and navigate the OpenShift web console.

Find the host name of your OpenShift web console inspecting the routes on the **openshift-console** project.

```
[student@workstation ~]$ oc get route console -n openshift-console \
> -o jsonpath='{.spec.host}{"\n"}'
console-openshift-console.apps.cluster.domain.example.com
```

Log in using your developer user account.

- 9.5. Select the **youruser-movies-dev** project.

In the left navigation bar, click **Builds** → **Build Configs**, and verify that a build configuration named **movies-pipeline** is visible.

- 9.6. Start a new build from the command line to initiate the pipeline:

```
[student@workstation ~]$ oc start-build movies-pipeline
build.build.openshift.io/movies-pipeline-1 started
```

You can also use the OpenShift web console to start a build. Click the **movies-pipeline** build configuration, and then click **Actions** → **Start Build** in the top right corner to start a build.

10. Investigate the test failures using the Jenkins pipeline execution log. Fix the failing unit tests and run the pipeline again. The pipeline now fails in the **Static Code Analysis** stage.

- 10.1. From the OpenShift web console left menu, click **Builds** → **Builds** to open the **Builds** page for the application. Click **movies-pipeline-1** to open the **Build Details** page. In the **Build Details** page, click **View Logs** (below **Build 1** with the red 'x' mark) to view open the Jenkins pipeline execution log.

- 10.2. The Jenkins pipeline execution log shows an error in the **Run Unit Tests** stage:

```
...output omitted...
[ERROR] Tests run: 4, Failures: 2, Errors: 0, Skipped: 0, Time elapsed: 10.401 s
<<< FAILURE! - in com.redhat.movies.MoviesApplicationTests
[ERROR] testGetAllMovies(com.redhat.movies.MoviesApplicationTests)  Time elapsed:
0.172 s  <<< FAILURE!
java.lang.AssertionError: expected:<7> but was:<6>
  at com.redhat.movies.MoviesApplicationTests.testGetAllMovies
(MoviesApplicationTests.java:52)

[ERROR] testGetStatus(com.redhat.movies.MoviesApplicationTests)  Time elapsed:
0.008 s  <<< FAILURE!
org.junit.ComparisonFailure: expected:<[Ready]> but was:<[OK]>
  at com.redhat.movies.MoviesApplicationTests.testGetStatus
(MoviesApplicationTests.java:65)
...output omitted...
```

You should see two tests failing. The failed test methods, along with the line numbers in source code, are provided in the error log. The first failure indicates that there are six movies in the catalog, but the test wrongly asserts that there are seven. The second failure indicates that the response from the API is the string **OK**, but the test is asserting that the response is **Ready**.

- 10.3. Fix the failing unit tests. Open the **~/DO288-apps/movies/src/test/java/com/redhat/movies/MoviesApplicationTests.java** file in a text editor.

- 10.4. Locate the following line in the **testGetAllMovies()** test method:

```
Assert.assertEquals(7, movies.size());
```

Replace it with the following correct assertion:

```
Assert.assertEquals(6, movies.size());
```

- 10.5. Locate the following line in the **testGetStatus()** test method:

```
Assert.assertEquals("Ready", response.getBody());
```

Replace it with the following correct assertion:

```
Assert.assertEquals("OK", response.getBody());
```

Save your changes.

- 10.6. Commit your changes and push the fixed test file to your **youruser-review-cicd** branch.

```
[student@workstation ~]$ cd ~/DO288-apps/movies
[student@workstation movies]$ git commit -a -m "Fixed failed unit test."
...output omitted...
[student@workstation movies]$ git push
...output omitted...
[student@workstation movies]$ cd ~
```

10.7. Start a new build to run the pipeline again.

```
[student@workstation ~]$ oc start-build movies-pipeline
build.build.openshift.io/movies-pipeline-2 started
```

10.8. Monitor the progress of the pipeline execution for the build **movies-pipeline-2** from the **Build Details** page.

The unit tests should now pass. The pipeline now fails in the **Static Code Analysis** stage.

11. Fix the errors pointed out by the static code analysis tools, and then run the pipeline again.

11.1. Check the Jenkins pipeline execution log by clicking **View Logs**. The logs shows that the static code analysis tool is complaining about unused imports and variables in the code.

```
...output omitted...
[INFO] --- maven-pmd-plugin:3.12.0:check (default-cli) @ movies ---
[INFO] PMD Failure: com.redhat.movies.MoviesController:5 Rule:UnusedImports
Priority:4 Avoid unused imports such as 'java.io.File'.
[INFO] PMD Failure: com.redhat.movies.MoviesController:17 Rule:UnusedPrivateField
Priority:3 Avoid unused private fields such as 'flag'..
...output omitted...
```

11.2. Fix the code in the class identified by the static code analysis tool. Open the `~/DO288-apps/movies/src/main/java/com/redhat/movies/MoviesController.java` file in a text editor.

11.3. Delete the following line in the imports section at the top of the file:

```
import java.io.File;
```

11.4. Delete the following line in the file:

```
private String flag = "READY";
```

Save your changes.

11.5. Commit your changes and push the fixed test file to your **youruser-review-cicd** branch.

```
[student@workstation ~]$ cd ~/DO288-apps/movies
[student@workstation movies]$ git commit -a \
> -m "Removed unused imports and variables."
...output omitted...
[student@workstation movies]$ git push
...output omitted...
[student@workstation movies]$ cd ~
```

- 11.6. Start a new build to run the pipeline again.

```
[student@workstation ~]$ oc start-build movies-pipeline
build.build.openshift.io/movies-pipeline-3 started
```

- 11.7. Monitor the progress of the pipeline execution for the build **movies-pipeline-3** from the **Build Details** page.

The pipeline passes the static code analysis stage and waits for approval in the **Promote to Staging Env** stage.

12. Verify that the microservice is deployed and running in the development environment.

- 12.1. Before promoting the microservice to the staging environment, verify that the application is running in the development environment.

Verify that the movies microservice is running:

```
[student@workstation ~]$ oc get pods -n ${RHT_OCP4_DEV_USER}-movies-dev
NAME        READY   STATUS    RESTARTS   AGE
...output omitted...
movies-1-jpjvc   1/1     Running      0          17m
```

- 12.2. Verify that the staging environment does not have any pods running:

```
[student@workstation ~]$ oc get pods -n ${RHT_OCP4_DEV_USER}-movies-stage
No resources found.
```

- 12.3. Get the route URL for the movies microservice:

```
[student@workstation ~]$ oc get route/movies \
> -n ${RHT_OCP4_DEV_USER}-movies-dev \
> -o jsonpath='{.spec.host}{"\n"}'
movies-youruser-movies-dev.apps.cluster.domain.example.com
```

- 12.4. Navigate to the **/movies** endpoint in the route URL from the previous step, and verify that the movies microservice shows movie data.

```
[student@workstation ~]$ curl \
> movies-${RHT_OCP4_DEV_USER}-movies-dev.${RHT_OCP4_WILDCARD_DOMAIN}/movies
[
  {
    "movieId" : 1,
    "name" : "The Godfather",
```

```

    "genre" : "Crime/Thriller"
  },
  {
    "movieId" : 2,
    "name" : "Star Wars",
    "genre" : "Sci-Fi"
  ...output omitted...
  "movieId" : 6,
  "name" : "The Silence of the Lambs",
  "genre" : "Drama"
}
]

```

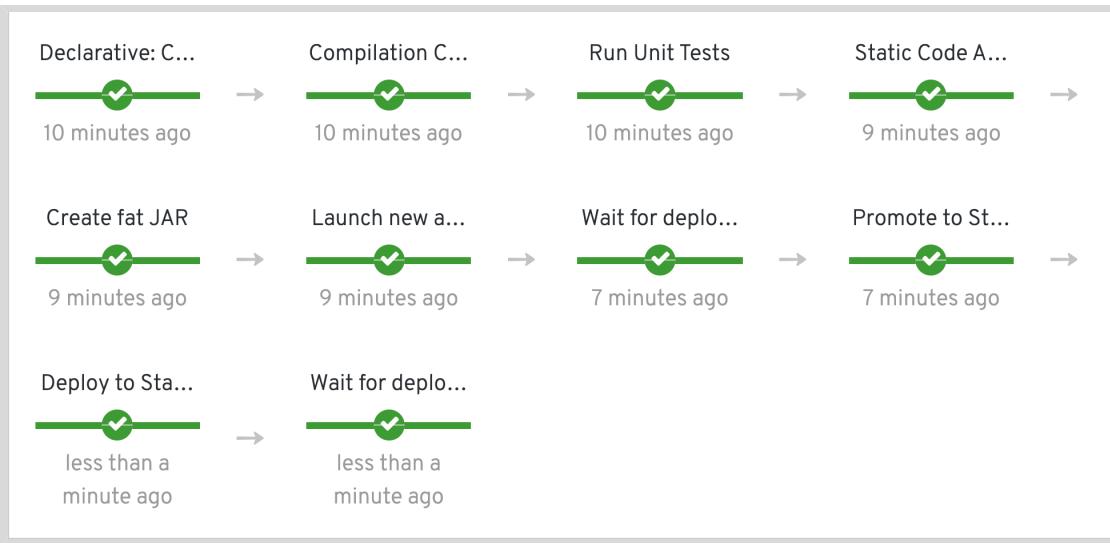
13. Promote the microservice to the staging environment.
 - 13.1. Once you are satisfied that the microservice is working as intended, promote the microservice to the staging environment.
 - 13.2. From the **Build Details** page in the OpenShift web console for the build **movies-pipeline-3**, click **Input Required** to view the Jenkins pipeline execution log.
 - 13.3. From the Jenkins console left menu, click **Paused for Input**. Click **Proceed** in the **Promote to Staging?** prompt.
 - 13.4. Jenkins switches to the execution log window and continues with executing the **Deploy to Staging Env** stage.
14. Verify that the pipeline execution is successful. Verify that the microservice is deployed and running in the staging environment.
 - 14.1. Pipeline execution should complete and you should see the following in the execution log:

```

...output omitted...
Deployment to Staging env is complete.
Access the API endpoint at the URL
http://movies-youruser-movies-stage.apps.cluster.domain.example.com/movies.
...output omitted...
[Pipeline] End of Pipeline
Finished: SUCCESS

```

- 14.2. Click the URL for the movies microservice deployed in the staging environment. Verify that the microservice is functionally similar to the version deployed in the development environment.
- 14.3. The **Build Details** page for **movies-pipeline-3** shows the complete pipeline execution status.



15. Grade your work.

Run the following command on **workstation** to verify that all tasks were accomplished:

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

16. Cleanup: Delete the **youruser-jenkins**, **youruser-movies-dev**, and **youruser-movies-stage** projects.

```
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-movies-dev
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-movies-stage
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-jenkins
```

Finish

On **workstation**, run the **lab todo-migrate finish** command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

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

This concludes the lab.

Summary

In this chapter, you learned:

- A CI/CD pipeline is an automated process to frequently integrate and deploy code to production in a project with multiple developers.
- OpenShift Container Platform has built-in support for deploying CI/CD pipelines using Jenkins.
- Pipelines consist of multiple stages, and you can control where each stage is run using agents.
- The OpenShift Client Jenkins Plugin provides an easy to use DSL to script pipelines and interact with an OpenShift cluster.

Chapter 9

Building Applications for OpenShift

Goal

Create and deploy applications on OpenShift.

Objectives

- Integrate a containerized application with non-containerized services.
- Deploy containerized third-party applications following recommended practices for OpenShift.
- Use a Red Hat OpenShift Application Runtime to deploy an application.

Sections

- Integrating External Services (and Guided Exercise)
- Deploying Containerized Applications (and Guided Exercise)
- Deploying Applications with Red Hat OpenShift Application Runtimes (and Guided Exercise)

Lab

Building Cloud-Native Applications for OpenShift

Integrating External Services

Objectives

After completing this section, you should be able to integrate a containerized application with non-containerized services.

Review of Red Hat OpenShift Services

A typical service in OpenShift has both a name and a selector. A service uses its selector to identify pods that should receive application requests sent to the service. OpenShift applications use the service name to connect to the service endpoints.

Similarly, an FQDN allows an application to use a name to access the endpoints of a public service. However, OpenShift services enable application access to service endpoints without requiring public exposure of the service.

An application discovers a service using either environment variables, or the OpenShift internal DNS server. Using environment variables requires the service to be defined before the application pod is created; otherwise, the application will not receive the environment variables.

Using the OpenShift internal DNS server is more flexible because it allows applications to discover services dynamically. The service name becomes a local DNS host name for all pods inside the same OpenShift cluster that contains the service. OpenShift adds the **svc.cluster.local** domain suffix to the DNS resolver search path of all containers. OpenShift also assigns the **service-name.project-name.svc.cluster.local** host name to each service.

For example, if the **myapi** service exists in the **myproject** service, then all pods in the same OpenShift cluster can resolve the **myapi.myproject.svc.cluster.local** host name to get the service IP address. The following short host names are also available:

- Pods from the same project can use the **myapi** service name as a short host name, without any domain suffix.
- Pods from a different project can use the service name and **myapi.myproject** project name as a short host name, without the **svc.cluster.local** domain suffix.

Defining External Services

OpenShift supports multiple approaches to defining services that do not contain selectors. This allows an OpenShift service to point to one or more hosts outside of the OpenShift cluster.

Consider an application that you wish to containerize, which depends on an existing database service that is not readily available inside your OpenShift cluster. You do not need to migrate the database service to OpenShift before containerizing the application. Instead, begin designing your application to interact with OpenShift services, including the database service. Simply create an OpenShift service that references the external database service endpoints.

When you create OpenShift services for the endpoints of external services, your applications are able to discover both internal and external services. Additionally, if the endpoints of an external service change, then you do not need to reconfigure affected applications. Instead, update the endpoints for the corresponding OpenShift service.

Creating An External Service

The easiest approach to creating an external service is to use the **oc create service externalname** command with the **--external-name** option:

```
[user@host ~]$ oc create service externalname myservice \
> --external-name myhost.example.com
```

The previous example can also accept an IP address instead of a DNS name.

Applications running inside the OpenShift cluster then use the service name the same way they would use regular service, as either an environment variable or as a local host name.

Defining Endpoints for a Service

A typical service creates *endpoint* resources dynamically, based on the selector attribute of the service. The **oc status** and **oc get all** commands do not display these resources. You can use the **oc get endpoints** command to display them.

If you use the **oc create service externalname --external-name** command to create a service, then the command also creates an endpoint resource that points to the host name or IP address given as an argument.

If you do not use the **--external-name** option, it does not create an endpoint resource. In this case, use the **oc create -f** command and a resource definition file to explicitly create the endpoint resources.

If you create an endpoint from a file, you can define multiple IP addresses for the same external service, and rely on the OpenShift service load-balancing features. In this scenario, OpenShift does not add or remove addresses to account for the availability of each instance. An external application must update the list of IP addresses in the endpoint resource.

See the references at the end of this section for more information about endpoint resource definition files.



References

Review the *Type ExternalName* section of the Service concepts documentation for Kubernetes at
<https://kubernetes.io/docs/concepts/services-networking/service/#externalname>

Review the OpenShift DNS naming conventions in the *Networking* chapter of the *Architecture Guide* for Red Hat OpenShift Container Platform 4.5; at
https://access.redhat.com/documentation/en-us/red_hat_openshift_container_platform/4.5/html/networking/index

Authors note: The following resource does not exist yet in the 4.x version of the documentation for OpenShift. However, the information linked below is still relevant to the current 4.5 release of the product.

Review the *Integrating External Services* chapter of the *Developer Guide* for Red Hat OpenShift Container Platform 3.11; at
https://docs.openshift.com/container-platform/3.11/dev_guide/integrating_external_services.html

► Guided Exercise

Integrating an External Service

In this exercise, you will deploy an application on OpenShift that communicates with a database running outside the OpenShift cluster.

Outcomes

You should be able to:

- Deploy the To Do List application from source code.
- Create a database service for the application that points to a MariaDB database server outside the OpenShift cluster.
- Verify that the application uses the data preloaded into the database.

Before You Begin

To perform this exercise, ensure you have access to:

- A running OpenShift cluster.
- The To Do List application in the Git repository (**todo-single**).
- A Nexus server that provides the npm dependencies required by the application (**restify**, **sequelize**, and **mysql**).
- The Node.js 12 S2I builder image.
- A prepopulated MariaDB database server running outside your OpenShift cluster.

Run the following command on **workstation** to validate the prerequisites and deploy the To Do List application:

```
[student@workstation ~]$ lab external-service start
```

The previous command runs the **oc-new-app.sh** script in the **~/DO288/labs/external-service** folder to deploy the application. You can review this script if you need more information about the To Do List application resources used during this exercise.

► 1. Inspect the To Do List application resources.

- 1.1. Load the configuration of your classroom environment.

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

- 1.2. Log in to OpenShift using your developer user account.

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful.
...output omitted...
```

- 1.3. Enter the `youruser-external-service` project that hosts the To Do List application:

```
[student@workstation ~]$ oc project ${RHT_OCP4_DEV_USER}-external-service
Now using project "youruser-external-service" on server
"https://api.cluster.domain.example.com:6443".
```

- 1.4. Verify that the OpenShift project has a single application named **todoapp** that is built from sources:

```
[student@workstation ~]$ oc status
...output omitted...
http://todo-youruser-external-service.apps.domain.cluster.example.com to pod port
8080-tcp (svc/todoapp)
dc/todoapp deploys istag/todoapp:latest <
bc/todoapp source builds https://github.com/youruser/D0288-apps on openshift/
nodejs:12
deployment #1 deployed 26 seconds ago - 1 pod
...output omitted...
```

- 1.5. Check that the application is ready and running:

[student@workstation ~]\$ oc get pod				
NAME	READY	STATUS	RESTARTS	AGE
todoapp-1-6z6qg	1/1	Running	0	1m
todoapp-1-build	0/1	Completed	0	4m
todoapp-1-deploy	0/1	Completed	0	1m

- 1.6. Inspect the environment variables inside the application pod to get the database connection parameters:

```
[student@workstation ~]$ oc rsh todoapp-1-6z6qg env | grep DATABASE
DATABASE_PASSWORD=redhat123
DATABASE_SVC=tododb
DATABASE_USER=todoapp
DATABASE_INIT=false
DATABASE_NAME=todo
```

- ▶ 2. Verify that the To Do List application cannot reach the database server specified by the **DATABASE_SVC** environment variable, which is **tododb**.
- 2.1. Get the host name where the application is exposed. Because the host name is very long, save it into a shell variable.

```
[student@workstation ~]$ HOSTNAME=$(oc get route todoapp \
> -o jsonpath='{.spec.host}')
[student@workstation ~]$ echo ${HOSTNAME}
todoapp-youruser-external-service.apps.cluster.domain.example.com
```

- 2.2. Use the **curl** command, the host name from the previous step, and the API resource path **/todo/api/items/6** to fetch an item from the database. The error message from application indicates that it cannot resolve the **tododb** service host name.

```
[student@workstation ~]$ curl -si http://${HOSTNAME}/todo/api/items/6
HTTP/1.1 500 Internal Server Error
...output omitted...
{"message": "getaddrinfo ENOTFOUND tododb tododb:3306"}
```

► 3. Inspect the external database.

- 3.1. Find the host name of the external MariaDB server. This host name is the same as your OpenShift cluster's wildcard domain, replacing **apps** with **mysql**.

```
[student@workstation ~]$ echo mysql.ocp-${RHT_OCP4_WILDCARD_DOMAIN#"apps."}
mysql.ocp-cluster.domain.example.com
```

- 3.2. Use the **mysqlshow** command to connect to the **todo** database in the external MariaDB server, and verify that it contains the **Item** table.

Use the host name from the previous step. Use **todoapp** as the user and **redhat123** as the password:

```
[student@workstation ~]$ mysqlshow -hmysql.cluster.domain.example.com \
> -utodoapp -predhat123 todo
Database: todo
+-----+
| Tables |
+-----+
| Item   |
+-----+
```

► 4. Create an OpenShift service that connects to the external database instance, and verify that the application now gets data from the external database.

- 4.1. Use the **oc create svc** command to create a service based on an external name, and the database server host name from the previous step.

```
[student@workstation ~]$ oc create svc externalname tododb \
> --external-name mysql.cluster.domain.example.com
service/tododb created
```

- 4.2. Verify that the **tododb** service exists and shows an external IP, but no cluster IP:

```
[student@workstation ~]$ oc get svc
NAME      TYPE        CLUSTER-IP      EXTERNAL-IP      ...
todoapp   ClusterIP   172.30.140.201  <none>          ...
tododb    ExternalName <none>          mysql.cluster.domain.example.com ...
```

- 4.3. Verify that the application now can get data from the external database.

Use the **curl** command again:

```
[student@workstation ~]$ curl -si http://${HOSTNAME}/todo/api/items/6
HTTP/1.1 200 OK
...
{"id":6,"description":"Verify that the To Do List application works","done":false}
```

- ▶ 5. Clean up. Delete the **youruser-external-service** project from OpenShift.

```
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-external-service
```

Finish

On **workstation**, run the **lab external-service finish** command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises. The finish action releases this project and its resources.

```
[student@workstation ~]$ lab external-service finish
```

This concludes the guided exercise.

Containerizing Services

Objectives

After completing this section, students should be able to review and deploy containerized third-party application by following recommended practices for OpenShift.

Reviewing Containerized Applications to Deploy on OpenShift

Occasionally, you must deploy an application on OpenShift whose vendor either provides a Dockerfile to build its container image, or provides a prebuilt container image in a registry server. This application may offer a back-end service that your custom application requires, such as a database or messaging system, or a service that your development process requires, such as a hosted container registry, artifact repository, or collaboration tool.

The fact that the application is already containerized by its vendor does not necessarily mean it runs well on OpenShift. Your first attempt to deploy that application using the `oc new-app` command may fail for one of several possible reasons. Resolving these issues may require customization of the application's deployment configuration or changes to its Dockerfile.

The following is a non-exhaustive list of common issues:

- Your application does not run under the default OpenShift security policies defined by the **restricted** Security Context Constraint (SCC). The application's container image may expect to run as the **root** or another fixed user. It may also require custom SELinux policies and other privileged settings.

Usually, you can change the Dockerfile so that the application complies with the default OpenShift security policies. If not, then carefully evaluate the risk of running the application under a more relaxed security policy.

- Your application requires custom settings for resource requests and resource limits. This usually happens with legacy, monolithic applications, that were not designed for a microservice-based architecture. OpenShift cluster administrators usually set up default limit ranges that provide default values for resource requests and limits to your project, and these may be too small for your legacy application.

You can specify increased resource requests and resource limits on your deployment configuration to override the defaults set by the cluster administrator. If your application requirements are over the resource quota set by your OpenShift cluster administrators, the administrators must either increase the resource quota for your user, or provide a service account for your application with an increased resource quota.

- Your application image defines configuration settings that are fine for a standalone container runtime, but not for an OpenShift or a generic Kubernetes cluster. Applications usually read configuration settings from environment variables, and usually define defaults by using **ENV** instructions in a Dockerfile .

There is no need to change the Dockerfile to change its environment variables. You can override any environment variable, including those that are not explicitly set by **ENV** instructions in the Dockerfile, on your application deployment configuration.

- Your application requires persistent storage. Applications usually define storage requirements using **VOLUME** instructions in the Dockerfile. Sometimes these applications expect to use host folders for their volumes, which is usually forbidden by your OpenShift cluster administrators. These applications may expect to have direct access to network storage, which is also not recommended from a security standpoint.

The proper way to provide persistent storage for applications on OpenShift is to define persistent volume claim (PVC) resources, and to configure the application deployment configuration to attach these PVCs to the volumes from the application container. OpenShift manages network storage on behalf of your applications, and your cluster administrators manage security policies and performance levels for storage.

Your vendor may provide Kubernetes resource files to deploy applications, and these resource files may require customization to work under OpenShift. Deployment resources that work with upstream Kubernetes (and also with other vendor distributions of Kubernetes) may not work with OpenShift because standard Kubernetes does not come with secure default settings.

Sometimes, vendors provide Kubernetes deployment resources assuming the user is a cluster administrator. These vendors may not consider that an OpenShift cluster is usually shared by multiple organizations and teams, which would not be granted cluster administrator privileges for their own safety.

Reviewing The Dockerfile for the Nexus Application

The Nexus application, developed by Sonatype, is a repository manager commonly used by developers to store artifacts required by applications, such as dependencies. This application can store artifacts for several technologies such as Java, .NET, Docker, Node, Python, and Ruby.

Sonatype provides a Dockerfile to build the Nexus application using a base container image from Red Hat. The resulting image is available from the Red Hat Container Catalog, which attests that it complies with a core set of guidelines from Red Hat. The Dockerfile is configured to support OpenShift features, such as allowing container images to run as a random user account.

Base Image and Container Metadata

The Dockerfile project is available at <https://github.com/sonatype/docker-nexus3/tree/3.18.0>. The following is provided in the **Dockerfile.rhel** file:

```
...output omitted...
FROM      registry.access.redhat.com/rhel7/rhel ①

MAINTAINER Sonatype <cloud-ops@sonatype.com>
...output omitted...

LABEL name="Nexus Repository Manager" \
...output omitted...
io.k8s.description="The Nexus Repository Manager server \
with universal support for popular component formats." ②
io.k8s.display-name="Nexus Repository Manager" \
io.openshift.expose-services="8081:8081" \
io.openshift.tags="Sonatype,Nexus,Repository Manager"

...output omitted...
```

- ① Specifies the Red Hat Enterprise Linux 7 container image as the base image. There is also an alternative **Dockerfile** file that uses the Red Hat Universal Base Image 8 located in the GitHub project.
- ② Defines the image metadata for Kubernetes and OpenShift.

The Nexus Dockerfile also uses the Dockerfile **ARG** instruction. An **ARG** instruction defines a variable that only exists during the image build process. Unlike an **ENV** instruction, these variables are not a part of the resulting container image:

```
...output omitted...
ARG NEXUS_VERSION=3.18.0-01
ARG NEXUS_DOWNLOAD_URL=https://nexus/3/nexus-${NEXUS_VERSION}-unix.tar.gz
ARG NEXUS_DOWNLOAD_SHA256_HASH=e1d9...c99e
...output omitted...
```

The build processes uses these three variables to control and verify the installed version of Nexus in the container image.

The Dockerfile also defines environment variables that are present the built container image:

```
...output omitted...
# configure nexus runtime
ENV SONATYPE_DIR=/opt/sonatype
ENV NEXUS_HOME=${SONATYPE_DIR}/nexus \
    NEXUS_DATA=/nexus-data \
    NEXUS_CONTEXT='' \
    SONATYPE_WORK=${SONATYPE_DIR}/sonatype-work \
DOCKER_TYPE='rh-docker'
...output omitted...
```

The installation process uses the **DOCKER_TYPE** environment variable to customize installation and configuration. A value of **rh-docker** invokes any configuration changes necessary for a Red Hat certified container image.

The Nexus Installation Process

Sonatype maintains Chef cookbooks that standardize the installation of the Nexus application. Chef is an open source configuration management technology, similar to Puppet and Ansible, that aims to streamline the process of configuring and managing servers. A Chef *cookbook* is a collection of Chef *recipes*, and each recipe defines the configuration for a particular set of system resources.

The Nexus Dockerfile uses the Chef installation process to build the Nexus container image:

```
...output omitted...
ARG NEXUS_REPOSITORY_MANAGER_COOKBOOK_VERSION="release-0.5.20190212-..." ①
ARG NEXUS_REPOSITORY_MANAGER_COOKBOOK_URL="https://github.com/sonatype/..."

ADD solo.json.erb /var/chef/solo.json.erb ②

# Install using chef-solo
RUN curl -L https://www.getchef.com/chef/install.sh | bash \ ③
&& /opt/chef/embedded/bin/erb /var/chef/solo.json.erb > /var/chef/solo.json \
&& chef-solo \
```

```
--node_name nexus_repository_red_hat_docker_build \
--recipe-url ${NEXUS_REPOSITORY_MANAGER_COOKBOOK_URL} \
--json-attributes /var/chef/solo.json \
&& rpm -qa *chef* | xargs rpm -e \
&& rpm --rebuilddb \
&& rm -rf /etc/chef \
&& rm -rf /opt/chefdk \
&& rm -rf /var/cache/yum \
&& rm -rf /var/chef
...output omitted...
```

- ➊ The Dockerfile uses variables to control the version of the Chef cookbook that installs Nexus. The cookbook URL is provided as an argument to a Chef command in a later **RUN** instruction.
- ➋ In the same directory as the Dockerfile, the **solo.json.erb** file contains Chef configuration details. The Dockerfile adds this file to the container image to enable the Chef installation process.
- ➌ The installation of Nexus is accomplished with a single **RUN** instruction that:
 - Downloads and installs Chef.
 - Executes the Chef cookbook to install Nexus with the **chef-solo** command.
 - Removes downloaded files, Chef RPMs, and other extraneous files.

The Chef cookbook also configures file permissions so the data and log folders are writable by `gui=0`, thus complying with OpenShift default security policies.

Container Execution Environment

The bottom of the Dockerfile provides metadata to enable a container runtime to create a container from the image:

```
...output omitted...
VOLUME ${NEXUS_DATA} ①

EXPOSE 8081 ②
USER nexus ③

ENV INSTALL4J_ADD_VM_PARAMS="-Xms1200m -Xmx1200m ..." ④

ENTRYPOINT ["/uid_entrypoint.sh"] ⑤
CMD ["sh", "-c", "${SONATYPE_DIR}/start-nexus-repository-manager.sh"] ⑥
```

- ➊ Configures the **/nexus-data** directory as a volume, because the value of the **NEXUS_DATA** variable is **/nexus-data**. The Nexus application stores application data and resources in this directory.
- ➋ The Nexus application communicates over port 8081.
- ➌ The Nexus application runs as the **nexus** user. The Chef installation process creates and configures this user.
- ➍ The Nexus application works on a Java Virtual Machine (JVM). The JVM needs options to size its heap when running on a standalone container runtime. On OpenShift, you must override these options and let the JVM obey OpenShift resource limits and resource requests. Failure to do so may result in scheduling and stability issues.
- ➎ During installation, the Chef cookbook creates the **/uid_entrypoint.sh** script. The objective of the **/uid_entrypoint.sh** script is to recognize the user ID that is running the

- application, and to define it to the **nexus** user. This allows the application to execute normally with the default **restricted** security context constraint.
- ⑥ The command that starts the Nexus application. The entry point executes this command after it adjusts permissions to account for the randomly assigned process UID.

Customizing OpenShift Resources for the Nexus Image

If you build a Nexus container image from its official Dockerfile, and deploy it using a command such as **oc new-app --as-deployment-config --docker-image**, it may not work reliably. To ensure the container image works well, you must make a few customizations to its deployment configuration.

Setting Resource Requests and Limits

Red Hat recommends that applications deployed to OpenShift define resource requests and resource limits. If you are not using very old releases of the Java Virtual Machine (JVM), you do not need to define JVM heap options.

Resource requests state how much CPU and memory an application needs to run. The OpenShift scheduler finds a worker node in the cluster with sufficient available CPU and memory to run the application, so it will not fail to start with an out of memory error.

Resource Limits state how much the CPU and memory usage of an application may increase over time. OpenShift sets Linux kernel cgroups for the application container, which prevents it from ever violating those limits. The Linux kernel kills containers that violate these limits, and OpenShift starts replacement containers, thus mitigating issues caused by memory leaks, infinite loops, and deadlocks. Recent releases of the JVM automatically size the heap and other internal data structures to not violate current cgroups settings.

You define resource requests and resource limits inside the pod template of a deployment configuration:

```
...output omitted...
- apiVersion: v1
  kind: DeploymentConfig
...output omitted...
  spec:
...output omitted...
    template:
...output omitted...
    spec:
      containers:
...output omitted...
      resources:
        limits:
          cpu: "1"
          memory: 2Gi
        requests:
          cpu: 500m
          memory: 256Mi
...output omitted...
```

In the Nexus scenario, you must also override the **INSTALL4J_ADD_VM_PARAMS** environment variable to remove any JVM options related to memory sizing:

```

...output omitted...
- apiVersion: v1
  kind: DeploymentConfig
...output omitted...
  spec:
    ...output omitted...
    template:
      ...output omitted...
      spec:
        containers:
          - env:
              - name: INSTALL4J_ADD_VM_PARAMS
                value: -Djava.util.prefs.userRoot=/nexus-data/javaprefs
...output omitted...

```

The preceding example assumes that you are not overriding the **NEXUS_DATA** environment variable to specify a different volume path for the container's persistent storage.

Implementing Probes

Red Hat recommends that applications deployed to OpenShift define health probes. Nexus provides an API to determine if the service is alive, and if it has any deadlocks. You can not use a simple HTTP probe, however, because the Nexus API requires authentication and the API returns an HTTP **200** status code even if Nexus is unhealthy.

You can create a script that authenticates with the Nexus API and parses responses from the API. The following script verifies if the Nexus service is ready to serve requests:

```

#!/bin/sh
curl -si -u admin:$(cat /nexus-data/admin.password) \
  http://localhost:8081/service/metrics/ping | grep pong

```

- ➊ The **-u** option passes a username and password in the HTTP GET request.
- ➋ Nexus provides the **/service/metrics/ping** endpoint, which must return the **pong** value when the service is ready. If **pong** is not in the response, the **grep** command exits with a nonzero status code.

The **admin** user is authorized to make requests to health check endpoints. When the Nexus application initializes for the first time, it generates a random password for the **admin** user. Nexus stores the random password in the **/nexus-data/admin.password** file.

In a similar way, the following script determines if the Nexus service is alive:

```

#!/bin/sh
curl -si -u admin:$(cat /nexus-data/admin.password) \
  http://localhost:8081/service/metrics/healthcheck | grep healthy | \
  grep true

```

Because these probe scripts are simple, you can put the content of each script in the appropriate probe stanza of the Nexus container specification:

```

...output omitted...
- apiVersion: v1
  kind: DeploymentConfig
...output omitted...
  spec:
    ...output omitted...
      template:
        ...output omitted...
          spec:
            containers:
...output omitted...
  livenessProbe:
    exec:
      command:
        - /bin/sh ①
        - "-c" ②
        - > ③
          curl -si -u admin:$(cat /nexus-data/admin.password)
          http://localhost:8081/service/metrics/healthcheck |
          grep healthy | grep true
    failureThreshold: 3
    initialDelaySeconds: 10
    periodSeconds: 10
...output omitted...

```

- ① Use the `/bin/sh` shell.
- ② The `-c` option indicates that a single command is executed in the shell.
- ③ The YAML multiline continuation indicator. This allows you to split a long string over several lines.

For a large probe script, consider modifying the Dockerfile and adding the probe scripts to the container image.



References

Further information about the Nexus service is available in the Nexus Repository Manager Documentation at
<https://help.sonatype.com/repomanager3>

For more information about Nexus application health and metrics, see
Nexus 3 Server Metrics and Health Information – Sonatype Support
<https://support.sonatype.com/hc/en-us/articles/226254487-Nexus-3-Server-Metrics-and-Health-Information>

For more information about JVM memory settings on containers, see
Java inside docker: What you must know to not FAIL
<https://developers.redhat.com/blog/2017/03/14/java-inside-docker/>

► Guided Exercise

Deploying a Containerized Nexus Server

In this exercise, you will deploy a Nexus server as containerized application, following OpenShift recommended practices.

Outcomes

You should be able to:

- Build a container image from a Dockerfile.
- Verify that a template uses an image stream that uses a container image from a private registry.
- Verify that a template defines resource requests and resource limits for a Java application, and overrides environment variables that would set heap sizes.
- Configure a template to use a persistent volume claim.
- Configure a template to use liveness and readiness probes.
- Start a new application using the template file.

Before You Begin

To perform this exercise, ensure you have access to:

- A running OpenShift cluster.
- The Red Hat Universal Base Image 8 (**ubi8/ubi**).
- The source files for the Nexus application image in the Git repository (**nexus3**).

Run the following command on **workstation** to validate the prerequisites and download the files required to complete this exercise:

```
[student@workstation ~]$ lab nexus-service start
```

► 1. Review and build the Nexus **Dockerfile** file.

- 1.1. Enter the **nexus3** subdirectory of your local clone of the **DO288-apps** Git repository, and then checkout the **master** branch of the course repository to ensure that you start this exercise from a known good state:

```
[student@workstation ~]$ cd ~/DO288-apps/nexus3
[student@workstation nexus3]$ git checkout master
...output omitted...
```

- 1.2. Inspect the **Dockerfile** file, and observe that the Nexus application persists its files in the **/nexus-data** folder:

```
[student@workstation nexus3]$ grep VOLUME Dockerfile
VOLUME ${NEXUS_DATA}
[student@workstation nexus3]$ grep NEXUS_DATA Dockerfile
NEXUS_DATA=/nexus-data \
...output omitted...
```

The Dockerfile defines the **NEXUS_DATA** environment variable with a value of `/nexus-data`. The Dockerfile uses this variable to define a volume where Nexus stores application data.

13. Inspect the **Dockerfile** file and observe that it defines an environment variable that sets Java Virtual Machine (JVM) heap sizes. Later, you must override this environment variable so that the OpenShift deployment configuration controls the pod memory settings.

```
[student@workstation nexus3]$ grep ENV Dockerfile
...output omitted...
ENV INSTALL4J_ADD_VM_PARAMS="-Xms1200m -Xmx1200m -XX:MaxDirectMemorySize=2g -
Djava.util.prefs.userRoot=${NEXUS_DATA}/javaprefs"
```

14. Load your classroom environment configuration.

Run the following command to load the environment variables created in the first guided exercise:

```
[student@workstation nexus3]$ source /usr/local/etc/ocp4.config
```

15. Build the container image and push it to your personal account at Quay.io. You can either run or copy and paste the commands from **build-nexus-image.sh** script at `~/D0288/labs/nexus-service`.

During the build, ignore the warning that "HOSTNAME is not supported for OCI image format".

```
[student@workstation nexus3]$ sudo podman build -t nexus3 .
...output omitted...
STEP 33: COMMIT nexus3
[student@workstation nexus3]$ sudo podman login -u ${RHT_OCP4_QUAY_USER} quay.io
Password:
Login Succeeded!
[student@workstation nexus3]$ sudo skopeo copy \
> containers-storage:localhost/nexus3 \
> docker://quay.io/${RHT_OCP4_QUAY_USER}/nexus3
...output omitted...
Writing manifest to image destination
Storing signatures
```

**Important**

The build process is not intended to take longer than 5 minutes to complete. During testing of this exercise, much longer build times were reported.

If your build is not complete after five minutes, type **Ctrl+C** to exit the build process. Use the following command to copy a prebuilt Nexus image directly to your personal Quay.io account, and skip the build process:

```
[student@workstation nexus3]$ sudo skopeo copy \
> docker://quay.io/redhattraining/nexus3:latest \
> docker://quay.io/${RHT_OCP4_QUAY_USER}/nexus3
```

▶ **2.** Complete the template to deploy Nexus as a service.

A partial template is provided to deploy Nexus. Update the template to deploy Nexus as a service.

2.1. Create a copy of the starter template to edit, and exit the **~/D0288-apps/nexus3** folder.

```
[student@workstation nexus3]$ cp ~/D0288/labs/nexus-service/nexus-template.yaml \
> ~/nexus-template.yaml
[student@workstation nexus3]$ cd ~
[student@workstation ~]$
```

2.2. Inspect the **~/nexus-template.yaml** file to verify that it uses the container image built previously.

```
[student@workstation ~]$ grep -A1 "kind: DockerImage" ~/nexus-template.yaml
  kind: DockerImage
    name: quay.io/youruser/nexus3:latest
```

2.3. Inspect the **~/nexus-template.yaml** file to verify that it defines resource request and resource limits for the application pod:

```
[student@workstation ~]$ grep -B1 -A5 limits: ~/nexus-template.yaml
  resources:
    limits:
      cpu: "1"
      memory: 2Gi
    requests:
      cpu: 500m
      memory: 256Mi
```

2.4. Open the **~/nexus-template.yaml** file with a text editor, and then override the **INSTALL4J_ADD_VM_PARAMS** environment variable to not include any JVM heap sizing configuration, and to define only the Java system properties required by the application.

```

...output omitted...
- apiVersion: v1
kind: DeploymentConfig
...output omitted...
- env:
  - name: INSTALL4J_ADD_VM_PARAMS
    value: -Djava.util.prefs.userRoot=/nexus-data/javaprefs
...output omitted...

```

- 2.5. The Nexus application contains a **/service/metrics/healthcheck** endpoint that verifies the application is alive. Access to the endpoint requires authentication. When the application starts, the **admin** user password is stored in the **/nexus-data/admin.password** file.

Review the liveness probe in the deployment configuration resource. Open the **~/nexus-template.yaml** file with a text editor and configure the probe to start 120 seconds after the container starts. Also, configure the probe for a maximum waiting execution time of 30 seconds.

```

...output omitted...
- apiVersion: v1
kind: DeploymentConfig
...output omitted...
livenessProbe:
  exec:
    command:
      - /bin/sh
      - "-c"
      - '>
        curl -siu admin:$(cat /nexus-data/admin.password)
        http://localhost:8081/service/metrics/healthcheck |
        grep healthy | grep true
      ...output omitted...
    initialDelaySeconds: 120
    ...output omitted...
    timeoutSeconds: 30
...output omitted...

```

- 2.6. The Nexus application contains a **/service/metrics/ping** endpoint, which verifies that the application is ready. Access to this endpoint also requires authentication.

Review the readiness probe in the deployment configuration resource. Open the **~/nexus-template.yaml** file with a text editor and configure the probe to start 120 seconds after the container starts. Also, configure the probe for a maximum waiting execution time of 30 seconds.

```

...output omitted...
- apiVersion: v1
kind: DeploymentConfig
...output omitted...
readinessProbe:
  exec:
    command:

```

```
- /bin/sh
- "-c"
- >
curl -siu admin:$(cat /nexus-data/admin.password)
http://localhost:8081/service/metrics/ping |
grep pong
...output omitted...
initialDelaySeconds: 120
...output omitted...
timeoutSeconds: 30
...output omitted...
```

- 2.7. In the deployment configuration resource, configure the volume mount to use the **nexus-data** volume at a mount path of **/nexus-data**:

```
...output omitted...
- apiVersion: v1
kind: DeploymentConfig
...output omitted...
volumeMounts:
- mountPath: /nexus-data
  name: nexus-data
...output omitted...
```

- 2.8. In the deployment configuration resource, name the volume **nexus-data**, and then configure the volume to use the **nexus-data-pvc** persistent volume claim:

```
...output omitted...
- apiVersion: v1
kind: DeploymentConfig
...output omitted...
volumes:
- name: nexus-data
  persistentVolumeClaim:
    claimName: nexus-data-pvc
...output omitted...
```

- 2.9. A persistent volume claim is required to persist Nexus data. In the persistent volume claim resource, name the persistent volume claim **nexus-data-pvc** by updating the **metadata.name** attribute:

```
...output omitted...
- apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  labels:
    app: ${NEXUS_SERVICE_NAME}
  name: nexus-data-pvc
...output omitted...
```

- 2.10. The template file is now complete. Save your edits.

To verify the changes made during this step, compare your template file to the solution template file at **~/DO288/solutions/nexus-service/nexus-**

template.yaml. If you are uncertain about your edits, you can copy the solution file and continue to the next step.

- 3. Create a new project. Add a secret that allows any project user to pull the Nexus container image from Quay.io.

- 3.1. Log in to OpenShift using your developer user account:

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful
...output omitted...
```

- 3.2. Create a new project for the application:

```
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-nexus-service
Now using project "yourname-nexus-service" on server
"https://api.cluster.domain.example.com:6443".
...output omitted...
```

- 3.3. Use Podman without the **sudo** command to log in to Quay.io.

```
[student@workstation ~]$ podman login -u ${RHT_OCP4_QUAY_USER} quay.io
Password:
Login Succeeded!
```

- 3.4. Create a secret to access your Quay.io personal account.

```
[student@workstation ~]$ oc create secret generic quayio \
> --from-file .dockerconfigjson=${XDG_RUNTIME_DIR}/containers/auth.json \
> --type kubernetes.io/dockerconfigjson
secret/quayio created
```

- 3.5. Link the secret to the **default** service account.

```
[student@workstation ~]$ oc secrets link default quayio --for pull
```

- 4. Create a new application.

- 4.1. Create a new application called **nexus3** from the template file. Pass as the **HOSTNAME** parameter a value that is based on your developer user name and the wildcard domain of your OpenShift cluster.

You can either copy the complete command, or execute it directly from **/home/student/D0288/labs/nexus-service/oc-new-app.sh**.

```
[student@workstation ~]$ oc new-app --as-deployment-config --name nexus3 \
> -f ~/nexus-template.yaml \
> -p HOSTNAME=nexus-${RHT_OCP4_DEV_USER}.${RHT_OCP4_WILDCARD_DOMAIN}
...output omitted...
--> Creating resources ...
imagestream.image.openshift.io "nexus3" created
```

```
deploymentconfig.apps.openshift.io "nexus3" created
service "nexus3" created
route.route.openshift.io "nexus3" created
persistentvolumeclaim "nexus-data-pvc" created
--> Success
...output omitted...
```

- 4.2. Wait until the application pod is running, but not ready. Follow the pod logs to observe the lengthy initialization procedure of the Nexus server. When you see the "Started" message, you can stop following the log.

```
[student@workstation ~]$ oc get pods
NAME           READY   STATUS    RESTARTS   AGE
nexus3-1-deploy 0/1     Completed  0          4m34s
nexus3-1-kfwwh  0/1     Running   0          1m25s
[student@workstation ~]$ oc logs -f nexus3-1-kfwwh
...output omitted...
-----
Started Sonatype Nexus OSS 3.18.0-01
-----
...output omitted...
```

Press **Ctrl+C** to exit the **oc logs** command.

- 4.3. Wait for the application to be ready and running. OpenShift may take a few seconds to get a healthy state from the application health probes.

```
[student@workstation ~]$ oc get pods
NAME           READY   STATUS    RESTARTS   AGE
nexus3-1-deploy 0/1     Completed  0          4m34s
nexus3-1-kfwwh  1/1     Running   0          4m25s
```

▶ 5. Test the Nexus application.

- 5.1. Retrieve the route for the Nexus application:

```
[student@workstation ~]$ oc get route
NAME      HOST/PORT
nexus3   nexus-yourname.apps.cluster.domain.example.com ...
```

- 5.2. Open a web browser and navigate to application route. The page displays the Nexus application.

If you wish to log in as the **admin** user, you must obtain the password from the **/nexus-data/admin.password** file in the container. It is not necessary to log in as part of this test.

- 6. Delete the project in OpenShift, as well as the container and image in the external container registry. Because Quay.io allows recovering old container images, you must also delete your repository on Quay.io.

6.1. Delete the OpenShift project:

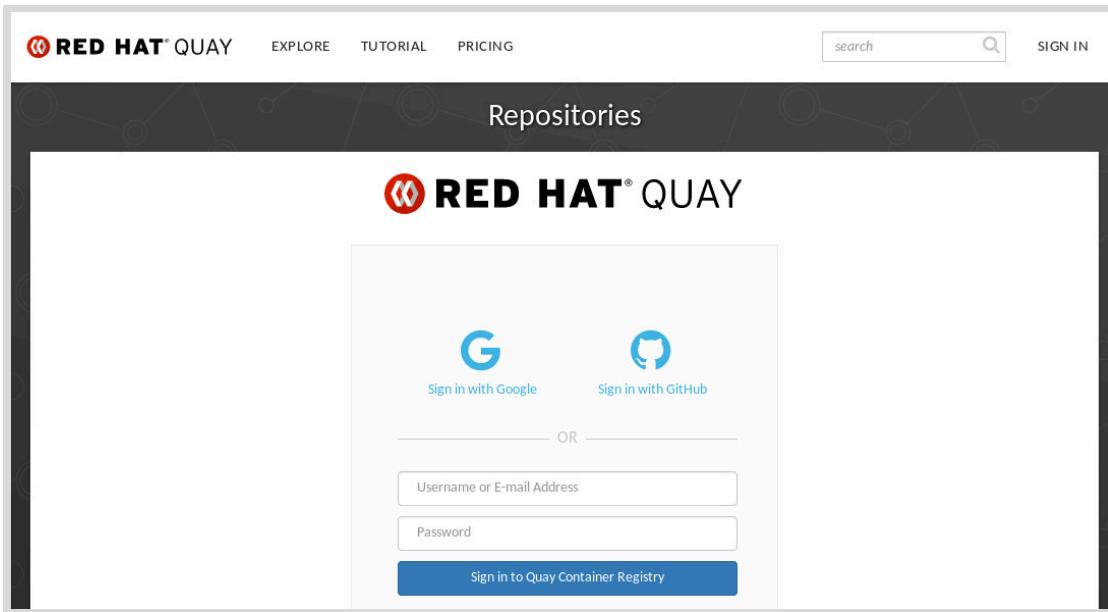
```
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-nexus-service
project.project.openshift.io "youruser.nexus-service" deleted
```

6.2. Delete the container image from the external registry:

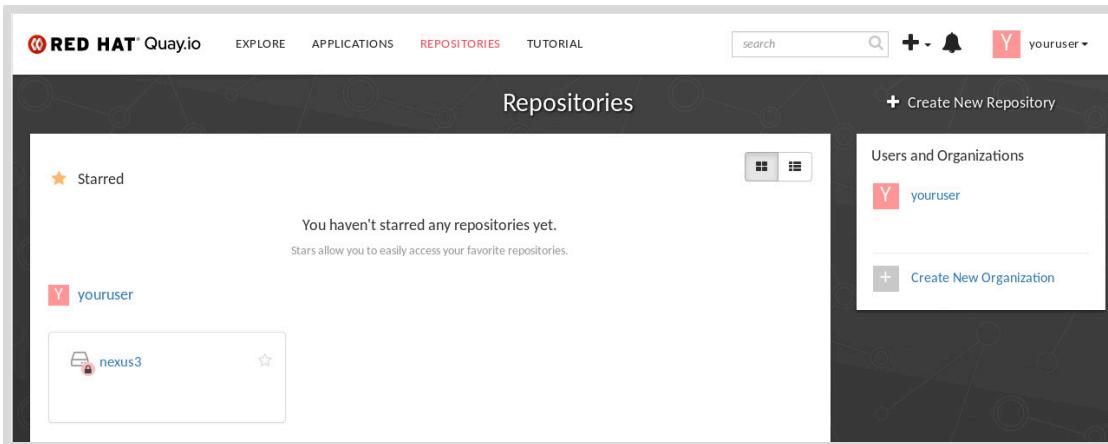
```
[student@workstation ~]$ skopeo delete \
> docker://quay.io/${RHT_OCP4_QUAY_USER}/nexus3
```

6.3. Log in to Quay.io using your personal free account.

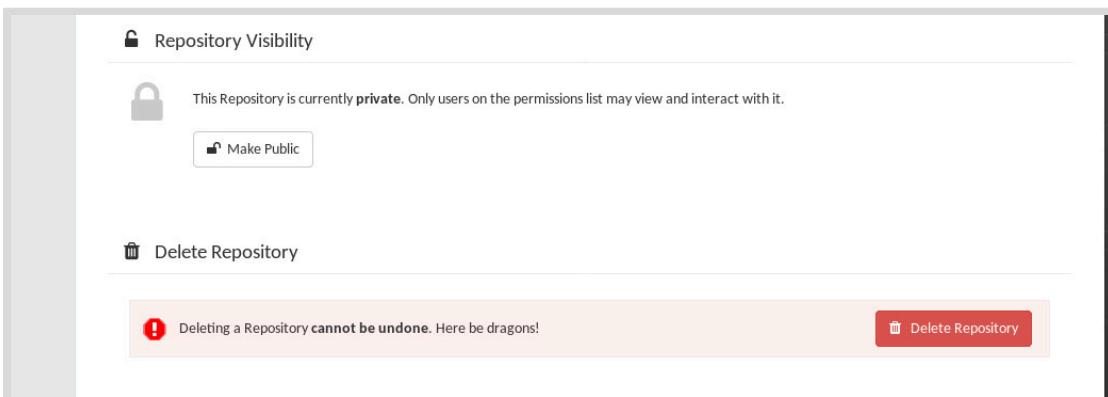
Navigate to <https://quay.io>, and then click **Sign In** to provide your user credentials. Click **Sign in to Quay Container Registry** to log in to Quay.io.



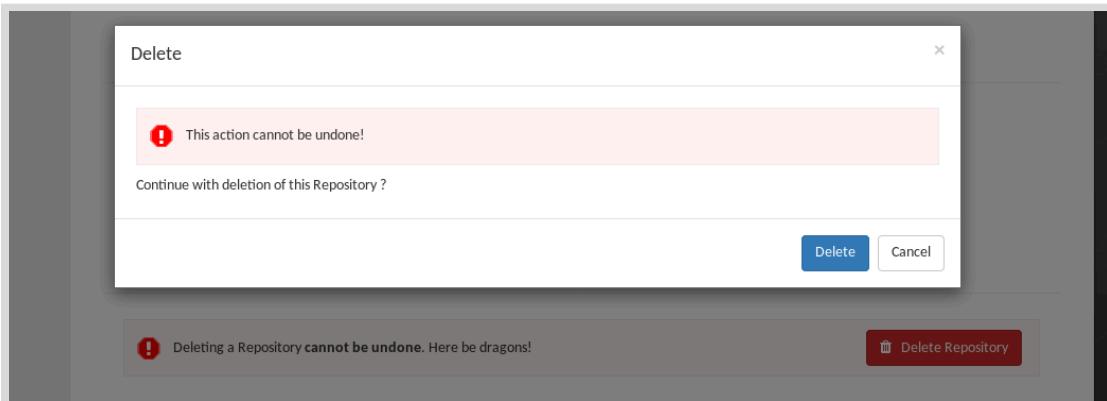
- 6.4. On the Quay.io main menu, click **Repositories** and look for **nexus**. The lock icon indicates that it is a private repository that requires authentication for both pulls and pushes. Click **nexus3** to display the **Repository Activity** page.



- 6.5. On the **Repository Activity** page for the **nexus3** repository, scroll down and then click the gear icon to display the **Settings** tab. On the **Settings** tab, scroll down and click **Delete Repository**.



- 6.6. In the **Delete** dialog box, click **Delete** to confirm that you want to delete the **nexus3** repository. After a few moments you are returned to the **Repositories** page. Sign out of Quay.io.



Finish

On **workstation**, run the **lab nexus-service finish** command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises. The finish action releases this project and its resources.

```
[student@workstation ~]$ lab nexus-service finish
```

This concludes the guided exercise.

Deploying Applications with Red Hat OpenShift Application Runtimes

Objectives

After completing this section, you should be able to deploy an application using a Red Hat OpenShift Application Runtime.

Cloud Native Application Development

The Cloud Native Computing Foundation (CNCF) defines cloud native as:

Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach.

These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil.

The CNCF identifies Kubernetes as a cloud native technology, and established a certification standard for Kubernetes distributions. When you develop and deploy applications on a certified Kubernetes distribution, you avoid platform lock-in because your work loads are portable and operate with other certified distributions. To remain certified, distributions must ensure timely updates to maintain feature parity with the open source Kubernetes project.

OpenShift Container Platform is a CNCF-certified Kubernetes distribution. An application that is deployed on OpenShift, and that is designed to use the services provided by the platform, is considered a cloud native application.

Red Hat OpenShift Application Runtimes

Red Hat OpenShift Application Runtimes (RHOAR) is a collection of prebuilt, containerized runtime foundations for building cloud-native applications on Red Hat OpenShift. RHOAR provides a Red Hat optimized and supported approach for developing microservices applications that target OpenShift as the deployment platform.

RHOAR supports multiple runtimes, languages, frameworks, and architectures. It offers the choice and flexibility to pick the right frameworks and runtimes for the right job. Applications developed with RHOAR can run on any infrastructure where Red Hat OpenShift Container Platform can run, offering freedom from vendor lock-in.

RHOAR provides:

- Access to Red Hat built and supported binaries for selected microservices development frameworks and runtimes.
- Access to Red Hat built and supported binaries for integration modules that replace or enhance a framework's implementation of a microservices pattern to use OpenShift features.

- Developer support for writing applications using selected microservice development frameworks, runtimes, and integration modules, as well as integration with selected external services, such as database servers.
- Production support for deploying applications using selected microservice development frameworks, runtimes, and integration modules, as well as integrations on a supported OpenShift cluster.

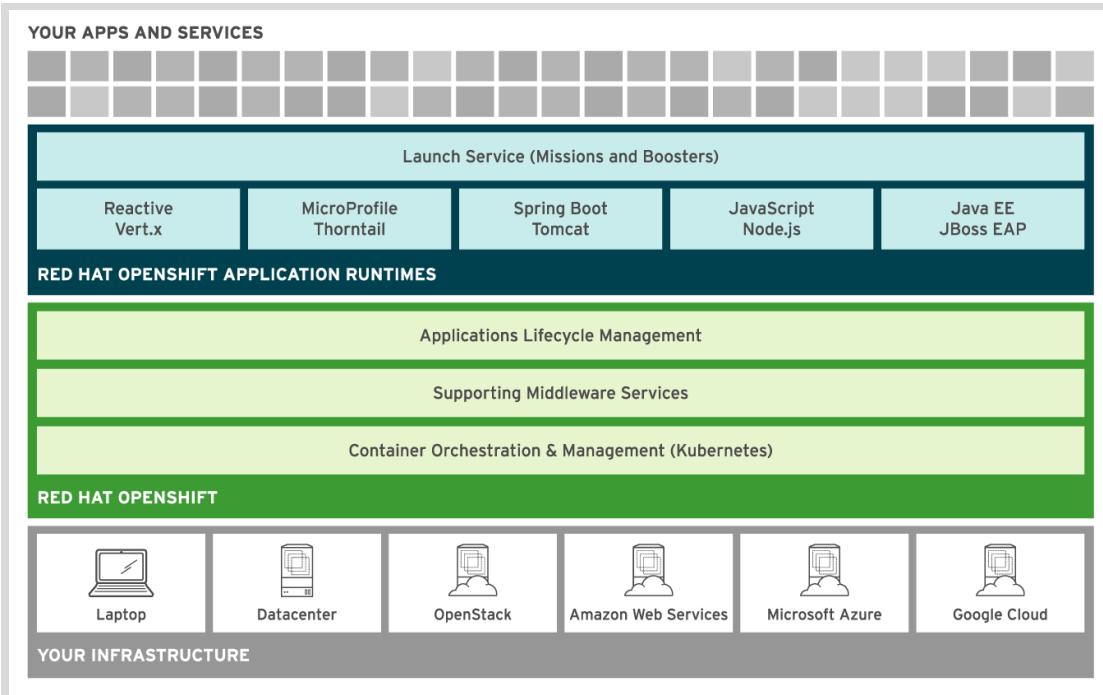


Figure 9.6: Red Hat OpenShift Application Runtimes architecture

Red Hat supports the following popular development frameworks and runtimes as part of RHOAR:

Thorntail

An implementation of the MicroProfile standard that builds cloud-native applications using both existing and new Java EE APIs.

Eclipse Vert.x

A reactive, low-latency development framework based on asynchronous I/O and event streams.

Spring Boot

A cloud-native development framework based on the popular Spring Framework and auto configuration.

Node.js

A JavaScript runtime that provides an I/O model based on events and non blocking operations, which enables you to write high performance applications that are both lightweight and efficient.



Note

For more details on RHOAR, see Red Hat OpenShift Application Runtimes Overview
[<https://developers.redhat.com/products/rhoar/overview/>]

Fabric8 Maven Plug-in

The Fabric8 Maven Plug-in is one of the components of the open source Fabric8 project. Fabric8 is a development platform that supports the full software development life cycle of cloud native applications.

You can use the Fabric8 Maven plug-in to deploy Java applications to an OpenShift cluster using a binary build input source. This means that the plug-in compiles and packages the application, generates the container image with the packaged artifact, and creates OpenShift resources to deploy the application on OpenShift.

Plug-in Configuration

To use the Fabric8 Maven plug-in with a project, update the project's **pom.xml** Maven configuration file to enable and configure the plug-in. You must add a **plugin** entry to the **plugins** listing in the **build** section of the **pom.xml**. The configuration that follows is a minimal configuration to enable the Fabric8 Maven plug-in for a Java application:

```
<build>
  <plugins>
    <plugin>
      <groupId>io.fabric8</groupId> ①
      <artifactId>fabric8-maven-plugin</artifactId> ②
      <version>{fabric8.version}<version> ③
      <executions>
        <execution>
          <goals> ④
            <goal>resource</goal>
            <goal>build</goal>
          </goals>
        </execution>
      </executions>
      <!-- additional configuration here --> ⑤
    </plugin>
  </plugins>
</build>
```

- ① ② Specifies the Fabric8 Maven plug-in group and artifact ID. Maven uses these values to locate and download the plug-in.
- ③ Determines the version of the plug-in that Maven downloads and uses. You can use a community or Red Hat version of the plug-in.

In this example, the value of the **version** tag is **{fabric8.version}**. The **{}** characters indicate that **fabric8.version** is a Maven property. You can find the value of the **fabric8.version** property in the **properties** section of the **pom.xml** file.

- ④ Configures the plug-in to run the resource and the build goals with the **mvn install** command.
- ⑤ Other fine-grained configuration options exist for the Fabric8 Maven plug-in. You can use the options to customize, among other things, configuration of OpenShift resources needed for your application. If no additional options are provided, the Fabric8 Maven plug-in creates a set of simple OpenShift resources for your application.

Alternatively, you can use Fabric8 YAML fragments to customize OpenShift resources for your application. In this course, you use Fabric8 YAML fragments to create customized OpenShift resources, which are discussed later.

Add this XML segment to your project's **pom.xml** file to add the Fabric Maven plug-in. Refer to the References at the end of this lecture for more details about the Fabric8 Maven plug-in configuration.

Plug-in Goals

A Maven plug-in goal represents a well-defined task in the software development life cycle process. You execute a Maven plug-in goal with the **mvn** command:

```
[user@host sample-app]$ mvn <plug-in goal name>
```

The Fabric8 Maven plug-in provides a set of goals to deal with the development of cloud native Java applications:

fabric8:resource

Creates Kubernetes and OpenShift resource descriptors. The plug-in stores all generated descriptors in the project's **target/classes/META-INF/fabric8** subdirectory.

fabric8:build

Compiles and packages the Java application to create a binary artifact, and then uses the artifact to build the associated application container image.

The plug-in uses *generators* to auto-detect build parameters that are needed to compile and package the application. Of particular importance, generators identify an appropriate builder image to use for the application. For generic Java applications, the default builder image is `fabric8/s2i-java` (community) or `jboss-fuse-6/fis-java-openshift` (Red Hat), depending on the version of the Fabric8 Maven plug-in.

If the plug-in detects access to an OpenShift cluster, it initiates an OpenShift binary build. The plug-in supports both Source-To-Image and Docker binary builds. By default, the plug-in executes a Source-To-Image build strategy.

For OpenShift builds, the plug-in creates an application build configuration and image stream resource on the OpenShift cluster. For both Source and Docker build strategies, the plug-in updates the image stream with the new container image.

fabric8:apply

Applies the generated resources to a connected Kubernetes or OpenShift cluster.

fabric8:resource-apply

Runs the **fabric8:resource** and **fabric8:apply** goals.

fabric8:deploy

Runs the **fabric8:resource**, **fabric8:build**, and **fabric8:apply** goals.

fabric8:undeploy

Removes resources from the OpenShift cluster.

The Fabric8 Maven plug-in supports other goals that are beyond the scope of this course. Refer to the References at the end of this lecture for more details about the goals.

Customizing OpenShift Resources

With minimal project configuration, the Fabric8 Maven plug-in generates a set of OpenShift resources to support the deployment of your application on OpenShift. In some scenarios, the generated OpenShift resources may not be sufficient for OpenShift deployment. Customize the generated resources in one of two ways: add OpenShift resource YAML fragments to the project's **src/main/fabric8** subdirectory, or add additional configuration to the project **pom.xml** file. In this course, you use YAML resource fragments to customize the OpenShift configuration for a project.

If you add only the minimal Fabric8 configuration to your project, then the plug-in creates a service and deployment configuration resource for your application. If the associated container image exposes a port, then the plug-in also creates a route resource to expose the service. The plug-in writes each resource definition to a file in the **target/classes/META-INF/fabric8/openshift** directory. All of these resource definitions are combined into a **openshift.yml** file, in the **target/classes/META-INF/fabric8** subdirectory for the project.

The plug-in also processes YAML fragment files in the **src/main/fabric8** directory. The plug-in uses the content in each fragment to override the corresponding default resource definition. The content of each file mimics the structure of an OpenShift resource, but omits any information that is unchanged. These files are intended to be small and compact, representing only the configuration that must change from the default resource configuration.

Each fragment file follows a file naming convention. The plug-in uses the fragment file name to identify the OpenShift resource to override. Fragment file names follow the pattern **[name]-type.yml**.

The **name** is optional, and represents the name of the resource. If a name is not provided, the plug-in uses the application name as the name of the resource.

The **type** corresponds to the kind of OpenShift resource that this fragment modifies. The type value must be one of the following:

Kind	Filename
Service	svc, service
Route	route
Deployment	deployment
DeploymentConfig	dc, deploymentconfig
ConfigMap	cm, configmap
Secret	secret

For example, consider the content of **src/main/fabric8/route.yml** below:

```
spec:
  host: app.alternate.com
```

This fragment changes the default host name for the application route to `app.alternate.com`.

Red Hat Middleware for OpenShift

Red Hat provides a set of middleware container images that allow you to deploy Red Hat middleware applications to OpenShift.

For example, use the Red Hat Java S2I for OpenShift container image to deploy an application packaged as a *fat JAR file*. A fat jar archive is a runnable application that contains all classes and resources packaged together in the same jar file, including runtime dependencies.

The Red Hat Container Catalog includes container images for the following products:

- Red Hat JBoss EAP for OpenShift: Enables building and deploying an application using the Red Hat JBoss Enterprise Application Platform.
- Red Hat AMQ for OpenShift: Enables deploying an AMQ message broker that is based on Apache ActiveMQ.
- Red Hat Fuse for OpenShift: Provides a set of tools and container images that enable development, deployment, and management of integration microservices within OpenShift.
- Red Hat OpenShift Service Mesh: Enables operations teams and developers to provide traffic monitoring, access control, discovery, security, and resiliency to a group of services.
- Red Hat 3scale API Management: Share, secure, distribute, control, and monetize APIs.

Builder images for other Red Hat middleware products are also available. Refer to the References at the end of this lecture for more information on these and other container images.



References

Further information about Cloud Native Computing Foundation is available at
<https://www.cncf.io/>

Further information about RHOAR is available at
<https://developers.redhat.com/products/rhoar/overview/>

The open source documentation for the Fabric8 Maven plug-in is available at
<https://maven.fabric8.io/>

Further information about the Fabric8 Maven plug-in is available in the *Fabric8 Maven Plug-In* appendix of the *Fuse on OpenShift* guide from the Red Hat JBoss Fuse 7.3 product documentation at
https://access.redhat.com/documentation/en-us/red_hat_fuse/7.3/html-single/fuse_on_openshift_guide/index#fabric8-maven-plugin

Further information about configuring Maven to use supported dependencies is available in the *Prepare Your Development Environment* section of the *Get Started for Developers* chapter of the *Fuse on OpenShift* guide from the Red Hat Fuse 7.3 product documentation at
https://access.redhat.com/documentation/en-us/red_hat_fuse/7.3/html-single/fuse_on_openshift_guide/index#get-started-prepare-dev-env

► Guided Exercise

Deploying an Application with Red Hat OpenShift Application Runtimes

In this exercise, you will use the Fabric8 Maven plug-in to deploy a microservice to the OpenShift cluster.

Outcomes

You should be able to:

- Configure a custom OpenShift deployment configuration resource using the Fabric8 Maven plug-in.
- Configure an OpenShift configuration map resource using the Fabric8 Maven plug-in.
- Deploy a microservice using the Fabric8 Maven plug-in.

Before You Begin

To perform this exercise, ensure that you have access to:

- A running OpenShift cluster.
- The microservice application in the Git repository (**micro-java**).

Run the following command on **workstation** to validate the prerequisites and to download the files required to complete this exercise:

```
[student@workstation ~]$ lab micro-java start
```

► 1. Inspect the Java source code of the **micro-java** sample application.

- Enter the **micro-java** subdirectory of your local clone of the **D0288-apps** Git repository. Checkout the **master** branch of the course repository to ensure that you start this exercise from a known good state:

```
[student@workstation ~]$ cd ~/D0288-apps/micro-java
[student@workstation micro-java]$ git checkout master
...output omitted...
```

- Create a new branch to save any changes that you make during this exercise:

```
[student@workstation micro-java]$ git checkout -b micro-config
Switched to a new branch 'micro-config'
[student@workstation micro-java]$ git push -u origin micro-config
...output omitted...
* [new branch]      micro-config -> micro-config
Branch micro-config set up to track remote branch micro-config from origin.
```

13. Review the **JaxRsActivator.java** source code file in the **src/main/java/com/redhat/training/openshift/hello** subdirectory of the micro-java source code:

```
...output omitted...
@ApplicationPath("/api")
public class JaxRsActivator extends Application {
}
...output omitted...
```

The source code specifies that the application is accessed at a path of **/api**.

14. Review the **HelloResource.java** source code file in the **src/main/java/com/redhat/training/openshift/hello** subdirectory of the micro-java source code:

```
package com.redhat.training.openshift.hello;

import javax.inject.Inject;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;

@Path("/")
public class HelloResource① {

    @GET
    @Path("/hello")②
    @Produces("text/plain")
    public String hello() {
        String hostname = System.getenv().getOrDefault("HOSTNAME", "unknown");③
        String message = System.getenv().getOrDefault("APP_MSG", null);④
        String response = "";

        if (message == null) {
            response = "Hello world from host "+hostname+"\n";⑤
        } else {
            response = "Hello world from host ["+hostname+"].\n";
            response += "Message received = "+message+"\n";⑥
        }
        return response;
    }
}
```

- ① This class defines a REST resource for the application.
- ② Because you access the application at **/api**, you access this application resource at **/api/hello**.
- ③ ④ The application uses the **HOSTNAME** and **APP_MSG** environment variables in the application.
- ⑤ If the **APP_MSG** environment variable is not defined, the application resource responds with a message containing the host name of the server on which the application is running.
- ⑥ If the **APP_MSG** environment variable is defined, the response message contains the value of both the **HOSTNAME** and **APP_MSG** environment variables.

► 2. Review the configuration of the Fabric8 Maven plug-in in the project **pom.xml** file:

2.1. Review the project-level attributes of the **pom.xml**, found near the top of the file:

```
<?xml version="1.0" encoding="UTF-8"?>
...output omitted...
<groupId>com.redhat.training.openshift</groupId>
<artifactId>hello</artifactId>1
<version>1.0</version>2
...output omitted...
<properties>
...output omitted...
    <!-- Thorntail dependency versions -->
    <version.thorntail>2.4.0.Final</version.thorntail>3

    <!-- other plugin versions -->
...output omitted...
    <version.fabric8.plugin>4.1.0</version.fabric8.plugin>4
...output omitted...
</properties>
...output omitted...
```

1 **2** The name and version of the application. The Fabric8 Maven plug-in creates an image stream tag resource from these values, **hello:1.0**.

3 **4** These version properties set the version of the Thorntail application runtime and Fabric8 Maven plug-in, respectively. These properties are referenced further down in the **pom.xml** file where the Thorntail and Fabric Maven plug-ins are configured.

2.2. Review the list of plug-ins in the **build** section of the **pom.xml**, near the end of the file:

```
...output omitted...
<build>
...output omitted...
    <plugins>
...output omitted...
        <plugin>1
            <groupId>io.thorntail</groupId>
            <artifactId>thorntail-maven-plugin</artifactId>
            <version>${version.thorntail}</version>
...output omitted...
        </plugin>
        <plugin>2
            <groupId>io.fabric8</groupId>
            <artifactId>fabric8-maven-plugin</artifactId>
            <version>${version.fabric8.plugin}</version>
            <executions>
                <execution>
                    <id>fmp</id>
                    <goals>
                        <goal>resource</goal>
                        <goal>build</goal>
```

```

        </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
...output omitted...

```

① ② Two plug-ins are configured for the project.

- ① The Thorntail Maven plug-in packages the source code as a Thorntail application. The plug-in version is determined by the value of the **version.thorntail** property, which is set in the **properties** section of the **pom.xml** file.
- ② The Fabric8 Maven plug-in creates OpenShift resources for the application, and builds a container image from the Thorntail-packaged code. The plug-in version is determined by the value of the **version.fabric8.plugin** property, which is also set in the **properties** section of the **pom.xml** file.

► 3. Generate OpenShift resources for the application.

- 3.1. From the project directory, execute the **mvn clean** command to remove any build artifacts from previous builds. Then, generate OpenShift resources with the **fabric8:resource** Maven goal:

```

[student@workstation micro-java]$ mvn clean
...output omitted...
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
...output omitted...
[student@workstation micro-java]$ mvn fabric8:resource
[INFO] Scanning for projects...
...output omitted...
[INFO] --- fabric8-maven-plugin:4.1.0:resource (default-cli) @ hello ---
...output omitted...
[INFO] F8: fmp-controller: Adding a default DeploymentConfig
[INFO] F8: fmp-service: Adding a default service 'hello' with ports [8080]
[INFO] F8: fmp-revision-history: Adding revision history limit to 2
[INFO] F8: validating .../fabric8/openshift/hello-service.yml ...❶
[INFO] F8: validating .../fabric8/openshift/hello-deploymentconfig.yml ...
[INFO] F8: validating .../fabric8/openshift/hello-route.yml resource
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
...output omitted...

```

❶ The **fabric8:resource** Maven goal creates three YAML files. Collectively, these files define a service, deployment configuration, and route resource for the sample application. These files are located in the project **target/classes/META-INF/fabric8/openshift** subdirectory.

- 3.2. Review the generated **hello-deploymentconfig.yml** file in the **target/classes/META-INF/fabric8/openshift** directory.

```
---
apiVersion: apps.openshift.io/v1
kind: DeploymentConfig
metadata:
...output omitted...
name: hello
spec:
replicas: 1
...output omitted...
template:
spec:
containers:
- env:
- name: KUBERNETES_NAMESPACE
valueFrom:
fieldRef:
fieldPath: metadata.namespace
image: hello:1.0
...output omitted...
```

The file defines a deployment configuration named **hello** that deploys a single container using an image from the **hello:1.0** image stream tag.

- 3.3. Review the generated **hello-service.yml** file in the **target/classes/META-INF/fabric8/openshift** directory.

```
---
apiVersion: v1
kind: Service
metadata:
...output omitted...
name: hello
spec:
ports:
- name: http
port: 8080
protocol: TCP
targetPort: 8080
selector:
app: hello
provider: fabric8
group: com.redhat.training.openshift
```

This file defines a service named **hello** for the application pods defined in the **hello** deployment configuration.

- 3.4. Review the generated **hello-route.yml** file in the **target/classes/META-INF/fabric8/openshift** directory.

```
---
apiVersion: route.openshift.io/v1
kind: Route
metadata:
...output omitted...
```

```

name: hello
spec:
  port:
    targetPort: 8080
  to:
    kind: Service
    name: hello

```

This file defines a route resource named **hello** that exposes the **hello** service.

- 3.5. The **fabric8:resource** Maven goal also generates a combined list of all resources. Review the generated **openshift.yml** file in the project **target/classes/META-INF/fabric8** subdirectory:

```

...output omitted...
kind: "List"
...output omitted...
  kind: "Service"
...output omitted...
  kind: "DeploymentConfig"
...output omitted...
  kind: "Route"
...output omitted...

```

► 4. Build and deploy the application.

- 4.1. Load your classroom environment configuration.

Run the following command to load the environment variables created in the first guided exercise:

```
[student@workstation micro-java]$ source /usr/local/etc/ocp4.config
```

- 4.2. Log in to OpenShift using your developer username:

```

[student@workstation micro-java]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful.
...output omitted...

```

- 4.3. Create a new project for the application. Prefix the project name with your developer username.

```

[student@workstation micro-java]$ oc new-project ${RHT_OCP4_DEV_USER}-micro-java
Now using project "youruser-micro-java" on server ...
...output omitted...

```

- 4.4. Build and deploy the application with the Fabric8 Maven plug-in: Monitor the output to verify that the build runs in OpenShift mode, and that OpenShift resources were created.

```
[student@workstation micro-java]$ mvn fabric8:deploy
[INFO] Scanning for projects...
...output omitted...
[INFO] --- fabric8-maven-plugin:4.1.0:resource (fmp) @ hello ---
...output omitted...
[INFO] --- fabric8-maven-plugin:4.1.0:build (fmp) @ hello --- ①
[INFO] F8: Running in OpenShift mode
[INFO] F8: Using OpenShift build with strategy S2I
[INFO] F8: Running generator thorntail-v2
...output omitted...
[INFO] --- fabric8-maven-plugin:4.1.0:deploy (default-cli) @ hello --- ②
[INFO] F8: Using OpenShift ... with manifest .../openshift.yml
[INFO] OpenShift platform detected
[INFO] F8: Using project: youruser-micro-java ③
[INFO] F8: Creating a Service from openshift.yml ④ ... name hello
...output omitted...
[INFO] F8: Creating a DeploymentConfig from openshift.yml name hello
...output omitted...
[INFO] F8: Creating Route youruser-micro-java:hello host: null
[INFO] F8: HINT: Use the command `oc get pods -w` to watch your pods start up
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
...output omitted...
```

- ① The build phase begins. In this phase, the plug-in uses a Source-to-Image binary build to create a container image for the application.
- ② The deployment phase begins.
- ③ The plug-in detects the active OpenShift project. All resources are created in this project.
- ④ The plug-in uses the **openshift.yml** to create an OpenShift service, deployment configuration, and route resource.

► 5. Review the resources created by the Fabric8 Maven plug-in.

```
[student@workstation micro-java]$ oc status
In project youruser-micro-java on server ...

http://hello-youruser-micro-java... to pod port 8080 (svc/hello)
dc/hello deploys istag/hello:1.0 <- bc/hello-s2i source builds ...
  deployment #1 deployed 2 minutes ago - 1 pod
...output omitted...
```

A route, service, and deployment configuration resource exist in your project, each with a name of **hello**. A build configuration and image stream tag resource also exist in the project.

► 6. Test the application.

6.1. Wait for the application to be ready and running:

```
[student@workstation micro-java]$ oc get pods
NAME          READY   STATUS    RESTARTS   AGE
hello-1-5pw6q 1/1     Running   1          14m
hello-1-deploy 0/1     Completed  0          14m
hello-s2i-1-build 0/1     Completed  0          16m
```

- 6.2. Test external access to the application. Remember to add **/api/hello** to the end of the route URL to access the **HelloResource** REST resource for the application.

```
[student@workstation D0288-apps]$ ROUTE_URL=$(oc get route \
> hello --template='{{.spec.host}}')
[student@workstation D0288-apps]$ curl ${ROUTE_URL}/api/hello
Hello world from host hello-1-5pw6q
```

Because the application deployment configuration does not define a value for the **ADD_MSG** environment variable, the output only contains the container host name.

- ▶ 7. Update the project to deploy application pods with a value for the **APP_MSG** environment variable.

- 7.1. Review the **cm.yaml** fragment file in the **~/D0288/labs/micro-java** directory:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: env-config
data:
  APP_MSG: sample external configuration
```

The above YAML fragment defines a configuration map resource with a name of **env-config**. This configuration map defines a **APP_MSG** variable with a value of **sample external configuration**.

- 7.2. Review the **deployment.yaml** fragment file in the **~/D0288/labs/micro-java** directory:

```
spec:
  template:
    spec:
      containers:
        - envFrom:
          - configMapRef:
              name: env-config
```

The Fabric8 Maven plug-in uses this fragment to update the default deployment configuration it generates. In particular, each deployed container is injected with environment variables from the **env-config** configuration map.

- 7.3. Copy the fragment files to the application **src/main/fabric8** subdirectory.

```
[student@workstation micro-java]$ cp -v ~/DO288/labs/micro-java/*.yml \
> ./src/main/fabric8/
'...labs/micro-java/cm.yml' -> './src/main/fabric8/cm.yml'
'...labs/micro-java/deployment.yml' -> './src/main/fabric8/deployment.yml'
```

7.4. Commit the YAML fragments to the **micro-config** branch:

```
[student@workstation micro-java]$ git add src/main/fabric8/*.yml
[student@workstation micro-java]$ git commit -am "Add YAML fragments."
...output omitted...
```

- 8. Redeploy the application. Verify that a Fabric8 Maven plug-in creates a configuration map resource. Verify that the application responds with the value of the **APP_MSG** variable from the configuration map.

8.1. Redeploy the application with the Fabric8 Maven plug-in:

```
[student@workstation micro-java]$ mvn fabric8:deploy
...output omitted...
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
...output omitted...
```

8.2. Verify the presence of the **env-config** configuration map:

```
[student@workstation micro-java]$ oc get cm/env-config
NAME          DATA   AGE
env-config    1      84m
```

- 8.3. Wait for the new application pods to deploy. When the output of the **oc get pods** indicates that a new deployment is ready and running, continue to the next step:

```
[student@workstation micro-java]$ oc get pods
NAME        READY   STATUS    RESTARTS   AGE
hello-1-deploy  0/1     Completed  0          12m
hello-3-deploy  0/1     Completed  0          117s
hello-3-n2rn2   1/1     Running   0          108s
hello-s2i-1-build  0/1     Completed  0          15m
hello-s2i-2-build  0/1     Completed  0          4m42s
```

8.4. Verify the application response includes the value of the **APP_MSG** variable:

```
[student@workstation micro-java]$ curl ${ROUTE_URL}/api/hello
Hello world from host [hello-3-m9k4j].
Message received = sample external configuration
```

**Note**

If the previous `curl` returns an error HTML page stating that the application is not available, it probably means the application container is still not ready to serve requests. Wait a few seconds and try the same command again.

This kind of issue can be avoided by adding a readiness probe to the deployment configuration.

- ▶ **9.** Clean up: Delete all resources created during this exercise.

```
[student@workstation micro-java]$ oc delete project \
> ${RHT_OCP4_DEV_USER}-micro-java
```

Finish

On **workstation**, run the `lab micro-java finish` script to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises. The finish action releases this project and its resources.

```
[student@workstation ~]$ lab micro-java finish
```

This concludes the guided exercise.

▶ Lab

Building Cloud-Native Applications for OpenShift

Performance Checklist

In this lab, you will use the Fabric8 Maven plug-in to deploy an application to OpenShift that communicates with a database running outside the OpenShift cluster.

Outcomes

You should be able to:

- Create a database service for the application that points to a MariaDB database server outside the OpenShift cluster.
- Configure custom OpenShift resources using Fabric8 YAML fragments.
- Deploy the To Do List back-end application using the Fabric8 Maven plug-in.

Before You Begin

To perform this exercise, you need access to:

- A running OpenShift cluster.
- The todo-api-micro sample application in the Git repository.
- The external MariaDB database server.

Run the following command on **workstation** to validate the prerequisites, populate the database, and download files required to complete this exercise:

```
[student@workstation ~]$ lab todo-migrate start
```

Requirements

The To Do List back-end development team is migrating a Thorntail application to OpenShift. The application source code is located in the **todo-api-micro** subdirectory of the cloned Git repository.

You must configure the application and deploy it to an OpenShift cluster according to the following requirements:

- The project name is **youruser-todo-migrate**. Your developer user must own the project.
- A **tododb** service exists in your OpenShift project that connects to an external database. To obtain the FQDN of the external database, replace **apps** in your OpenShift cluster wildcard domain with **mysql**. For example, if the wildcard domain of your cluster is `apps.cluster.domain.example.com`, then the database domain is `mysql.cluster.domain.example.com`.

- An OpenShift configuration map resource is created as part of the deployment.

The configuration map defines variables that are required to access the external database:

- **DATABASE_USER: todoapp**
- **DATABASE_PASSWORD: redhat123**
- **DATABASE_SVC_HOSTNAME: tododb**
- **DATABASE_NAME: todo**

Use a Fabric8 YAML fragment to create the configuration map resource.



Important

To simplify the lab, you define the database password in the configuration map resource. A better practice is to define the database password in a secret resource.

- A custom OpenShift deployment configuration resource is created as part of the deployment.

Use a Fabric8 YAML fragment to add all variables from the configuration map resource into the application container as environment variables.

- You commit all code changes to the remote Git repository.

To test for a successful deployment, the application **todo/api/items/6** endpoint should return JSON data.

Steps

1. Verify connectivity to the external database server.
2. Create an OpenShift service called **tododb** that connects to the external database instance. The service must be created in the **youruser-todo-migrate** project.
3. Create a **todo-migrate** branch from the **master** branch in your local clone of the **DO288-apps** repository. Change to the **todo-api-micro** project subdirectory.
4. Build and deploy the application. Verify that the application pod fails to deploy because the required environment variables are not defined.
5. Create the YAML fragment that Fabric8 uses to generate the custom configuration map resource that defines the database environment variables.
You may choose to define the required environment variables in a custom configuration map resource, or directly in the deployment configuration.
6. Create the YAML fragment that Fabric8 uses to generate the custom deployment configuration resource.
7. Apply the custom configuration map and deployment configuration resource to the project. Verify that the application deploys without any failures.
Use the external route to test access to the application **todo/api/items/6** resource. A successful response returns JSON data.
8. After application tests succeed, commit and push your code changes to the remote repository.

Evaluation

As the **student** user on the **workstation** machine, use the **lab** command to grade your work. Correct any reported failures and rerun the command until successful.

```
[student@workstation ~]$ lab todo-migrate grade
```

Finish

On **workstation**, run the **lab todo-migrate finish** command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab todo-migrate finish
```

This concludes the lab.

► Solution

Building Cloud-Native Applications for OpenShift

Performance Checklist

In this lab, you will use the Fabric8 Maven plug-in to deploy an application to OpenShift that communicates with a database running outside the OpenShift cluster.

Outcomes

You should be able to:

- Create a database service for the application that points to a MariaDB database server outside the OpenShift cluster.
- Configure custom OpenShift resources using Fabric8 YAML fragments.
- Deploy the To Do List back-end application using the Fabric8 Maven plug-in.

Before You Begin

To perform this exercise, you need access to:

- A running OpenShift cluster.
- The todo-api-micro sample application in the Git repository.
- The external MariaDB database server.

Run the following command on **workstation** to validate the prerequisites, populate the database, and download files required to complete this exercise:

```
[student@workstation ~]$ lab todo-migrate start
```

Requirements

The To Do List back-end development team is migrating a Thorntail application to OpenShift. The application source code is located in the **todo-api-micro** subdirectory of the cloned Git repository.

You must configure the application and deploy it to an OpenShift cluster according to the following requirements:

- The project name is **youruser-todo-migrate**. Your developer user must own the project.
- A **tododb** service exists in your OpenShift project that connects to an external database. To obtain the FQDN of the external database, replace **apps** in your OpenShift cluster wildcard domain with **mysql**. For example, if the wildcard domain of your cluster is `apps.cluster.domain.example.com`, then the database domain is `mysql.cluster.domain.example.com`.
- An OpenShift configuration map resource is created as part of the deployment.

The configuration map defines variables that are required to access the external database:

- **DATABASE_USER: todoapp**
- **DATABASE_PASSWORD: redhat123**
- **DATABASE_SVC_HOSTNAME: tododb**
- **DATABASE_NAME: todo**

Use a Fabric8 YAML fragment to create the configuration map resource.



Important

To simplify the lab, you define the database password in the configuration map resource. A better practice is to define the database password in a secret resource.

- A custom OpenShift deployment configuration resource is created as part of the deployment.

Use a Fabric8 YAML fragment to add all variables from the configuration map resource into the application container as environment variables.

- You commit all code changes to the remote Git repository.

To test for a successful deployment, the application **todo/api/items/6** endpoint should return JSON data.

Steps

1. Verify connectivity to the external database server.

- 1.1. Load your classroom environment configuration.

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

- 1.2. Determine the host name of the external database server.

```
[student@workstation ~]$ MYSQL_DB=$(echo \
> mysql.ocp-$RHT_OCP4_WILDCARD_DOMAIN#"apps.")
```

- 1.3. Connect to the external MySQL database.

```
[student@workstation ~]$ mysql -h${MYSQL_DB} -utodoapp -predhat123 todo
Reading table information for completion of table and column names
...output omitted...
MariaDB [todo]>
```

- 1.4. Exit the MySQL client to return to the shell prompt.

```
MariaDB [todo]> exit
Bye
[student@workstation ~]$
```

2. Create an OpenShift service called **tododb** that connects to the external database instance. The service must be created in the **youruser-todo-migrate** project.

2.1. Log in to OpenShift using your developer user account:

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful
...output omitted...
```

2.2. Create a new project to host the application:

```
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-todo-migrate
```

2.3. Create a service based on an external name:

```
[student@workstation ~]$ oc create service externalname tododb \
> --external-name ${MYSQL_DB}
service "tododb" created
```

2.4. Verify that the **tododb** service exists and shows an external IP, but no cluster IP:

```
[student@workstation ~]$ oc get svc
NAME      TYPE           ...   EXTERNAL-IP          PORT(S)    AGE
tododb   ExternalName   ...   mysql.cluster.domain.example.com <none>    6s
```

3. Create a **todo-migrate** branch from the **master** branch in your local clone of the **D0288-apps** repository. Change to the **todo-api-micro** project subdirectory.

```
[student@workstation ~]$ cd ~/D0288-apps
[student@workstation D0288-apps]$ git checkout master
...output omitted...
[student@workstation D0288-apps]$ git checkout -b todo-migrate
Switched to a new branch 'todo-migrate'
[student@workstation D0288-apps]$ git push -u origin todo-migrate
...output omitted...
[student@workstation D0288-apps]$ cd todo-api-micro
[student@workstation todo-api-micro]$
```

4. Build and deploy the application. Verify that the application pod fails to deploy because the required environment variables are not defined.

4.1. Deploy the application.

```
[student@workstation todo-api-micro]$ mvn fabric8:deploy
```

4.2. After the application deploys, monitor the logs for the application pod.

```
[student@workstation todo-api-micro]$ oc get pods
NAME             READY   STATUS            RESTARTS   AGE
todo-api-1-deploy 0/1     Completed        0          5m43s
todo-api-1-hj5hn 0/1     CrashLoopBackOff  2          90s
todo-api-s2i-1-build 0/1     Completed        0          7m12s
```

The exact **STATUS** of the pod may vary, but the number of **RESTARTS** increases.

```
[student@workstation todo-api-micro]$ oc logs -f todo-api-1-hj5hn
...output omitted...
... ERROR [...] ... ("system-property" => "thorntail.datasources.data-
sources.MySQLDS.connection-url") - failure
description: "WFLYCTL0211: Cannot resolve expression
'jdbc:mysql://${env.DATABASE_SVC_HOSTNAME}:3306/${env.DATABASE_NAME}'"
```

A failure occurs because the **DATABASE_SVC_HOSTNAME** and **DATABASE_NAME** are not defined.

5. Create the YAML fragment that Fabric8 uses to generate the custom configuration map resource that defines the database environment variables.

You may choose to define the required environment variables in a custom configuration map resource, or directly in the deployment configuration.

- 5.1. Create the **src/main/fabric8/cm.yml** file with the following content:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: db-config
data:
  DATABASE_USER: todoapp
  DATABASE_PASSWORD: redhat123
  DATABASE_SVC_HOSTNAME: tododb
  DATABASE_NAME: todo
```

Alternatively, a solution YAML fragment file is available in the **/home/student/D0288/solutions/todo-migrate** directory. You can copy it to the **src/main/fabric8** subdirectory:

```
[student@workstation todo-api-micro]$ cp \
> ~/D0288/solutions/todo-migrate/cm.yml src/main/fabric8
```

6. Create the YAML fragment that Fabric8 uses to generate the custom deployment configuration resource.

- 6.1. Create the **src/main/fabric8/deployment.yml** file with the following content:

```
spec:
  template:
    spec:
      containers:
        - envFrom:
          - configMapRef:
            name: db-config
```

The configuration map reference name must match the name of the configuration map resource.

Alternatively, a solution YAML fragment file is available in the **/home/student/D0288/solutions/todo-migrate** directory. You can copy it to the **src/main/fabric8** subdirectory:

```
[student@workstation todo-api-micro]$ cp \
> ~/D0288/solutions/todo-migrate/deployment.yml src/main/fabric8
```

7. Apply the custom configuration map and deployment configuration resource to the project. Verify that the application deploys without any failures.

Use the external route to test access to the application **todo/api/items/6** resource. A successful response returns JSON data.

- 7.1. Apply the new OpenShift resources to your project.

```
[student@workstation todo-api-micro]$ mvn fabric8:resource-apply
```

- 7.2. Review the resources created by the Fabric8 Maven plug-in.

```
[student@workstation todo-api-micro]$ oc describe dc/todo-api \
> | grep -A1 "Environment Variables"
  Environment Variables from:
    db-config ConfigMap Optional: false
```

The configuration map resource name must match the name of the configuration map resource that contains the database variables.

```
[student@workstation todo-api-micro]$ oc get configmap
NAME           DATA   AGE
db-config       4      5m16s
...output omitted...
[student@workstation todo-api-micro]$ oc get pods
NAME           READY   STATUS    RESTARTS   AGE
...output omitted...
pod/todo-api-3-frhtt   1/1     Running   0          35s
...output omitted...
```

- 7.3. Wait for the application to be ready and running:

```
[student@workstation todo-api-micro]$ oc logs -f todo-api-3-frhtt
...output omitted...
... INFO [org.wildfly.swarm] (main) THORN99999: Thorntail is Ready
```

**Note**

You will find an error in the application logs related to SSL:

```
... ERROR ... Establishing SSL connection without server's identity verification is not recommended. ...
```

This error does not cause functional errors in the application. The error occurs because the remote database uses a self-signed certificate to establish an SSL connection with the application.

In this lab, you can ignore this error.

- 7.4. Verify that the application gets data from the external database.

Use the **curl** command to test that the application **todo/api/items/6** resource returns JSON data.

```
[student@workstation todo-api-micro]$ ROUTE_URL=$(oc get route todo-api \
> --template={{.spec.host}})
[student@workstation todo-api-micro]$ curl -s ${ROUTE_URL}/todo/api/items/6 \
> | python -m json.tool
{
    "description": "Verify that the To Do List application works",
    ...output omitted...
}
```

8. After application tests succeed, commit and push your code changes to the remote repository.

```
[student@workstation todo-api-micro]$ git add src/main/fabric8/*
[student@workstation todo-api-micro]$ git commit -m "add YAML fragments"
...output omitted...
[student@workstation todo-api-micro]$ git push origin todo-migrate
...output omitted...
```

Evaluation

As the **student** user on the **workstation** machine, use the **lab** command to grade your work. Correct any reported failures and rerun the command until successful.

```
[student@workstation ~]$ lab todo-migrate grade
```

Finish

On **workstation**, run the **lab todo-migrate finish** command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab todo-migrate finish
```

This concludes the lab.

Summary

In this chapter, you learned:

- A service name becomes a local DNS host name for all pods inside an OpenShift cluster.
- An external service is created with the **oc create service externalname** command, using the the **external-name** option.
- Red Hat recommends that production deployments define health probes.
- Red Hat provides a set of middleware container images to deploy applications in OpenShift, including applications packaged as fat JAR files.
- The Fabric8 Maven plug-in provides features to generate OpenShift resources and trigger OpenShift processes, such as builds and deployments.

Chapter 10

Comprehensive Review: Red Hat OpenShift Development II: Containerizing Applications

Goal

Review tasks from *Red Hat OpenShift Development II: Containerizing Applications*

Objectives

Review tasks from *Red Hat OpenShift Development II: Containerizing Applications*

Sections

Comprehensive Review

Lab

- Lab: Designing a Container Image for OpenShift
- Lab: Containerizing and Deploying a Service
- Lab: Building and Deploying a Multicontainer Application

Comprehensive Review

Objectives

After completing this section, you should be able to review and refresh knowledge and skills learned in *Red Hat OpenShift Development II: Containerizing Applications*.

Reviewing Red Hat OpenShift Development II: Containerizing Applications

Before beginning the comprehensive review for this course, students should be comfortable with the topics covered in each chapter.

Students can refer to earlier sections in the textbook for extra study.

Chapter 1, Deploying and Managing Applications on an OpenShift Cluster

Deploy applications using various application packaging methods to an OpenShift cluster and manage their resources.

- Describe the architecture and new features in OpenShift 4.
- Deploy an application to the cluster from a Dockerfile with the CLI.
- Deploy an application from a container image and manage its resources using the web console.
- Deploy an application from source code and manage its resources using the command-line interface.

Chapter 2, Designing Containerized Applications for OpenShift

Select an application containerization method for an application and package it to run on an OpenShift cluster.

- Select an appropriate application containerization method.
- Build a container image with advanced Dockerfile directives.
- Select a method for injecting configuration data into an application and create the necessary resources to do so.

Chapter 3, Publishing Enterprise Container Images

Interact with an enterprise registry and publish container images to it.

- Manage container images in registries using Linux container tools.
- Access the OpenShift internal registry using Linux container tools.
- Create image streams for container images in external registries.

Chapter 4, Building Applications

Describe the OpenShift build process, trigger and manage builds.

- Describe the OpenShift build process.
- Manage application builds using the BuildConfig resource and CLI commands.
- Trigger the build process with supported methods.
- Process post build logic with a post-commit build hook.

Chapter 5, Customizing Source-to-Image Builds

Customize an existing S2I builder image and create a new one.

- Describe the required and optional steps in the Source-to-Image build process.
- Customize an existing S2I builder image with scripts.
- Create a new S2I builder image with S2I tools.

Chapter 6, Creating Applications from OpenShift Templates

Describe the elements of a template and create a multicontainer application template.

- Describe the elements of an OpenShift template.
- Build a multicontainer application from a custom template.

Chapter 7, Managing Application Deployments

Monitor application health and implement various deployment methods for cloud-native applications.

- Implement liveness and readiness probes.
- Select the appropriate deployment strategy for a cloud-native application.
- Manage the deployment of an application with CLI commands.

Chapter 8, Implementing Continuous Integration and Continuous Deployment Pipelines in OpenShift

Create and deploy Jenkins pipelines to facilitate continuous integration and deployment with OpenShift.

The objectives for this chapter are not included in the Comprehensive Review lab.

Chapter 9, Building Applications for OpenShift

Create and deploy applications on OpenShift.

- Integrate a containerized application with non-containerized services.
- Deploy containerized third-party applications following recommended practices for OpenShift.
- Use a Red Hat OpenShift Application Runtime to deploy an application.

► Lab

Designing a Container Image for OpenShift

In this lab, you will optimize the container image for the To Do List application front end, and publish it to a registry server to be consumed by an OpenShift cluster.

Outcomes

You should be able to:

- Optimize a Dockerfile to reduce the number of layers in the generated container image.
- Change a Dockerfile so that the generated container image can be deployed using the standard OpenShift security policies.
- Publish the generated container image to an external registry.
- Create an OpenShift image stream, in a shared project, to deploy applications using the generated container image.

Before You Begin

To perform this exercise, ensure that you have access to:

- A running OpenShift cluster.
- The Linux container tools.
- The Universal Base Image (UBI) 8 container image.
- A personal GitHub fork and a local clone of the **DO288-apps** repository, which contains a folder with the To Do List application front-end source and its Dockerfile in the **todo-frontend** folder.
- A personal, free account at Quay.io.

Run the following command on **workstation** to validate the prerequisites and download the solution files.

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

Instructions

All of review labs for this course are based on the same use case; that is, a multicontainer version of the To Do List application. When you have finished all the review labs, the application should be deployed as three pods:

- A single-page web front-end pod, based on Nginx.
- An HTTP API back-end pod, based on Node.js.
- A database pod, based on MySQL.

The first review lab builds the To Do List front end as a custom container image, using a Dockerfile. The front-end development team has not migrated to OpenShift yet, and your job is to make sure that their container image follows Red Hat recommendations for deployment to OpenShift.

You must deploy the To Do List front end according to the following specifications:

- Retrieve the To Do List front-end HTML sources, and Dockerfile, from a local clone of your personal fork of the **DO288-apps** Git repository, in the **todo-frontend** folder. Create a branch named **review-dockerfile** to save your changes.
- The provided Dockerfile generates an image that is compliant with Open Container Initiative (OCI) container engines, but may require changes to comply with Red Hat recommendations for OpenShift. Make no change to the HTML and JavaScript sources from the To Do List front end.
- Minimize the number of layers in the To Do List front-end container image.
- Publish the front-end container image into your personal Quay.io account as **quay.io/yourquayuser/front-end:latest**.
- Create an image stream called **todo-frontend**, in the **youruser-review-common** project, that points to the front-end container image in your personal Quay.io account.
- Do not make any changes to security context constraints and service accounts in the **youruser-review-dockerfile** project. You may need to grant permissions on the **youruser-review-common** project so other projects can access its image streams and images.
- Deploy the To Do List front end on the **youruser-review-dockerfile** project, using **frontend** as the name of your OpenShift resources.
- Set the environment variable **BACKEND_HOST** on the front end deployment to **api.example.com** as a placeholder, until you get a working back end to test.
- Test the To Do List front end using the default host name that OpenShift generates for new routes:

`http://frontend-youruser-review-dockerfile.apps.cluster.example.com`

Evaluation

As the **student** user on **workstation**, run the following command to confirm success on this exercise. Correct any reported failures and rerun the command until successful.

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

Finish

Do not perform any clean up. The OpenShift projects and resources created during this review lab will be used to complete the third review lab.

Run the **finish** command to signal that you completed this exercise:

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

To restart this review lab, use the **cleanup** command. Note that you cannot perform to the third review lab if you clean up this one.

This concludes the lab.

► Solution

Designing a Container Image for OpenShift

In this lab, you will optimize the container image for the To Do List application front end, and publish it to a registry server to be consumed by an OpenShift cluster.

Outcomes

You should be able to:

- Optimize a Dockerfile to reduce the number of layers in the generated container image.
- Change a Dockerfile so that the generated container image can be deployed using the standard OpenShift security policies.
- Publish the generated container image to an external registry.
- Create an OpenShift image stream, in a shared project, to deploy applications using the generated container image.

Before You Begin

To perform this exercise, ensure that you have access to:

- A running OpenShift cluster.
- The Linux container tools.
- The Universal Base Image (UBI) 8 container image.
- A personal GitHub fork and a local clone of the **DO288-apps** repository, which contains a folder with the To Do List application front-end source and its Dockerfile in the **todo-frontend** folder.
- A personal, free account at Quay.io.

Run the following command on **workstation** to validate the prerequisites and download the solution files.

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

Instructions

All of review labs for this course are based on the same use case; that is, a multicontainer version of the To Do List application. When you have finished all the review labs, the application should be deployed as three pods:

- A single-page web front-end pod, based on Nginx.
- An HTTP API back-end pod, based on Node.js.
- A database pod, based on MySQL.

The first review lab builds the To Do List front end as a custom container image, using a Dockerfile. The front-end development team has not migrated to OpenShift yet, and your job is to make sure that their container image follows Red Hat recommendations for deployment to OpenShift.

You must deploy the To Do List front end according to the following specifications:

- Retrieve the To Do List front-end HTML sources, and Dockerfile, from a local clone of your personal fork of the **DO288-apps** Git repository, in the **todo-frontend** folder. Create a branch named **review-dockerfile** to save your changes.
- The provided Dockerfile generates an image that is compliant with Open Container Initiative (OCI) container engines, but may require changes to comply with Red Hat recommendations for OpenShift. Make no change to the HTML and JavaScript sources from the To Do List front end.
- Minimize the number of layers in the To Do List front-end container image.
- Publish the front-end container image into your personal Quay.io account as **quay.io/yourquayuser/front-end:latest**.
- Create an image stream called **todo-frontend**, in the **youruser-review-common** project, that points to the front-end container image in your personal Quay.io account.
- Do not make any changes to security context constraints and service accounts in the **youruser-review-dockerfile** project. You may need to grant permissions on the **youruser-review-common** project so other projects can access its image streams and images.
- Deploy the To Do List front end on the **youruser-review-dockerfile** project, using **frontend** as the name of your OpenShift resources.
- Set the environment variable **BACKEND_HOST** on the front end deployment to **api.example.com** as a placeholder, until you get a working back end to test.
- Test the To Do List front end using the default host name that OpenShift generates for new routes:
`http://frontend-youruser-review-dockerfile.apps.cluster.example.com`

1. Create a branch to store your changes to the Dockerfile of the To Do List front-end, and change the Dockerfile to follow Red Hat recommendations for container images.
 - 1.1. Enter your local clone of the **DO288-apps** Git repository, and then checkout the **master** branch of the course repository to ensure that you start this exercise from a known good state:

```
[student@workstation ~]$ cd DO288-apps
[student@workstation DO288-apps]$ git checkout master
...output omitted...
```

- 1.2. Create a new branch to save any changes that you make during this exercise:

```
[student@workstation D0288-apps]$ git checkout -b review-dockerfile
Switched to a new branch 'review-dockerfile'
[student@workstation D0288-apps]$ git push -u origin review-dockerfile
...output omitted...
* [new branch]      review-dockerfile -> review-dockerfile
Branch review-dockerfile set up to track remote branch review-dockerfile from
origin.
```

- 1.3. Optimize the provided Dockerfile to reduce the number of layers in the container image.

Open the **~/D0288-apps/todo-frontend/Dockerfile** file in a text editor, and then merge adjacent **LABEL** and **RUN** instructions. After all edits are made, the file contents should be similar to the following:

```
FROM registry.access.redhat.com/ubi8:8.0

LABEL version="1.0" \
      description="To Do List application front-end" \
      creationDate="2017-12-25" \
      updatedDate="2019-08-01"

ENV BACKEND_HOST=localhost:8081

RUN yum install -y --disableplugin=subscription-manager --nodocs \
    nginx nginx-mod-http-perl \
    && yum clean all

COPY nginx.conf /etc/nginx/

RUN touch /run/nginx.pid \
    && chgrp -R nginx /var/log/nginx /run/nginx.pid \
    && chmod -R g+rwx /var/log/nginx /run/nginx.pid

COPY src/ /usr/share/nginx/html

EXPOSE 8080

USER nginx

CMD nginx -g "daemon off;"
```

If you do not want to make the edits yourself, copy the **~/D0288/solutions/review-dockerfile/Dockerfile-optimized** file to **~/D0288-apps/todo-frontend/Dockerfile**, and then continue to the next step.

- 1.4. Change the Dockerfile to generate a container image that runs under the default OpenShift security policies, and does not require a fixed User ID. Configure the image to run as User ID 1001, following OpenShift conventions.

Open the **~/D0288-apps/todo-frontend/Dockerfile** in a text editor and change the existing **RUN** instructions; this changes all writable files to belong to the group 0 (zero), and to be writable by the group zero. After all edits are made, the file contents should be similar to the following:

```

FROM registry.access.redhat.com/ubi8:8.0

LABEL version="1.0" \
      description="To Do List application front-end" \
      creationDate="2017-12-25" \
      updatedDate="2019-08-01"

ENV BACKEND_HOST=localhost:8081

RUN yum install -y --disableplugin=subscription-manager --nодocs \
    nginx nginx-mod-http-perl \
    && yum clean all

COPY nginx.conf /etc/nginx/

RUN touch /run/nginx.pid \
    && chgrp -R 0 /var/log/nginx /run/nginx.pid \
    && chmod -R g+rwx /var/log/nginx /run/nginx.pid

COPY src/ /usr/share/nginx/html

EXPOSE 8080

USER 1001

CMD nginx -g "daemon off;"
```

If you do not want to make these edits yourself, copy the **~/D0288/solutions/review-dockerfile/Dockerfile-default-scc** file to **~/D0288-apps/todo-frontend/Dockerfile**, and then continue to the next step.

2. Build the To Do List front-end container image, and optionally test it locally. A successful local test does not ensure that the image will work under OpenShift default security policies, but catches some errors that you might make when editing your Dockerfile.
 - 2.1. Navigate to the **~/D0288-apps/todo-frontend** folder, and then build the **todo-frontend** container image.

```
[student@workstation D0288-apps]$ cd todo-frontend
[student@workstation todo-frontend]$ sudo podman build -t todo-frontend .
...output omitted...
STEP 19: COMMIT todo-frontend
```

- 2.2. Start a local container to verify that your changes did not break the image.

```
[student@workstation todo-frontend]$ sudo podman run --name testfrontend \
> -d -p 8080:8080 todo-frontend
16a4...4a3a
```

- 2.3. Use Curl to verify that the image returns the To Do List front-end welcome page.

```
[student@workstation todo-frontend]$ curl -s localhost:8080 | grep h1
<h1>To Do List Application</h1>
```

- 2.4. Stop and remove your test container.

```
[student@workstation todo-frontend]$ sudo podman stop testfrontend
16a4...4a3a
[student@workstation todo-frontend]$ sudo podman rm testfrontend
16a4...4a3a
```

- 2.5. Commit and push your changes to the Dockerfile.

```
[student@workstation todo-frontend]$ git commit -a -m 'Fixed for OpenShift'
...output omitted...
[student@workstation todo-frontend]$ git push
...output omitted...
```

- 2.6. Leave your Dockerfile project folder, as you should only use the published image for the remainder of this exercise.

```
[student@workstation todo-frontend]$ cd ~
...output omitted...
[student@workstation ~]$
```

3. Publish the To Do List front-end container image on Quay.io. Source your classroom configuration variables from **/usr/local/etc/ocp4.config** before performing the push operation.

- 3.1. Load your classroom environment configuration.

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

- 3.2. Log in on Quay.io as **root**, so the Skopeo command in the next step can create a new image on your Quay.io account.

```
[student@workstation ~]$ sudo podman login -u ${RHT_OCP4_QUAY_USER} quay.io
Password: your Quay.io password
Login Succeeded!
```

- 3.3. Push the local **todo-frontend** container image to Quay.io.

```
[student@workstation ~]$ sudo skopeo copy \
> containers-storage:localhost/todo-frontend \
> docker://quay.io/${RHT_OCP4_QUAY_USER}/todo-frontend
...output omitted...
Writing manifest to image destination
Storing signatures
```

4. Make the To Do List front-end container image available to OpenShift users as the **todo-frontend** image stream in the **youruser-review-common** project.

- 4.1. Log in to OpenShift using your developer user account, and then create the ***youruser-review-common*** project.

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful.
...output omitted...
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-review-common
Now using project "youruser-review-common" on server
"https://api.cluster.domain.example.com:6443".
```

- 4.2. Log in again on Quay.io, this time as **student**, to save an access token for the secret you create in the next step.

```
[student@workstation ~]$ podman login -u ${RHT_OCP4_QUAY_USER} quay.io
Password: your Quay.io password
Login Succeeded!
```

- 4.3. Create a secret from the container registry API access token that was stored by Podman.

```
[student@workstation ~]$ oc create secret generic quayio \
> --from-file .dockerconfigjson=${XDG_RUNTIME_DIR}/containers/auth.json \
> --type kubernetes.io/dockerconfigjson
secret/quayio created
```

- 4.4. Create the **todo-frontend** image stream that points to the container image in your personal Quay.io account.

```
[student@workstation ~]$ oc import-image todo-frontend --confirm \
> --reference-policy local \
> --from quay.io/${RHT_OCP4_QUAY_USER}/todo-frontend
imagestream.image.openshift.io/todo-frontend imported
...output omitted...
latest
tagged from quay.io/yourquayuser/todo-frontend
prefer registry pullthrough when referencing this tag
* quay.io/yourquayuser/todo-frontend@sha256:9ca4...4651
  Less than a second ago
...output omitted...
```

5. Deploy the To Do List front-end image stream to the ***youruser-review-dockerfile*** project, and verify that it works.

- 5.1. Create the ***youruser-review-dockerfile*** project.

```
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-review-dockerfile
Now using project "youruser-review-dockerfile" on server
"https://api.cluster.domain.example.com:6443".
```

- 5.2. Grant service accounts from the new **youruser-review-dockerfile** project access to image streams from the **youruser-review-common** project.

```
[student@workstation ~]$ oc policy add-role-to-group \
> -n ${RHT_OCP4_DEV_USER}-review-common system:image-puller \
> system:serviceaccounts:${RHT_OCP4_DEV_USER}-review-dockerfile
clusterrole.rbac.authorization.k8s.io/system:image-puller added:
"system:serviceaccounts:youruser-review-dockerfile"
```

- 5.3. Deploy the **frontend** application using the **todo-frontend** image stream from the **youruser-review-common** project. Set the environment variable **BACKEND_HOST** to **api.example.com**.

```
[student@workstation ~]$ oc new-app --as-deployment-config --name frontend \
> -e BACKEND_HOST=api.example.com \
> -i ${RHT_OCP4_DEV_USER}-review-common/todo-frontend
...output omitted...
--> Creating resources ...
imagestreamtag.image.openshift.io "frontend:latest" created
deploymentconfig.apps.openshift.io "frontend" created
service "frontend" created
--> Success
...output omitted...
```

- 5.4. Wait for the application pod to be ready and running.

```
[student@workstation ~]$ oc get pod
NAME          READY   STATUS    RESTARTS   AGE
frontend-1-deploy   0/1     Completed   0          31s
frontend-1-8mwjm   1/1     Running    0          22s
```

- 5.5. Expose the **frontend** application pod:

```
[student@workstation ~]$ oc expose svc frontend
route "frontend" exposed
```

- 5.6. Get the host name of the route:

```
[student@workstation ~]$ oc get route
NAME      HOST/PORT
frontend  frontend-youruser-review-dockerfile.apps.cluster.example.com ...
```

- 5.7. Test the application using the **curl** command and the host name from the previous step.

```
[student@workstation ~]$ curl -s \
> http://frontend-${RHT_OCP4_DEV_USER}-review-dockerfile.\
> ${RHT_OCP4_WILDCARD_DOMAIN} | grep h1
<h1>To Do List Application</h1>
```

Evaluation

As the **student** user on **workstation**, run the following command to confirm success on this exercise. Correct any reported failures and rerun the command until successful.

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

Finish

Do not perform any clean up. The OpenShift projects and resources created during this review lab will be used to complete the third review lab.

Run the **finish** command to signal that you completed this exercise:

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

To restart this review lab, use the **cleanup** command. Note that you cannot perform to the third review lab if you clean up this one.

This concludes the lab.

▶ Lab

Containerizing and Deploying a Service

In this lab, you will deploy the To Do List application back end from source code, using configuration and database access credentials externalized to configuration maps and secrets, as well as a custom S2I script.

Outcomes

You should be able to:

- Create a secret to store database access credentials.
- Deploy an ephemeral MySQL database pod using the secret to provide configuration information.
- Customize the S2I build of the To Do List back end by adding a custom assemble script.
- Deploy the To Do List back end from source code in a Git repository, passing build environment variables.
- Create a configuration map to store application configuration parameters.
- Change the To Do List back-end deployment configuration to use the configuration map and the secret.
- Verify that the To Do List back end successfully initializes the MySQL database.

Before You Begin

To perform this exercise, ensure that you have access to:

- A running OpenShift cluster.
- The MySQL Server 5.7 container image.
- The Node.js 12 builder image.
- A personal GitHub fork and a local clone of the **DO288-apps** repository, which contains the folder with the To Do List application front-end source and its Dockerfile in the **todo-backend** folder.
- Access to a Nexus server with NPM dependencies required by the application.
- Successful completion of the previous comprehensive review lab.

Run the following command on **workstation** to validate the prerequisites and download the solution files.

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

Instructions

All of the review labs for this course are based on the same use case; that is, a multicontainer version of the To Do List application. When you have finished all the review labs, the application should be deployed as three pods:

- A single-page web front-end pod, based on Nginx.
- An HTTP API back-end pod, based on Node.js.
- A database pod, based on MySQL.

The second review lab builds the To Do List back end using the OpenShift Source-to-Image (S2I) process. The back-end development team is starting to use OpenShift, and your job is to make sure that the build process meets your organization's internal standards. You must also ensure that the deployment follows Red Hat's recommendations for externalizing the configuration of applications deployed to OpenShift.

You must deploy the To Do List back end according to the following specifications:

- Deploy a MySQL server on the ***youruser-review-service*** project, using **tododb** as the name of your OpenShift resources and the **rhsc1/mysql-57-rhel7** container image from the public Red Hat registry.
- Store database access credentials in a secret called **tododb** with keys: **user** and **password**. Use this secret to initialize environment variables for both the database and the back end pods.
- The environment variables required by the database pod are **MYSQL_USER**, **MYSQL_PASSWORD**, and **MYSQL_DATABASE**.
- The user name to access the database is **todoapp**, the password is **mypass**, and the database name is **todo**.
- Retrieve the To Do List back end Node.js sources and S2I scripts from a local clone of your personal fork of the **D0288-apps** Git repository, in the **todo-backend** folder. Create a branch named **review-service** to save your changes.
- Build and deploy the To Do List back end on the ***youruser-review-service*** project using **backend** as the name of your OpenShift resources. Use the **nodejs:12** image stream tag as the S2I builder.
- Download the required npm module dependencies from the following URL, which you provide as the value of the **npm_config_registry** build environment variable:

`http://${RHT_OCP4_NEXUS_SERVER}/repository/nodejs`

- The environment variables required by the application pod are **DATABASE_USER**, **DATABASE_PASSWORD**, **DATABASE_NAME**, and **DATABASE_SVC**. Initialize them from the same **tododb** secret used to initialize the environment variables for the database pod.
- Store application configuration parameters in a configuration map called **todoapp**, with the **init** key.
- The only configuration parameter not related to database access is the **DATABASE_INIT** environment variable. Pass the value **true** to force the application to create database tables on startup.
- Integrate the build process with the application life cycle management system for the organization. Use the **lifecycle.sh** script in the **~/D0288/labs/review-service** folder as a placeholder, until the real integration script is available. Do not make any changes to the script, except to call the S2I assemble script provided by the Node.js builder image.
- Test the To Do List back end using the default host name that OpenShift generates for new routes:

```
http://backend-youruser-review-  
service.apps.cluster.domain.example.com
```

Use the following HTTP API entry point that returns the number of items in the To Do List:

```
/todo/api/items-count
```

If you get zero items, then the back end is working and able to access the database.

This review lab does not require anything from the previous review lab, and its resources are not used by the third review lab as input to create the template for deploying the complete To Do List application. In the third review lab you will be provided with a cleaned template based on the resources from this lab, whether you complete this second review lab or not.

Evaluation

As the **student** user on **workstation**, run the following command to confirm success on this exercise. Correct any reported failures and rerun the command until successful.

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

Finish

Do not perform any clean up. You can continue to the third review lab without completing this second lab review; however you must complete the first review lab before starting the third review lab.

Run the **finish** command to signal that you completed this exercise:

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

To restart this review lab, use the **cleanup** command. Note that you cannot perform to the third review lab if you clean up this one.

This concludes the lab.

► Solution

Containerizing and Deploying a Service

In this lab, you will deploy the To Do List application back end from source code, using configuration and database access credentials externalized to configuration maps and secrets, as well as a custom S2I script.

Outcomes

You should be able to:

- Create a secret to store database access credentials.
- Deploy an ephemeral MySQL database pod using the secret to provide configuration information.
- Customize the S2I build of the To Do List back end by adding a custom assemble script.
- Deploy the To Do List back end from source code in a Git repository, passing build environment variables.
- Create a configuration map to store application configuration parameters.
- Change the To Do List back-end deployment configuration to use the configuration map and the secret.
- Verify that the To Do List back end successfully initializes the MySQL database.

Before You Begin

To perform this exercise, ensure that you have access to:

- A running OpenShift cluster.
- The MySQL Server 5.7 container image.
- The Node.js 12 builder image.
- A personal GitHub fork and a local clone of the **DO288-apps** repository, which contains the folder with the To Do List application front-end source and its Dockerfile in the **todo-backend** folder.
- Access to a Nexus server with NPM dependencies required by the application.
- Successful completion of the previous comprehensive review lab.

Run the following command on **workstation** to validate the prerequisites and download the solution files.

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

Instructions

All of the review labs for this course are based on the same use case; that is, a multicontainer version of the To Do List application. When you have finished all the review labs, the application should be deployed as three pods:

- A single-page web front-end pod, based on Nginx.
- An HTTP API back-end pod, based on Node.js.
- A database pod, based on MySQL.

The second review lab builds the To Do List back end using the OpenShift Source-to-Image (S2I) process. The back-end development team is starting to use OpenShift, and your job is to make sure that the build process meets your organization's internal standards. You must also ensure that the deployment follows Red Hat's recommendations for externalizing the configuration of applications deployed to OpenShift.

You must deploy the To Do List back end according to the following specifications:

- Deploy a MySQL server on the ***youruser-review-service*** project, using **tododb** as the name of your OpenShift resources and the **rhscl/mysql-57-rhel7** container image from the public Red Hat registry.
- Store database access credentials in a secret called **tododb** with keys: **user** and **password**. Use this secret to initialize environment variables for both the database and the back end pods.
- The environment variables required by the database pod are **MYSQL_USER**, **MYSQL_PASSWORD**, and **MYSQL_DATABASE**.
- The user name to access the database is **todoapp**, the password is **mypass**, and the database name is **todo**.
- Retrieve the To Do List back end Node.js sources and S2I scripts from a local clone of your personal fork of the **D0288-apps** Git repository, in the **todo-backend** folder. Create a branch named **review-service** to save your changes.
- Build and deploy the To Do List back end on the ***youruser-review-service*** project using **backend** as the name of your OpenShift resources. Use the **nodejs:12** image stream tag as the S2I builder.
- Download the required npm module dependencies from the following URL, which you provide as the value of the **npm_config_registry** build environment variable:

`http://${RHT_OCP4_NEXUS_SERVER}/repository/nodejs`

- The environment variables required by the application pod are **DATABASE_USER**, **DATABASE_PASSWORD**, **DATABASE_NAME**, and **DATABASE_SVC**. Initialize them from the same **tododb** secret used to initialize the environment variables for the database pod.
- Store application configuration parameters in a configuration map called **todoapp**, with the **init** key.
- The only configuration parameter not related to database access is the **DATABASE_INIT** environment variable. Pass the value **true** to force the application to create database tables on startup.
- Integrate the build process with the application life cycle management system for the organization. Use the **lifecycle.sh** script in the **~/D0288/labs/review-service** folder as a placeholder, until the real integration script is available. Do not make any changes to the script, except to call the S2I assemble script provided by the Node.js builder image.
- Test the To Do List back end using the default host name that OpenShift generates for new routes:

```
http://backend-youruser-review-
service.apps.cluster.domain.example.com
```

Use the following HTTP API entry point that returns the number of items in the To Do List:

```
/todo/api/items-count
```

If you get zero items, then the back end is working and able to access the database.

This review lab does not require anything from the previous review lab, and its resources are not used by the third review lab as input to create the template for deploying the complete To Do List application. In the third review lab you will be provided with a cleaned template based on the resources from this lab, whether you complete this second review lab or not.

1. Source your classroom configuration variables from `/usr/local/etc/ocp4.config`, log in to OpenShift, and create the **`youruser-review-service`** project. Deploy a MySQL server to the project. When the database deployment has finished, disable its configuration change triggers.

1.1. Load your classroom environment configuration.

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

- 1.2. Log in to OpenShift using your developer user account, and create the **`youruser-review-service`** project.

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful.
...output omitted...
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-review-service
Now using project "youruser-review-service" on server
"https://api.cluster.domain.example.com:6443".
```

- 1.3. Deploy an ephemeral database server from the MySQL container image. Provide values for all required environment variables.

```
[student@workstation ~]$ oc new-app --as-deployment-config --name tododb \
> --docker-image registry.access.redhat.com/rhscl/mysql-57-rhel7 \
> -e MYSQL_USER=todoapp \
> -e MYSQL_PASSWORD=mypass \
> -e MYSQL_DATABASE=todo
...output omitted...
--> Creating resources ...
imagestream.image.openshift.io "tododb" created
deploymentconfig.apps.openshift.io "tododb" created
service "tododb" created
--> Success
...output omitted...
```

- 1.4. Wait until the database pod is ready and running.

```
[student@workstation ~]$ oc get pod
NAME          READY   STATUS    RESTARTS   AGE
tododb-1-deploy  0/1     Completed  0          36s
tododb-1-tjmvs  1/1     Running   0          28s
```

- 1.5. Disable configuration triggers in the database deployment configuration. This is not strictly required, but makes the next steps faster by avoiding multiple deployments.

```
[student@workstation ~]$ oc set triggers dc/tododb --from-config --remove
deploymentconfig.apps.openshift.io/tododb triggers updated
```

2. Create the **tododb** secret to store the database connection parameters, and then change the deployment configuration to initialize the environment variables from the secret.

- 2.1. Create the **tododb** secret with keys to store the user name and password.

```
[student@workstation ~]$ oc create secret generic tododb \
> --from-literal user=todoapp \
> --from-literal password=mypass
secret/tododb created
```

- 2.2. Change the environment variables in the database deployment configuration to use the secret keys. Use the **--prefix MYSQL_** option to ensure the environment variable names match the ones expected by the database pod.

```
[student@workstation ~]$ oc set env dc/tododb \
> --prefix MYSQL_ \
> --from secret/tododb
deploymentconfig.apps.openshift.io/tododb updated
```

- 2.3. Verify that the deployment configuration initializes the environment variables from the secret.

```
[student@workstation ~]$ oc set env dc/tododb --list
# deploymentconfigs tododb, container tododb
MYSQL_DATABASE=todo
# MYSQL_PASSWORD from secret tododb, key password
# MYSQL_USER from secret tododb, key user
```

3. Redeploy the database to apply changes made to the deployment configuration.

- 3.1. Deploy a new database pod using the updated deployment configuration.

```
[student@workstation ~]$ oc rollout latest dc/tododb
deploymentconfig.apps.openshift.io/tododb rolled out
```

- 3.2. Wait for the new database pod to be ready and running.

```
[student@workstation ~]$ oc get pod
NAME          READY   STATUS    RESTARTS   AGE
tododb-1-deploy  0/1     Completed  0          3m45s
tododb-2-deploy  0/1     Completed  0          36s
tododb-2-dgxwm  1/1     Running   0          46s
```

- 3.3. Verify that the database environment variables are initialized properly.

```
[student@workstation ~]$ oc rsh tododb-2-dgxwm env | grep MYSQL_
MYSQL_DATABASE=todo
MYSQL_PASSWORD=mypass
MYSQL_USER=todoapp
...output omitted...
```

4. Customize the application build with custom S2I scripts.

- 4.1. Enter your local clone of the **D0288-apps** Git repository, and checkout the **master** branch of the course's repository to get the sources of the To Do List back end:

```
[student@workstation ~]$ cd D0288-apps
[student@workstation D0288-apps]$ git checkout master
...output omitted...
```

- 4.2. Create a new branch to save any changes you make during this exercise:

```
[student@workstation D0288-apps]$ git checkout -b review-service
Switched to a new branch 'review-service'
[student@workstation D0288-apps]$ git push -u origin review-service
...output omitted...
* [new branch]      review-service -> review-service
Branch review-service set up to track remote branch review-service from origin.
```

- 4.3. Create a folder to store custom S2I scripts, and then copy the **lifecycle.sh** script so that it is used as the assemble S2I script.

Copy the **lifecycle.sh** script to **./.s2i/bin/assemble** so that it is executed by the OpenShift S2I process.

```
[student@workstation D0288-apps]$ mkdir -p ~/D0288-apps/todo-backend/.s2i/bin
[student@workstation D0288-apps]$ cp ~/D0288/labs/review-service/lifecycle.sh \
> ~/D0288-apps/todo-backend/.s2i/bin/assemble
```

- 4.4. Inspect the **nodejs:12** image stream tag to determine the location of the S2I scripts inside its builder image.

```
[student@workstation D0288-apps]$ oc describe istag nodejs:12 -n openshift \
> | grep io.openshift.s2i.scripts-url
"io.openshift.s2i.scripts-url": "image:///usr/libexec/s2i",
```

- 4.5. Edit the **~/D0288-apps/todo-backend/.s2i/bin/assemble** script to call the standard S2I assemble script from the Node.js S2I builder image.

```
#!/bin/bash

echo "Performing the S2I build..."

#TODO: add call to the standard S2I assemble script
/usr/libexec/s2i/assemble

rc=$?

if [ $rc -eq 0 ]; then
    echo "Recording successful build on the life cycle management system..."
else
    echo "Not calling the life cycle management system: S2I build failed!"
fi
exit $rc
```

4.6. Commit and push the changes to the Git repository.

```
[student@workstation D0288-apps]$ cd ~/D0288-apps/todo-backend
[student@workstation todo-backend]$ git add .s2i
[student@workstation todo-backend]$ git commit -m 'Add custom assemble script'
...output omitted...
[student@workstation todo-backend]$ git push
...output omitted...
```

4.7. Leave your Node.js project folder.

```
[student@workstation todo-backend]$ cd ~
[student@workstation ~]$
```

5. Deploy the To Do List application back end from source code. Do not pass any value for **DATABASE_INIT**. Pass values only for the environment variables that provide database access parameters. When the deployment is finished, disable configuration change triggers.

- 5.1. Deploy the To Do List back end from the source code in the Git repository. Provide values for all required environment variables. Remember to include the flag to enable a DeploymentConfig resource, the value for the **npm_config_registry** build environment variable, the **--contextDir** option, and the branch name in the Git URI:

```
[student@workstation ~]$ oc new-app --as-deployment-config --name backend \
> --build-env npm_config_registry=\
> http://${RHT_OCP4_NEXUS_SERVER}/repository/nodejs \
> -e DATABASE_NAME=todo \
> -e DATABASE_USER=todoapp \
> -e DATABASE_PASSWORD=mypass \
> -e DATABASE_SVC=tododb \
> --context-dir todo-backend \
> nodejs:12-https://github.com/${RHT_OCP4_GITHUB_USER}/D0288-apps#review-service
...output omitted...
--> Creating resources ...
    imagestream.image.openshift.io "backend" created
    buildconfig.build.openshift.io "backend" created
```

```
deploymentconfig.apps.openshift.io "backend" created
service "backend" created
--> Success
...output omitted...
```

- 5.2. Wait for the build to complete. Verify that the build logs show the message generated by the **lifecycle.sh** script.

```
[student@workstation ~]$ oc logs -f bc/backend
...output omitted...
Recording successful build on the life cycle management system...
...output omitted...
Push successful
```

- 5.3. Wait for the application pod to be ready and running.

```
[student@workstation ~]$ oc get pod
NAME          READY   STATUS    RESTARTS   AGE
backend-1-4k3nz  1/1     Running   0          1m
backend-1-build  0/1     Completed  0          2m
backend-1-deploy 0/1     Completed  0          2m
...output omitted...
tododb-2-dgxwm  1/1     Running   0          35m
```

- 5.4. Disable configuration triggers in the application deployment configuration. This is not strictly required, but makes the next steps faster by avoiding multiple deployments.

```
[student@workstation ~]$ oc set triggers dc/backend --from-config --remove
deploymentconfig.apps.openshift.io/backend triggers updated
```

6. Change the application deployment to initialize the environment variables from the secret.

- 6.1. Change the environment variables in the application deployment configuration to use the secret keys. Use the **--prefix DATABASE_** option to ensure the environment variable names match the ones expected by the back-end application pod.

```
[student@workstation ~]$ oc set env dc/backend \
> --prefix DATABASE_ \
> --from secret/tododb
deploymentconfig.apps.openshift.io/backend updated
```

- 6.2. Verify that the deployment configuration initializes the environment variables from the secret.

```
[student@workstation ~]$ oc set env dc/backend --list
# deploymentconfigs backend, container backend
DATABASE_NAME=todo
# DATABASE_PASSWORD from secret tododb, key password
DATABASE_SVC=tododb
# DATABASE_USER from secret tododb, key user
```

Do not force a new deployment, as this is done in a later step.

7. Create the **todoapp** configuration map and change the To Do List back-end deployment to initialize the application parameters from the configuration map.

- 7.1. Create the **todoapp** configuration map with a single key to store the **DATABASE_INIT** configuration parameter. Remember that the **DATABASE_** prefix is not part of the configuration map key.

```
[student@workstation ~]$ oc create cm todoapp --from-literal init=true
configmap "todoapp" created
```

- 7.2. Change the environment variables in the application deployment configuration to use the configuration map keys.

```
[student@workstation ~]$ oc set env dc/backend \
> --prefix=DATABASE_ \
> --from=cm/todoapp
deploymentconfig.apps.openshift.io/backend updated
```

- 7.3. Verify that the deployment configuration initializes the **DATABASE_INIT** environment variables from the configuration map.

```
[student@workstation ~]$ oc set env dc/backend --list
# deploymentconfigs backend, container backend
DATABASE_NAME=todo
# DATABASE_PASSWORD from secret tododb, key password
DATABASE_SVC=tododb
# DATABASE_USER from secret tododb, key user
# DATABASE_INIT from configmap todoapp, key init
```

8. Redeploy the application to apply the changes made to the deployment configuration.

- 8.1. Deploy a new application pod using the updated deployment configuration.

```
[student@workstation ~]$ oc rollout latest dc/backend
deploymentconfig.apps.openshift.io/backend rolled out
```

- 8.2. Wait for the new application pod to be ready and running.

NAME	READY	STATUS	RESTARTS	AGE
backend-1-build	0/1	Completed	0	31m
backend-1-deploy	0/1	Completed	0	31m
backend-2- 1bjrv	1/1	Running	0	26s
backend-2-deploy	0/1	Completed	0	31m
<i>...output omitted...</i>				
tododb-2- dgxwm	1/1	Running	0	35m

- 8.3. Verify that the application environment variables are initialized properly.

```
[student@workstation ~]$ oc rsh backend-2-1bjrv env | grep DATABASE_
DATABASE_NAME=todo
DATABASE_PASSWORD=mypass
DATABASE_SVC=tododb
DATABASE_USER=todoapp
DATABASE_INIT=true
```

9. Use the **curl** command to test the To Do List back end. The `/todo/api/items-count` should return zero items, and should not return a database error. Also, verify that the database contains a single, empty table called **Items**.

- 9.1. Expose the **backend** deployment.

```
[student@workstation ~]$ oc expose svc backend
route.route.openshift.io/backend exposed
```

- 9.2. Get the host name of the route:

```
[student@workstation ~]$ oc get route
NAME      HOST/PORT
backend   backend-youruser-review-service.apps.cluster.domain.example.com ...
```

- 9.3. Use the **curl** command and the host name from the previous step to verify that the To Do List back end initialized the database.

```
[student@workstation ~]$ curl -si \
> backend-$RHT_OCP4_DEV_USER-review-service.$RHT_OCP4_WILDCARD_DOMAIN\
> /todo/api/items-count
HTTP/1.1 200 OK
...
{"count":0}
```

- 9.4. Create a port-forwarding tunnel to the database pod. This step, and the next, are not strictly necessary, but may prove useful if you need to troubleshoot environment variable differences between the database and the application pod.

```
[student@workstation ~]$ oc port-forward tododb-2-dgxwm 30306:3306
Forwarding from 127.0.0.1:30306 -> 3306
Forwarding from [::1]:30306 -> 3306
```

- 9.5. Open another terminal window and use the **mysqlshow** command to verify that the MySQL server has a database called **todo** and a single table named **Items**.

```
[student@workstation ~]$ mysqlshow -utodoapp -pmypass -h127.0.0.1 -P30306 todo
Database: todo
+-----+
| Tables |
+-----+
| Item   |
+-----+
```

9.6. Type **Ctrl+C** to terminate the port-forwarding tunnel.

Evaluation

As the **student** user on **workstation**, run the following command to confirm success on this exercise. Correct any reported failures and rerun the command until successful.

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

Finish

Do not perform any clean up. You can continue to the third review lab without completing this second lab review; however you must complete the first review lab before starting the third review lab.

Run the **finish** command to signal that you completed this exercise:

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

To restart this review lab, use the **cleanup** command. Note that you cannot perform to the third review lab if you clean up this one.

This concludes the lab.

▶ Lab

Building and Deploying a Multicontainer Application

In this lab, you will create a template to deploy the To Do List application front end and back end, together with a MySQL database. You must make the template reusable by adding parameters, and also add health probes to the template.

Outcomes

You should be able to:

- Change the resource definitions for a secret in the template file.
- Add parameters to the template, and reference these parameters from resources inside the template.
- Add health probes to the template.
- Create the template as an OpenShift resource.
- Deploy an application from the template resource.
- Load test data into the database pod.

Before You Begin

To perform this exercise, ensure that you have access to:

- A running OpenShift cluster.
- The To Do List front-end container image (**todo-frontend**) on your personal Quay.io account. You created this image and pushed it to your personal Quay.io account during the first comprehensive review lab.
- The **frontend** image stream in the **youruser-review-common** project, from the first comprehensive review lab. If you have not completed the first comprehensive review lab, go back and follow the solution steps. The solution steps provide finished files and scripts so you can complete the comprehensive review lab faster.
- The MySQL Server 5.7 container image.
- The Node.js 12 builder image.
- A personal GitHub fork, and a local clone of the **DO288-apps** repository, that contains the folder with the To Do List application front-end source and its Dockerfile in the **todo-backend** folder.
- Access to a Nexus server with NPM dependencies, as required by the application.

Run the following command on **workstation** to validate the prerequisites and to download solution files.

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

Instructions

All course review labs are based on the same use case; that is, a multicontainer version of the To Do List application. When you have finished all the review labs, the application will be deployed as three pods:

- A single-page web front-end pod, based on Nginx.
- An HTTP API back-end pod, based on Node.js.
- A database pod, based on MySQL.

The third review lab creates a template to deploy the complete To Do List application, using the resource definitions created during the two previous review labs. To save time, you are provided with the resources already exported to a YAML file and cleaned up. Your job is to make the template reusable, and to implement Red Hat recommended practices by adding health probes to the template.

You must deploy the complete To Do List application according to the following specifications:

- Create the **todoapp** template resource in the **youruser-review-common** project.
- The template takes the following parameters. All of these parameters are required:
 - **DATABASE_IMAGE**: The URL of the MySQL server container image. Its default value is:
`registry.access.redhat.com/rhscl/mysql-57-rhel7`.
 - **BACK-END_REPO**: The URL of the back-end sources. Its default value is:
`https://github.com/yourgituser/D0288-apps`.
 - **BACK-END_CTXDIR**: The folder that contains the back-end sources. Its default value is:
todo-backend.
 - **BACK-END_BRANCH**: The branch to fetch the back-end sources. Its default value is:
master.
 - **NPM_PROXY**: The URL of the npm repository server. Its default value is:
`http://nexus-common.apps.cluster.domain.example.com/repository/nodejs`
 - **SECRET**: The secret for OpenShift webhooks. Its default value is randomly generated.
 - **PASSWORD**: The database connection password. It has no default value.
 - **HOSTNAME**: The host name used to access the To Do List front end from a web browser. It has no default value.
 - **BACKEND**: The host name used by the To Do List front end to access the To Do List back end. It has no default value.
 - **CLEAN_DATABASE**: Indicates whether the application initializes the database on start up. Its default value is "**false**". The double quotes are required.
- Use the **nodejs-mongodb-example** standard template as a reference for the syntax to define parameters, and to reference parameters inside the template resource list.

- The database user name **todoapp** is fixed in the template, as well as the database name **todo**.
- Use the following entry points from the HTTP API of the To Do List back-end as health probes:
 - Readiness: /todo/api/host
 - Liveness: /todo/api/items-count
- The liveness health probe fails until the database is populated.
- Configure both probes with the following attributes:
 - **initialDelaySeconds: 10**
 - **timeoutSeconds: 3**
- Deploy the template to the **youruser-review-multicontainer** project. Pass parameters to the template to get the following results:
 - The front-end application is available at:
`http://todoui-youruser.apps.cluster.domain.example.com`
 - The back-end application is available at:
`http://todoapi-youruser.apps.cluster.domain.example.com`
 - On the two previous URLs, use the shell variable **RHT_OCP4_WILDCARD_DOMAIN** to provide the value of `apps.cluster.domain.example.com`, and the shell variable **RHT_OCP4_DEV_USER** to provide the value of `youruser`. These variables are defined in the `/usr/local/etc/ocp4.config` shell file.
 - The database password is: **redhat**
 - The back-end application does *not* initialize the database.
- Populate the database using the **todo.sql** script in the `~/D0288/labs/review-multicontainer` folder.
- Because the template deploys the To Do List front end from the image stream you created during the first comprehensive review lab, you must grant permissions to service accounts from the **youruser-review-multicontainer** project. Do not make any changes to the template file, except to use the parameters, as stated above.

The **todoapp.yml** starter template file is exported from the **youruser-review-dockerfile** and **youruser-review-service** projects, created during the previous comprehensive review labs. Although you will not work with these projects any further, this lab requires the **youruser-review-common** project, and the container image and image stream, created by performing the first review lab.

Evaluation

As the **student** user on **workstation**, run the following command to confirm success on this exercise. Correct any reported failures and rerun the command until successful.

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

Finish

Run the **finish** command to signal you completed this exercise:

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

To restart all comprehensive review labs, use the **cleanup** command for each lab, in reverse order.

```
[student@workstation ~]$ lab review-multicontainer cleanup  
[student@workstation ~]$ lab review-service cleanup  
[student@workstation ~]$ lab review-dockerfile cleanup
```

This concludes the comprehensive review.

► Solution

Building and Deploying a Multicontainer Application

In this lab, you will create a template to deploy the To Do List application front end and back end, together with a MySQL database. You must make the template reusable by adding parameters, and also add health probes to the template.

Outcomes

You should be able to:

- Change the resource definitions for a secret in the template file.
- Add parameters to the template, and reference these parameters from resources inside the template.
- Add health probes to the template.
- Create the template as an OpenShift resource.
- Deploy an application from the template resource.
- Load test data into the database pod.

Before You Begin

To perform this exercise, ensure that you have access to:

- A running OpenShift cluster.
- The To Do List front-end container image (**todo-frontend**) on your personal Quay.io account. You created this image and pushed it to your personal Quay.io account during the first comprehensive review lab.
- The **frontend** image stream in the **youruser-review-common** project, from the first comprehensive review lab. If you have not completed the first comprehensive review lab, go back and follow the solution steps. The solution steps provide finished files and scripts so you can complete the comprehensive review lab faster.
- The MySQL Server 5.7 container image.
- The Node.js 12 builder image.
- A personal GitHub fork, and a local clone of the **DO288-apps** repository, that contains the folder with the To Do List application front-end source and its Dockerfile in the **todo-backend** folder.
- Access to a Nexus server with NPM dependencies, as required by the application.

Run the following command on **workstation** to validate the prerequisites and to download solution files.

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

Instructions

All course review labs are based on the same use case; that is, a multicontainer version of the To Do List application. When you have finished all the review labs, the application will be deployed as three pods:

- A single-page web front-end pod, based on Nginx.
- An HTTP API back-end pod, based on Node.js.
- A database pod, based on MySQL.

The third review lab creates a template to deploy the complete To Do List application, using the resource definitions created during the two previous review labs. To save time, you are provided with the resources already exported to a YAML file and cleaned up. Your job is to make the template reusable, and to implement Red Hat recommended practices by adding health probes to the template.

You must deploy the complete To Do List application according to the following specifications:

- Create the **todoapp** template resource in the **youruser-review-common** project.
- The template takes the following parameters. All of these parameters are required:
 - **DATABASE_IMAGE**: The URL of the MySQL server container image. Its default value is:
`registry.access.redhat.com/rhscl/mysql-57-rhel7`.
 - **BACK-END_REPO**: The URL of the back-end sources. Its default value is:
`https://github.com/yourgituser/D0288-apps`.
 - **BACK-END_CTXDIR**: The folder that contains the back-end sources. Its default value is:
todo-backend.
 - **BACK-END_BRANCH**: The branch to fetch the back-end sources. Its default value is:
master.
 - **NPM_PROXY**: The URL of the npm repository server. Its default value is:
`http://nexus-common.apps.cluster.domain.example.com/repository/nodejs`
 - **SECRET**: The secret for OpenShift webhooks. Its default value is randomly generated.
 - **PASSWORD**: The database connection password. It has no default value.
 - **HOSTNAME**: The host name used to access the To Do List front end from a web browser. It has no default value.
 - **BACKEND**: The host name used by the To Do List front end to access the To Do List back end. It has no default value.
 - **CLEAN_DATABASE**: Indicates whether the application initializes the database on start up. Its default value is "**false**". The double quotes are required.
- Use the **nodejs-mongodb-example** standard template as a reference for the syntax to define parameters, and to reference parameters inside the template resource list.

- The database user name **todoapp** is fixed in the template, as well as the database name **todo**.
- Use the following entry points from the HTTP API of the To Do List back-end as health probes:
 - Readiness: /todo/api/host
 - Liveness: /todo/api/items-count
- The liveness health probe fails until the database is populated.
- Configure both probes with the following attributes:
 - **initialDelaySeconds: 10**
 - **timeoutSeconds: 3**
- Deploy the template to the **youruser-review-multicontainer** project. Pass parameters to the template to get the following results:
 - The front-end application is available at:
`http://todoui-youruser.apps.cluster.domain.example.com`
 - The back-end application is available at:
`http://todoapi-youruser.apps.cluster.domain.example.com`
 - On the two previous URLs, use the shell variable **RHT_OCP4_WILDCARD_DOMAIN** to provide the value of `apps.cluster.domain.example.com`, and the shell variable **RHT_OCP4_DEV_USER** to provide the value of `youruser`. These variables are defined in the `/usr/local/etc/ocp4.config` shell file.
 - The database password is: **redhat**
 - The back-end application does *not* initialize the database.
- Populate the database using the **todo.sql** script in the `~/DO288/labs/review-multicontainer` folder.
- Because the template deploys the To Do List front end from the image stream you created during the first comprehensive review lab, you must grant permissions to service accounts from the **youruser-review-multicontainer** project. Do not make any changes to the template file, except to use the parameters, as stated above.

The **todoapp.yml** starter template file is exported from the **youruser-review-dockerfile** and **youruser-review-service** projects, created during the previous comprehensive review labs. Although you will not work with these projects any further, this lab requires the **youruser-review-common** project, and the container image and image stream, created by performing the first review lab.

- Review the starter template file.
 - Copy the `~/DO288/labs/review-multicontainer/todoapp.yaml` file to the `/home/student/` folder.

Use the **grep** command to inspect the template and verify the number and kind of resources in the template object list. There are two spaces before **kind:** in the **grep** command.

```
[student@workstation ~]$ cp ~/D0288/labs/review-multicontainer/todoapp.yaml \
> ~/todoapp.yaml
[student@workstation ~]$ grep '^  kind:' ~/todoapp.yaml
  kind: ImageStream
  kind: ImageStream
  kind: ConfigMap
  kind: Secret
  kind: BuildConfig
  kind: DeploymentConfig
  kind: DeploymentConfig
  kind: Service
  kind: Service
  kind: Route
  kind: DeploymentConfig
  kind: Service
  kind: Route
```

- 1.2. Use the **oc process** command to list the parameters already defined in the template.

```
[student@workstation ~]$ oc process --parameters -f ~/todoapp.yaml
NAME          ... VALUE
DATABASE_IMAGE ... registry.access.redhat.com/rhscl/mysql-57-rhel7
BACK_END_REPO  ... https://github.com/yourgituser/D0288-apps
BACK_END_CTXDIR ... todo-backend
BACK_END_BRANCH ... master
NPM_PROXY      ... http://nexus-common.apps.cluster.domain.example.com/
repository/nodejs
SECRET
PASSWORD
HOSTNAME
BACKEND
```

2. Add parameters to the template and use the parameters in the template object list.

To use parameters with a secret resource, update the binary representation of the secret data with a text representation.

Inspect the **nodejs-mongodb-example** template from **openshift**, if you need help with template syntax.

- 2.1. Load your classroom environment configuration, and then log in to OpenShift using your developer user account

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful.
...output omitted...
```

- 2.2. Inspect the **nodejs-mongodb-example** from the **openshift** project. Perform this step in a separate terminal window, so that you can view the sample template as you make edits in subsequent steps. Use this file as a reference for the syntax of the template parameters, secrets, and other resources.

```
[student@workstation ~]$ oc get template nodejs-mongodb-example -n openshift \
> -o yaml | less
```

- 2.3. Review and complete the parameters defined at the end of the **~/todoapp.yaml** file. Add the **CLEAN_DATABASE** parameter, and then complete the definition of the **SECRET** parameter.

The following listing displays the complete parameter definitions, and highlights the changes to make. Copy and paste all changes from this step, and those that follow, from the solution template file in the **~/DO288/solutions/review-multicontainer** folder.

```
...output omitted...
parameters:
- displayName: MySQL server container image full name (with registry)
  name: DATABASE_IMAGE
  required: true
  value: registry.access.redhat.com/rhscl/mysql-57-rhel7
- displayName: To Do List back-end Git repository URL
  name: BACK_END_REPO
  required: true
  value: https://github.com/yourgituser/DO288-apps
- displayName: To Do List back-end project root folder
  name: BACK_END_CTXDIR
  required: true
  value: todo-backend
- displayName: Git branch to build the To Do List back-end
  name: BACK_END_BRANCH
  required: true
  value: todo-backend
- displayName: Npm modules repository URL
  name: NPM_PROXY
  required: true
  value: http://nexus-common.apps.cluster.domain.example.com/repository/nodejs
- displayName: Secret for webhooks
  name: SECRET
  required: true
  from: '[a-zA-Z0-9]{40}'
  generate: expression
- displayName: MySQL database password for the todoapp user
  name: PASSWORD
  required: true
- displayName: Host name to access the To Do List front-end web application
  name: HOSTNAME
  required: true
- displayName: Host name to access the To Do List back-end HTTP API
  name: BACKEND
  required: true
- displayName: Flag to initialize (or not) the application database
```

```
name: CLEAN_DATABASE
required: true
value: "false"
```

- 2.4. Change the secret resource in the `~/todoapp.yaml` template file to be initialized from template parameters.

Review the **nodejs-mongodb-example** standard template first, and then use the **oc explain secret** command to get more information about the secret resource attributes, if you are unsure which edits to make.

Replace the **data** attribute with the **stringData** attribute, and then replace the encoded values for the **password** and **user** keys with plain text values. The **password** key references the template **PASSWORD** parameter:

```
...output omitted...
- apiVersion: v1
  stringData:
    password: ${PASSWORD}
    user: todoapp
  kind: Secret
  metadata:
    name: tododb
    type: Opaque
  ...output omitted...
```

- 2.5. Replace attribute values from the remaining resources in the **objects** array attribute with references to the appropriate template parameters.

Edit the resource definitions for the configuration map, and the front-end route, to use the appropriate template parameters.

The resource definitions for the database image stream and build configuration already reference the correct template parameters.

The following listing shows all references to parameters in the template object list, and highlights the attributes to change:

```
...output omitted...
objects:
  ...output omitted...
  - apiVersion: image.openshift.io/v1
    kind: ImageStream
    ...output omitted...
    name: tododb
    spec:
      ...output omitted...
      tags:
        - annotations:
            from:
              kind: DockerImage
              name: ${DATABASE_IMAGE}
  ...output omitted...
  - apiVersion: v1
    data:
      init: ${CLEAN_DATABASE}
    kind: ConfigMap
```

```
metadata:
  name: todoapp
...output omitted...
- apiVersion: v1
  stringData:
    password: ${PASSWORD}
    user: todoapp
  kind: Secret
  metadata:
    name: tododb
...output omitted...
- apiVersion: build.openshift.io/v1
  kind: BuildConfig
  metadata:
...output omitted...
    name: backend
  spec:
...output omitted...
  source:
    contextDir: ${BACK_END_CTXDIR}
    git:
      ref: ${BACK_END_BRANCH}
      uri: ${BACK_END_REPO}
...output omitted...
    strategy:
      sourceStrategy:
        env:
          - name: npm_config_registry
            value: ${NPM_PROXY}
...output omitted...
    triggers:
      - github:
          secret: ${SECRET}
          type: GitHub
      - generic:
          secret: ${SECRET}
          type: Generic
...output omitted...
- apiVersion: route.openshift.io/v1
  kind: Route
  metadata:
...output omitted...
    name: backend
...output omitted...
  spec:
    host: ${BACKEND}
...output omitted...
- apiVersion: apps.openshift.io/v1
  kind: DeploymentConfig
...output omitted...
    name: frontend
...output omitted...
    template:
...output omitted...
  spec:
```

```

  containers:
    - env:
        - name: BACKEND_HOST
          value: ${BACKEND}
      imagePullPolicy: Always
    ...output omitted...
  - apiVersion: route.openshift.io/v1
    kind: Route
    metadata:
    ...output omitted...
      name: frontend
    ...output omitted...
    spec:
      host: ${HOSTNAME}
    ...output omitted...

```

3. Add health probes to the To Do List back-end deployment configuration within the template.

- 3.1. The **backend** deployment configuration does not include health probes, so add them to the template definition.

Review the **nodejs-mongodb-example** standard template, and copy the health probes from that template to the **~/todoapp.yaml** file, adding them to the end of the **spec.template.spec.containers** attribute of the back-end deployment configuration. Change the probe attributes to match the application requirements.

The following listing shows the probe definition. Copy and paste all changes in this step, and preceding steps, from the solution template file in the **~/DO288/solutions/review-multicontainer** folder.

```

...output omitted...
- apiVersion: v1
  kind: DeploymentConfig
  metadata:
  ...output omitted...
    name: backend
  ...output omitted...
  spec:
  ...output omitted...
    template:
  ...output omitted...
    spec:
      containers:
    ...output omitted...
      terminationMessagePolicy: File
      livenessProbe:
        httpGet:
          path: /todo/api/items-count
          port: 8080
          initialDelaySeconds: 10
          timeoutSeconds: 3
      readinessProbe:
        httpGet:
          path: /todo/api/host
          port: 8080
          initialDelaySeconds: 10

```

```
    timeoutSeconds: 3
    dnsPolicy: ClusterFirst
...output omitted...
```

- 3.2. Use the **oc new-app** command with the **--dry-run** option to quickly check the completed edits.

Because the **oc new-app** command requires an OpenShift context to work, enter the **youruser-review-common** project.

The **--dry-run** option verifies that you passed values for all required parameters without a default value, and that the template definition is well-formed. This option does not verify that references to parameters follow the correct syntax, nor if any resource in the template object attribute has issues.

You can pass any value to the template parameters, because you are not creating any resources. The **oc new-app --dry-run** command displays an error message if a required parameter does not have a value, in either the parameter definition or in the command.

```
[student@workstation ~]$ oc project ${RHT_OCP4_DEV_USER}-review-common
Now using project "youruser-review-common" on server
"https://api.cluster.domain.example.com:6443".
...output omitted...
[student@workstation ~]$ oc new-app --dry-run -f ~/todoapp.yaml \
> -p PASSWORD=x -p HOSTNAME=y -p BACKEND=z
...output omitted...
--> Creating resources ...
  imagestream.image.openshift.io "backend" created (dry run)
  imagestream.image.openshift.io "tododb" created (dry run)
  configmap "todoapp" created (dry run)
  secret "tododb" created (dry run)
  buildconfig.build.openshift.io "backend" created (dry run)
  deploymentconfig.apps.openshift.io "backend" created (dry run)
  deploymentconfig.apps.openshift.io "tododb" created (dry run)
  service "backend" created (dry run)
  service "tododb" created (dry run)
  route.route.openshift.io "backend" created (dry run)
  deploymentconfig.apps.openshift.io "frontend" created (dry run)
  service "frontend" created (dry run)
  route.route.openshift.io "frontend" created (dry run)
--> Success (dry run)
...output omitted...
```

4. Create and review the template resource in the **youruser-review-common** project.

- 4.1. Create the template resource inside the **youruser-review-common** project.

```
[student@workstation ~]$ oc create -f ~/todoapp.yaml
template.template.openshift.io/todoapp created
```

- 4.2. Verify that the template resource is available, and defines all seven [MORE] parameters with the correct default values.

```
[student@workstation ~]$ oc process --parameters todoapp
NAME          ... VALUE
DATABASE_IMAGE ... registry.access.redhat.com/rhscl/mysql-57-rhel7
BACK_END_REPO  ... https://github.com/yourgituser/D0288-apps
BACK_END_CTXDIR ... todo-backend
BACK_END_BRANCH ... master
NPM_PROXY      ... http://nexus-common.apps.cluster.domain.example.com/
repository/nodejs
SECRET         ... [a-zA-Z0-9]{40}
PASSWORD
HOSTNAME
BACKEND
CLEAN_DATABASE ... false
```

5. Deploy the template to the ***youruser-review-multicontainer*** project. Verify that the front-end pod and database pod are ready and running, but after a few seconds the back-end pod returns a not ready state.

5.1. Create the ***youruser-review-multicontainer*** project.

```
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-review-multicontainer
Now using project "youruser-review-multicontainer" on server
"https://api.cluster.domain.example.com:6443".
...output omitted...
```

5.2. Grant service accounts from the new ***youruser-review-multicontainer*** project access to image streams from the ***youruser-review-common*** project.

```
[student@workstation ~]$ oc policy add-role-to-group \
> -n ${RHT_OCP4_DEV_USER}-review-common system:image-puller \
> system:serviceaccounts:${RHT_OCP4_DEV_USER}-review-multicontainer
clusterrole.rbac.authorization.k8s.io/system:image-puller added:
"system:serviceaccounts:youruser-review-multicontainer"
```

5.3. Deploy the To Do List application from the template, passing any required or optional attributes to fulfill the specifications of this review lab. You must provide values for the **PASSWORD**, **CLEAN_DATABASE**, **HOSTNAME**, and **BACKEND** parameters.

```
[student@workstation ~]$ oc new-app ${RHT_OCP4_DEV_USER}-review-common/todoapp \
> -p PASSWORD=redhat \
> -p CLEAN_DATABASE=false \
> -p HOSTNAME=todoui-${RHT_OCP4_DEV_USER}.${RHT_OCP4_WILDCARD_DOMAIN} \
> -p BACKEND=todoapi-${RHT_OCP4_DEV_USER}.${RHT_OCP4_WILDCARD_DOMAIN}
...output omitted...
--> Creating resources ...
imagestream.image.openshift.io "backend" created
imagestream.image.openshift.io "tododb" created
configmap "todoapp" created
secret "tododb" created
buildconfig.build.openshift.io "backend" created
deploymentconfig.apps.openshift.io "backend" created
deploymentconfig.apps.openshift.io "tododb" created
```

```
service "backend" created
service "tododb" created
route.route.openshift.io "backend" created
deploymentconfig.apps.openshift.io "frontend" created
service "frontend" created
route.route.openshift.io "frontend" created
--> Success
...output omitted...
```

- 5.4. Wait until the build finishes and all pods are ready and running. The back-end pod will look healthy during the first few seconds after it is deployed.

```
[student@workstation ~]$ oc logs -f bc/backend
...output omitted...
Push successful
[student@workstation ~]$ oc get pod
NAME          READY   STATUS    RESTARTS   AGE
backend-1-build  0/1     Completed  0          2m
backend-1-deploy 0/1     Completed  0          1m
backend-1-gcscq  1/1     Running   0          5s
frontend-1-deploy 0/1     Completed  0          2m
frontend-1-8qph0  1/1     Running   0          2m
tododb-1-deploy  0/1     Completed  0          2m
tododb-1-dvcqm  1/1     Running   0          2m
```

- 5.5. After a few seconds, the back-end pod restarts, because of the liveness probe.

```
[student@workstation ~]$ oc get pod
NAME          READY   STATUS    RESTARTS   AGE
...output omitted...
backend-1-gcscq  0/1     Running   1          51s
...output omitted...
```

- 5.6. Get the host name of the back-end route:

```
[student@workstation ~]$ oc get route
NAME      HOST/PORT
backend  todoapi-youruser.apps.cluster.domain.example.com  ...
frontend  todoui-youruser.apps.cluster.domain.example.com  ...
```

- 5.7. Use Curl to verify that the back-end pod cannot query the **Items** table. Use the HTTP API entry point of the liveness probe, and the host name from the previous step.

Notice that, if you wait too long to perform this command, then the back-end pod moves to a **CrashLoopBackOff** status, and you can no longer access the HTTP API. If this happens, continue to the next step.

You may get a router error if you try Curl during a restart of the pod. If you see this error, try again.

```
[student@workstation ~]$ curl -siw "\n" \
> todoapi-${RHT_OCP4_DEV_USER}.${RHT_OCP4_WILDCARD_DOMAIN}/todo/api/items-count
HTTP/1.1 500 Internal Server Error
...
>{"message":"ER_NO_SUCH_TABLE: Table 'todo.Item' doesn't exist"}
```

6. Populate the database and verify that the back-end pod is healthy. Open a web browser to verify that the application is working as expected. Do not make changes to any of the items in the database as this could affect the grading script.

- 6.1. Create a port-forwarding tunnel to the database pod.

```
[student@workstation ~]$ oc port-forward tododb-1-dvcqm 30306:3306
Forwarding from 127.0.0.1:30306 -> 3306
Forwarding from [::1]:30306 -> 3306
```

- 6.2. Open another terminal, and then use a MySQL client to run the `~/DO288/labs/review-multicontainer/todo.sql` script. Press **Ctrl+C** to terminate the port-forwarding tunnel.

```
[student@workstation ~]$ mysql -h127.0.0.1 -P30306 -utodoapp -predhat todo \
> < ~/DO288/labs/review-multicontainer/todo.sql
```

- 6.3. If you take too long to initialize the database, the back-end pod will enter the **CrashLoopBackOff** status. If this happens, redeploy the back-end pod and wait for the new pod to be ready and running.

```
[student@workstation ~]$ oc get pod
NAME          READY   STATUS      RESTARTS   AGE
...output omitted...
backend-1-gcscq   0/1     CrashLoopBackOff   6           6m
...output omitted...
[student@workstation ~]$ oc rollout latest dc/backend
deploymentconfig.apps.openshift.io/backend rolled out
[student@workstation ~]$ oc get pod
NAME          READY   STATUS      RESTARTS   AGE
...output omitted...
backend-2-lppd1   1/1     Running    0           1m
...output omitted...
```

- 6.4. Use Curl to verify that the To Do List application back end finds six items in the database. Use the same HTTP API entry point as the liveness probe.

```
[student@workstation ~]$ curl -siw "\n" \
> todoapi-${RHT_OCP4_DEV_USER}.${RHT_OCP4_WILDCARD_DOMAIN}/todo/api/items-count
HTTP/1.1 200 OK
...
>{"count":6}
```

- 6.5. Get the host name of the front-end route:

```
[student@workstation ~]$ oc get route
NAME      HOST/PORT
backend   todoapi-youruser.apps.cluster.domain.example.com ...
frontend  todoui-youruser.apps.cluster.domain.example.com ...
```

- 6.6. Open a web browser and access the To Do List front end, using the host name from the preceding step. Six items in the database display. Do not make any changes to these items as it could affect the grading script.

Id	Description	Done
1	Populate ex...	true
2	Create projec...	true
3	Deploy To ...	true
4	Create servi...	false
5	Add endpoint...	false
6	Verify that t...	false

Evaluation

As the **student** user on **workstation**, run the following command to confirm success on this exercise. Correct any reported failures and rerun the command until successful.

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

Finish

Run the **finish** command to signal you completed this exercise:

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

To restart all comprehensive review labs, use the **cleanup** command for each lab, in reverse order.

```
[student@workstation ~]$ lab review-multicontainer cleanup
[student@workstation ~]$ lab review-service cleanup
[student@workstation ~]$ lab review-dockerfile cleanup
```

This concludes the comprehensive review.

Appendix A

Creating a GitHub Account

Goal

Describe how to create a GitHub account for labs in the course.

Creating a GitHub Account

Objectives

After completing this section, you should be able to create a GitHub account and create public Git repositories for the labs in the course.

Creating a GitHub Account

You need a GitHub account to create one or more *public* Git repositories for the labs in this course. If you already have a GitHub account, you can skip the steps listed in this appendix.



Important

If you already have a GitHub account, ensure that you only create *public* Git repositories for the labs in this course. The lab grading scripts and instructions require unauthenticated access to clone the repository. The repositories must be accessible without providing passwords, SSH keys, or GPG keys.

To create a new GitHub account, perform the following steps:

1. Navigate to <https://github.com> using a web browser.
2. Enter the required details and then click **Sign up for GitHub**.

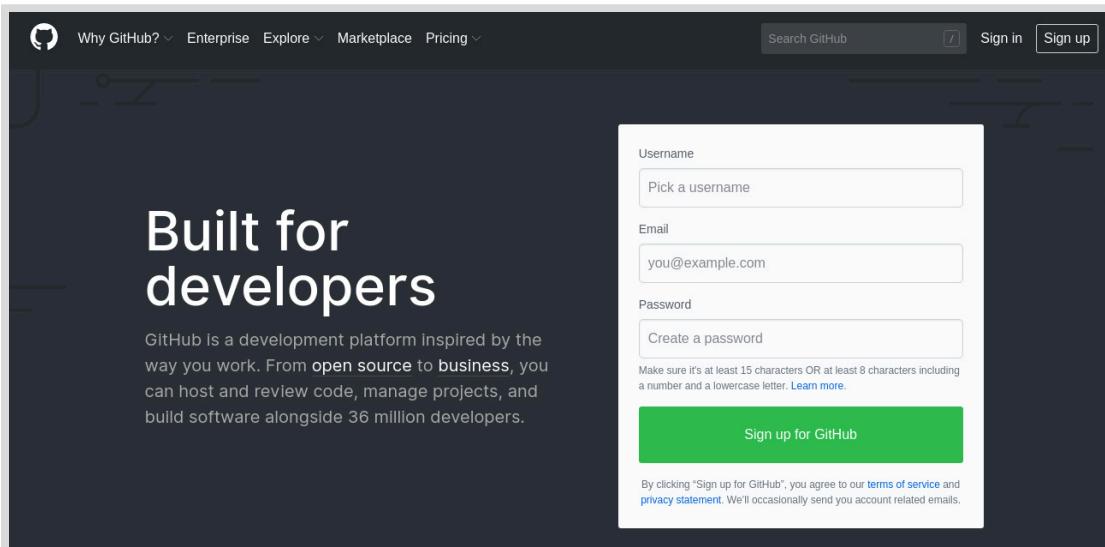


Figure A.1: Creating a GitHub account

3. You will receive an email with instructions on how to activate your GitHub account. Verify your email address and then sign in to the GitHub website using the username and password you provided during account creation.
4. After you have logged in to GitHub, you can create new Git repositories by clicking **New** in the **Repositories** pane on the left of the GitHub home page.

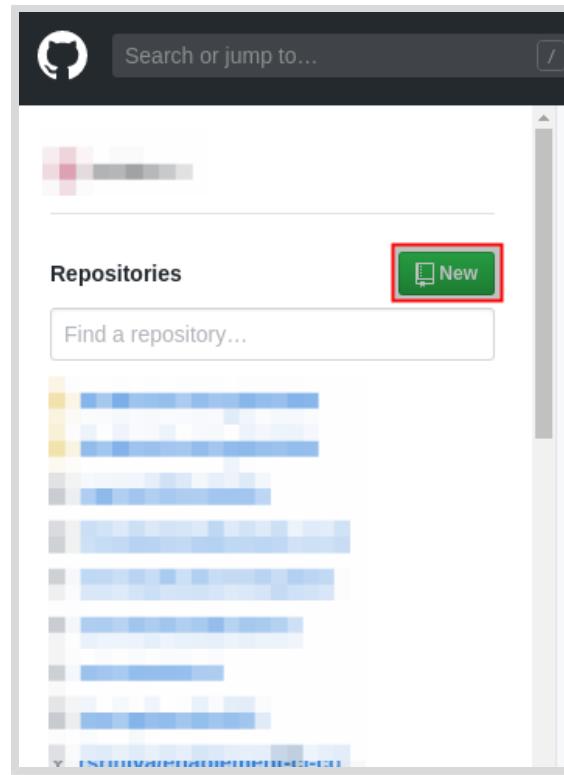


Figure A.2: Creating a new Git repository

Alternatively, click the plus icon (+) in the upper-right corner (to the right of the bell icon) and then click **New repository**.

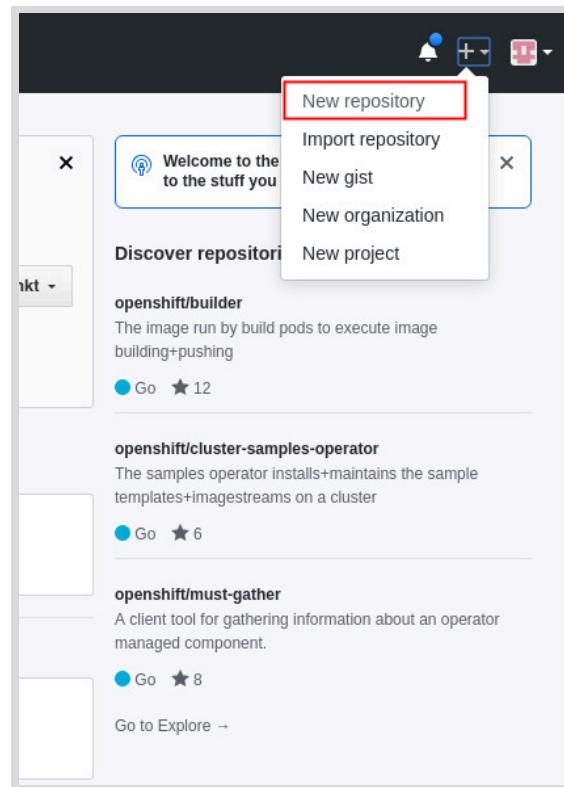


Figure A.3: Creating new Git repository



References

Signing up for a new GitHub account

<https://help.github.com/en/articles/signing-up-for-a-new-github-account>

Appendix B

Creating a Quay Account

Goal

Describe how to create a Quay account for labs in the course.

Creating a Quay Account

Objectives

After completing this section, you should be able to create a Quay account and create public container image repositories for the labs in the course.

Creating a Quay Account

You need a Quay account to create one or more *public* container image repositories for the labs in this course. If you already have a Quay account, you can skip the steps to create a new account listed in this appendix.



Important

If you already have a Quay account, ensure that you only create *public* container image repositories for the labs in this course. The lab grading scripts and instructions require unauthenticated access to pull container images from the repository.

To create a new Quay account, perform the following steps:

1. Navigate to <https://quay.io> using a web browser.
2. Click **Sign in** in the upper-right corner (next to the search bar).
3. On the **Sign in** page, you can log in using your Google or GitHub credentials (created in Appendix A).

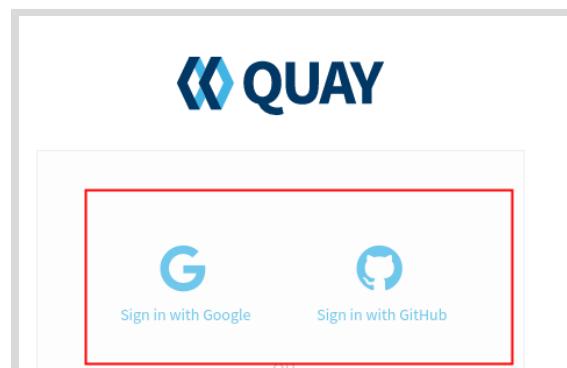


Figure B.1: Sign in using Google or GitHub credentials.

Alternatively, click **Create Account** to create a new account.

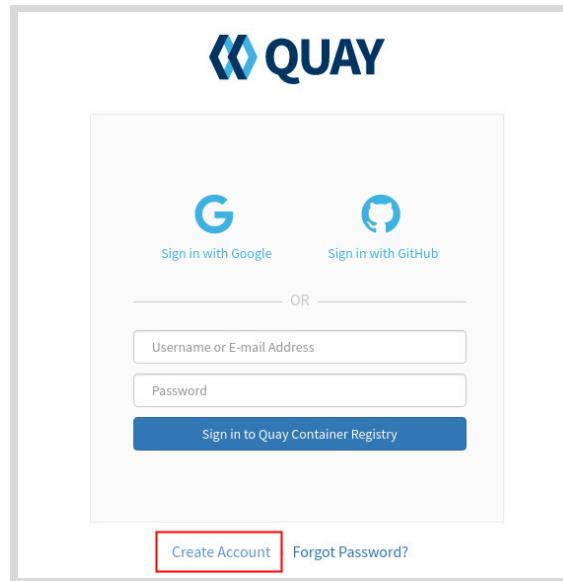


Figure B.2: Creating a new account

4. If you chose to skip the Google or GitHub log-in method and instead opted to create a new account, you will receive an email with instructions on how to activate your Quay account. Verify your email address and then sign in to the Quay website with the username and password you provided during account creation.
5. After you have logged in to Quay you can create new image repositories by clicking **Create New Repository** on the **Repositories** page.

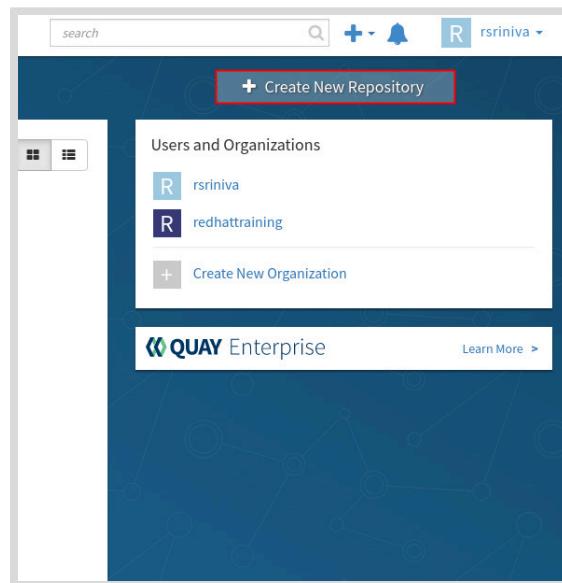


Figure B.3: Creating a new image repository

Alternatively, click the plus icon (+) in the upper-right corner (to the left of the bell icon), and then click **New Repository**.

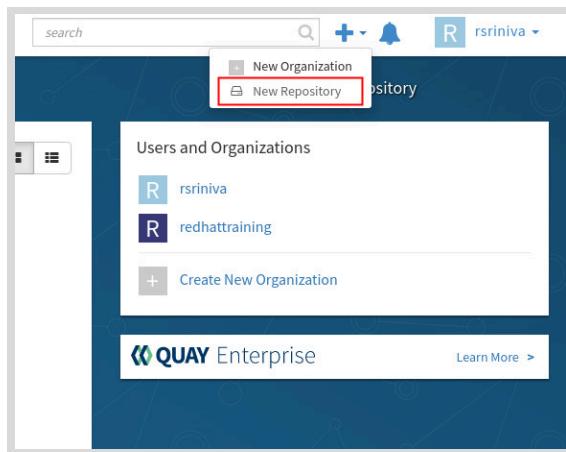


Figure B.4: Creating a new image repository



References

Getting Started with Quay.io

<https://docs.quay.io/solution/getting-started.html>

Appendix C

Useful Git Commands

Goal

Describe useful Git commands that are used for the labs in this course.

Git Commands

Objectives

After completing this section, you should be able to restart and redo exercises in this course. You should also be able to switch from one incomplete exercise to perform another, and later continue the previous exercise where you left off.

Working with Git Branches

This course uses a Git repository hosted on GitHub to store the application course code source code. At the beginning of the course, you create your own fork of this repository, which is also hosted on GitHub.

During this course, you work with a local copy of your fork, which you clone to the **workstation** VM. The term *origin* refers to the remote repository from which a local repository is cloned.

As you work through the exercises in the course, you use separate Git branches for each exercise. All changes you make to the source code happen in a new branch that you create only for that exercise. Never make any changes on the **master** branch.

A list of scenarios and the corresponding Git commands that you can use to work with branches, and to recover to a known good state are listed below.

Redoing an Exercise from Scratch

To redo an exercise from scratch after you have completed it, perform the following steps:

1. You commit and push all the changes in your local branch as part of performing the exercise. You finished the exercise by running its **finish** subcommand to clean up all resources:

```
[student@workstation ~]$ lab your-exercise finish
```

2. Change to your local clone of the **DO288-apps** repository and switch to the **master** branch:

```
[student@workstation ~]$ cd ~/DO288-apps
[student@workstation DO288-apps]$ git checkout master
```

3. Delete your local branch:

```
[student@workstation DO288-apps]$ git branch -d your-branch
```

4. Delete the remote branch on your personal GitHub account:

```
[student@workstation DO288-apps]$ git push origin --delete your-branch
```

5. Use the **start** subcommand to restart the exercise:

```
[student@workstation D0288-apps]$ cd ~
[student@workstation ~]$ lab your-exercise start
```

Abandoning a Partially Completed Exercise and Restarting it from Scratch

You may run into a scenario where you have partially completed a few steps in the exercise, and you want to abandon the current attempt, and restart it from scratch. Perform the following steps:

- Run the exercise's **finish** subcommand to clean up all resources.

```
[student@workstation ~]$ lab your-exercise finish
```

- Enter your local clone of the **D0288-apps** repository and discard any pending changes on the current branch using **git stash**:

```
[student@workstation ~]$ cd ~/D0288-apps
[student@workstation D0288-apps]$ git stash
```

- Switch to the **master** branch of your local repository:

```
[student@workstation D0288-apps]$ git checkout master
```

- Delete your local branch:

```
[student@workstation D0288-apps]$ git branch -d your-branch
```

- Delete the remote branch on your personal GitHub account:

```
[student@workstation D0288-apps]$ git push origin --delete your-branch
```

- You can now restart the exercise by running its **start** subcommand:

```
[student@workstation D0288-apps]$ cd ~
[student@workstation ~]$ lab your-exercise start
```

Switching to a Different Exercise from an Incomplete Exercise

You may run into a scenario where you have partially completed a few steps in an exercise, but you want to switch to a different exercise, and revisit the current exercise at a later time.

Avoid leaving too many exercises uncompleted to revisit later. These exercises tie up cloud resources and you may use up your allotted quota on the cloud provider and on the OpenShift cluster you share with other students. If you think it may be a while until you can go back to the current exercise, consider abandoning it and later restarting from scratch.

If you prefer to pause the current exercise and work on the next one, perform the following steps:

- Commit any pending changes in your local repository and push them to your personal GitHub account. You may want to record the step where you stopped the exercise:

```
[student@workstation ~]$ cd ~/D0288-apps  
[student@workstation D0288-apps]$ git commit -a -m 'Paused at step X.Y'  
[student@workstation D0288-apps]$ git push
```

2. Do not run the **finish** command of the original exercise. This is important to leave your existing OpenShift projects unchanged, so you can resume later.
3. Start the next exercise by running its **start** subcommand:

```
[student@workstation ~]$ lab your-exercise start
```

4. The next exercise switches to the **master** branch and optionally creates a new branch for its changes. This means the changes made to the original exercise in the original branch are left untouched.
5. Later, after you have completed the next exercise, and you want to go back to the original exercise, switch back to its branch:

```
[student@workstation ~]$ git checkout original-branch
```

Then you can continue with the original exercise at the step where you left off.



References

Git branch man page

<https://git-scm.com/docs/git-branch>

What is a Git branch?

<https://git-scm.com/book/en/v1/Git-Branching-What-a-Branch-Is>

Git Tools - Stashing

<https://git-scm.com/book/en/v1/Git-Tools-Stashing>