



Red Hat Training and Certification

Student Workbook (ROLE)

OCP 4.6 DO288

Red Hat OpenShift Development II: Containerizing Applications

Edition 2

Red Hat OpenShift Development II: Containerizing Applications



OCP 4.6 DO288

Red Hat OpenShift Development II: Containerizing Applications

Edition 2 20210818

Publication date 20210818

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

Editor: Sam Ffrench, 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, OpenShift, Fedora, Hibernate, Ansible, CloudForms, RHCA, RHCE, RHCSA, Ceph, and Gluster are trademarks or registered trademarks of Red Hat, Inc. or its subsidiaries in the United States and other countries.

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

Java® is a registered trademark of Oracle 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.

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

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

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

All other trademarks are the property of their respective owners.

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

Document Conventions	vii
Introduction	ix
Red Hat OpenShift Development II: Containerizing Applications	ix
Orientation to the Classroom Environment	x
Internationalization	xiii
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	20
Guided Exercise: Deploying an Application to an OpenShift Cluster	29
Managing Applications with the Web Console	36
Guided Exercise: Managing an Application with the Web Console	41
Managing Applications with the CLI	49
Guided Exercise: Managing an Application with the CLI	54
Lab: Deploying and Managing Applications on an OpenShift Cluster	63
Summary	71
2. Designing Containerized Applications for OpenShift	73
Selecting a Containerization Approach	74
Quiz: Selecting a Containerization Approach	78
Building Container Images with Advanced Containerfile Instructions	84
Guided Exercise: Building Container Images with Advanced Containerfile Instructions	92
Injecting Configuration Data into an Application	101
Guided Exercise: Injecting Configuration Data into an Application	109
Lab: Designing Containerized Applications for OpenShift	117
Summary	128
3. Publishing Enterprise Container Images	129
Managing Images in an Enterprise Registry	130
Guided Exercise: Using an Enterprise Registry	138
Allowing Access to the OpenShift Registry	144
Guided Exercise: Allowing Access to the OpenShift Registry	148
Creating Image Streams	152
Guided Exercise: Creating an Image Stream	159
Lab: Publishing Enterprise Container Images	163
Summary	172
4. Managing Builds on OpenShift	173
Describing the Red Hat OpenShift Build Process	174
Quiz: The OpenShift Build Process	178
Managing Application Builds	184
Guided Exercise: Managing Application Builds	188
Triggering Builds	194
Guided Exercise: Triggering Builds	197
Implementing Post-commit Build Hooks	203
Guided Exercise: Implementing Post-Commit Build Hooks	205
Lab: Building Applications for OpenShift	210
Summary	218
5. Customizing Source-to-Image Builds	219
Describing the Source-to-Image Architecture	220
Quiz: Describing the Source-to-Image Architecture	226
Customizing an Existing S2I Base Image	230
Guided Exercise: Customizing S2I Builds	233
Creating an S2I Base Image	238

Guided Exercise: Creating an S2I Base Image	244
Lab: Customizing Source-to-Image Builds	255
Summary	269
6. Deploying Multi-container Applications	271
Describing OpenShift Templates	272
Quiz: Describing OpenShift Templates	276
Creating a Helm Chart	278
Guided Exercise: Creating a Helm Chart	282
Customizing a Deployment with Kustomize	287
Guided Exercise: Customizing Deployments with Kustomize	291
Lab: Deploying Multi-container Applications	297
Summary	309
7. Managing Application Deployments	311
Monitoring Application Health	312
Guided Exercise: Activating Probes	318
Selecting the Appropriate Deployment Strategy	325
Guided Exercise: Implementing a Deployment Strategy	330
Managing Application Deployments with CLI Commands	338
Guided Exercise: Managing Application Deployments	343
Lab: Managing Application Deployments	350
Summary	361
8. Building Applications for OpenShift	363
Integrating External Services	364
Guided Exercise: Integrating an External Service	366
Containerizing Services	370
Guided Exercise: Containerizing Nexus as a Service	378
Deploying Cloud-Native Applications with JKube	388
Guided Exercise: Deploying an Application with JKube	394
Lab: Building Cloud-Native Applications for OpenShift	404
Summary	413
9. Comprehensive Review: Red Hat OpenShift Development II: Containerizing Applications	415
Comprehensive Review	416
Lab: Comprehensive Review	418
A. Creating a GitHub Account	433
Creating a GitHub Account	434
B. Creating a Quay Account	439
Creating a Quay Account	440
C. Useful Git Commands	443
Git Commands	444

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 II: 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

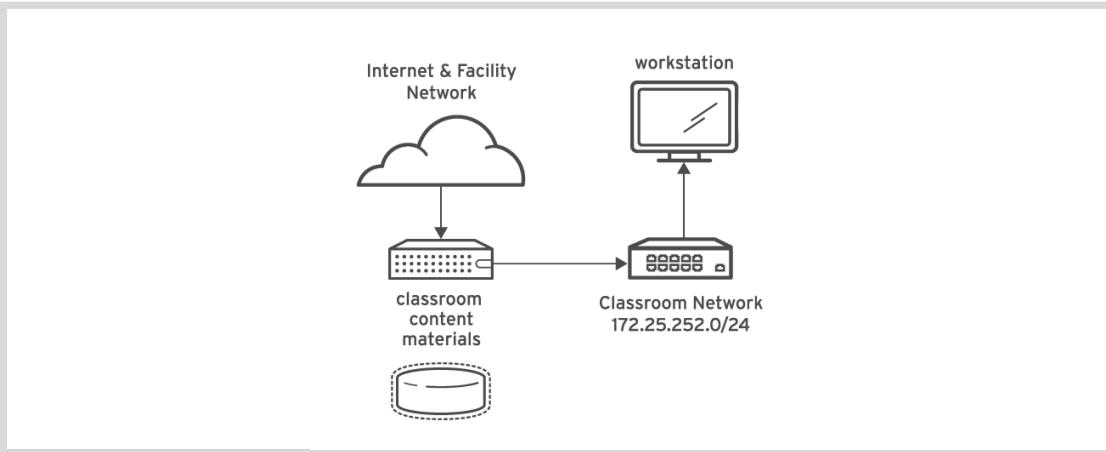


Figure O.1: Classroom environment

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
<code>content.example.com</code> , <code>materials.example.com</code> , <code>classroom.example.com</code>	172.25.252.254, 172.25.253.254, 172.25.254.254	Classroom utility server
<code>workstation.lab.example.com</code>	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.

Introduction

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

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

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



Important

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

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

Language Packs

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

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

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

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

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

Introduction

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

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



Important

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

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

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



References

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

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

Language Codes Reference



Note

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

Language Codes

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

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

Chapter 1

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.

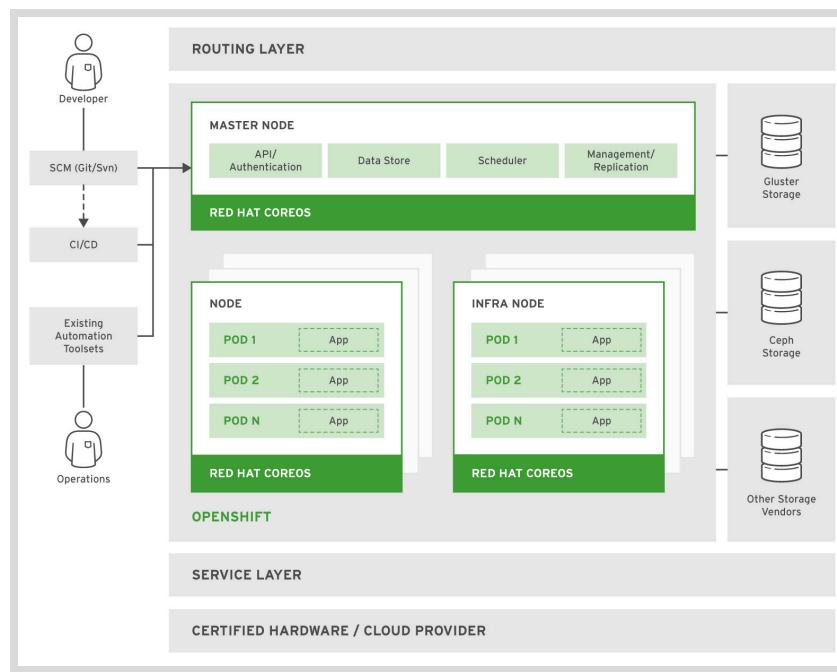


Figure 1.1: OpenShift 4 architecture

The following diagram illustrates the OpenShift Container Platform stack.

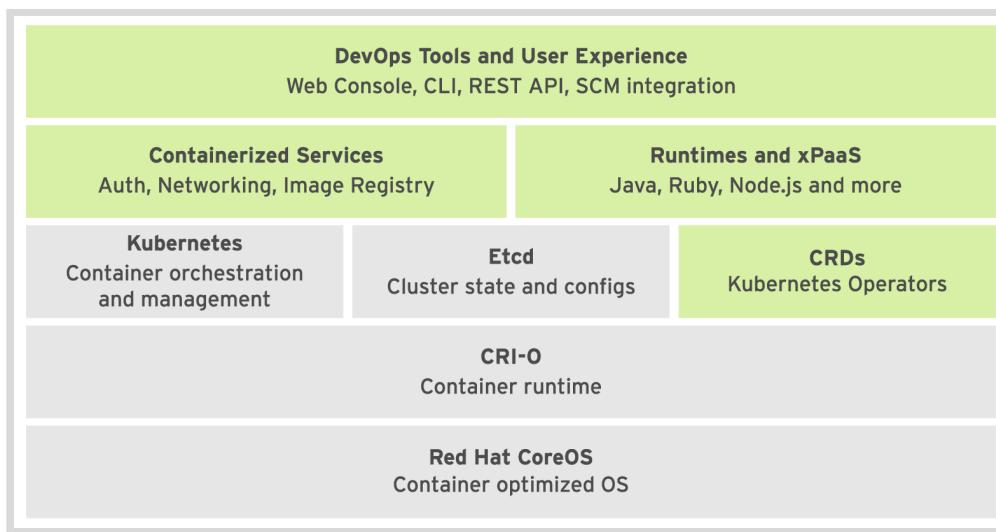


Figure 1.2: OpenShift component stack

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

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 RHOCP 4

RHOCP 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*.
- A personal access token from GitHub.

Instructions

You must perform all the following steps before you start any exercise.

► 1. Prepare your Github access token.

- 1.1. Navigate to <https://github.com> using a web browser and authenticate.
- 1.2. On the top of the page, click your profile icon, select the **Settings** menu, and then select **Developer settings** in the left pane of the page.

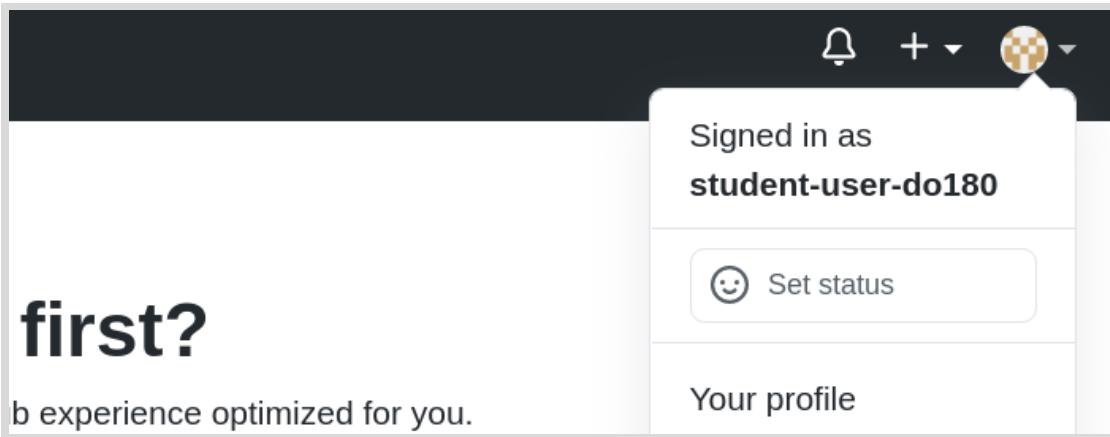


Figure 1.3: User Menu

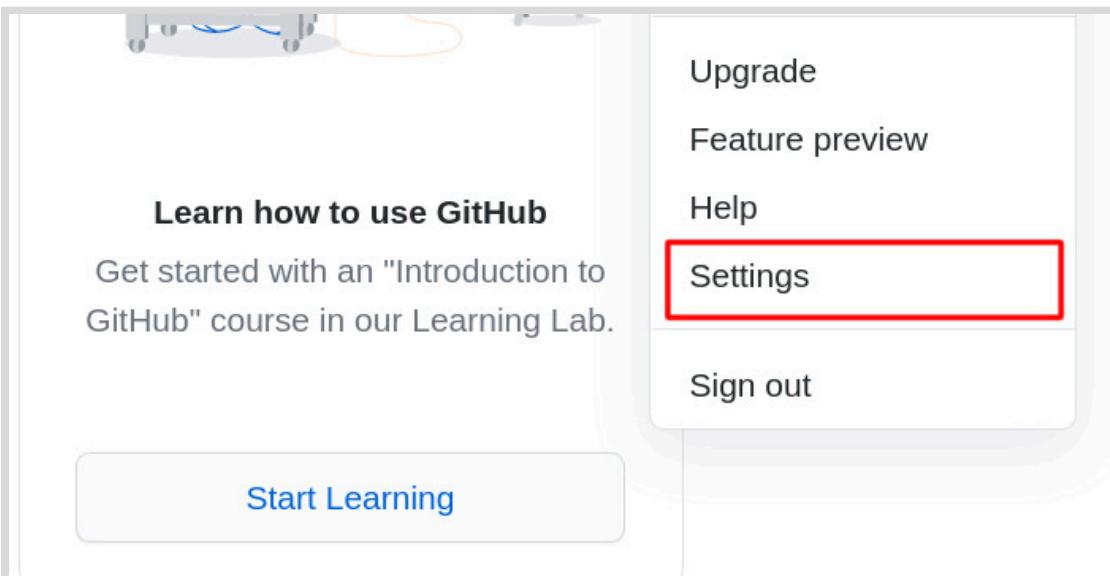


Figure 1.4: Settings Menu

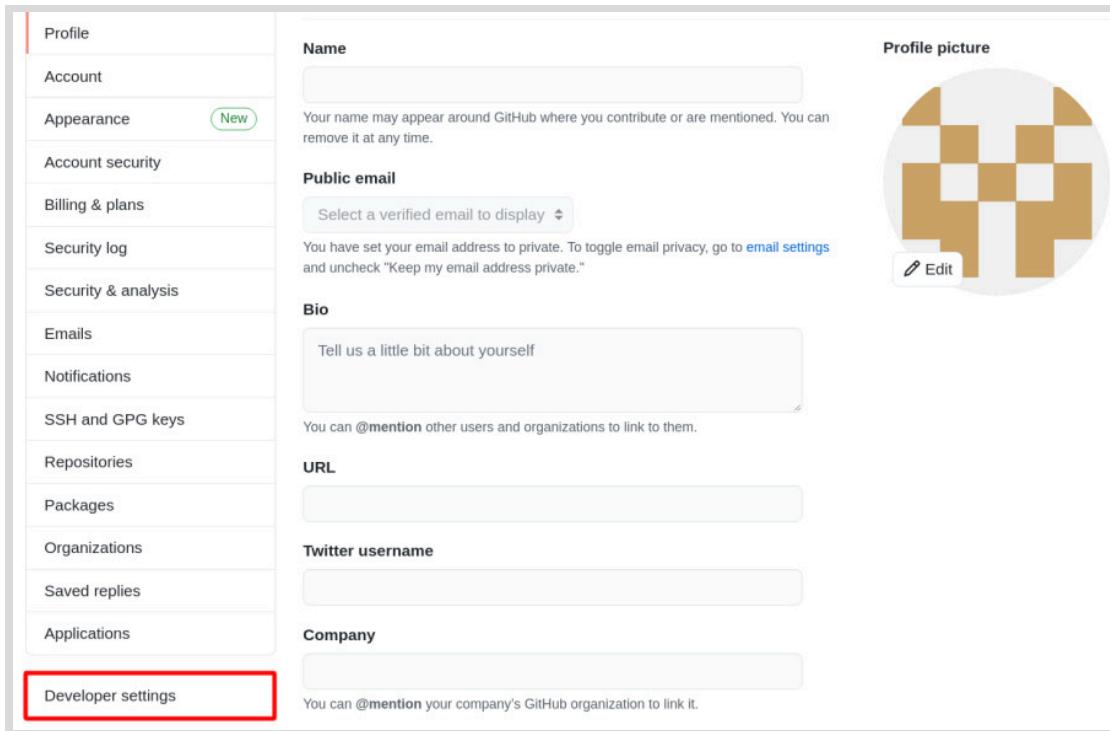


Figure 1.5: Developer settings

- 1.3. Select the Personal access token section on the left pane. On the next page, create your new token by clicking **Generate new token**, you are then prompted to enter your password.

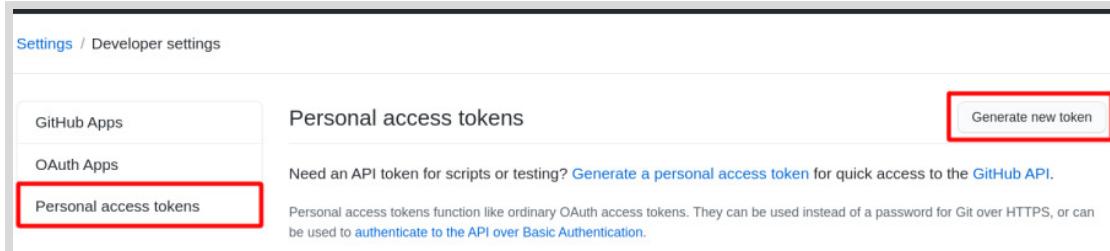


Figure 1.6: Personal access token pane

- 1.4. Write a short description about your new access token on the **Note** field.
- 1.5. Select the **public_repo** option and leave the other options unchecked. Create your new access token by clicking **Generate token**.

New personal access token

Personal access tokens function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to [authenticate to the API over Basic Authentication](#).

Note

Course DO280
What's this token for?

Select scopes

Scopes define the access for personal tokens. [Read more about OAuth scopes](#).

<input type="checkbox"/> repo	Full control of private repositories
<input type="checkbox"/> repo:status	Access commit status
<input type="checkbox"/> repo_deployment	Access deployment status
<input checked="" type="checkbox"/> public_repo	Access public repositories
<input type="checkbox"/> repo:invite	Access repository invitations
<input type="checkbox"/> security_events	Read and write security events

Figure 1.7: Personal access token configuration

- 1.6. Your new personal access token is displayed in the output. Using your preferred text editor, create a new file in student's home directory named `token` and ensure you paste in your generated personal access token.

Important

The personal access token can not be displayed again in GitHub. It is important to save the token somewhere safe, or you will have to create a new one in case you need the personal access token again.

Personal access tokens

Generate new token Revoke all

Tokens you have generated that can be used to access the [GitHub API](#).

Make sure to copy your new personal access token now. You won't be able to see it again!

✓ ghp_kgYGzWcGE1CrdovkzuzeLTWvYY6eBX2l0vck [Copy](#) [Delete](#)

Personal access tokens function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to [authenticate to the API over Basic Authentication](#).

Figure 1.8: Generated access token

**Important**

During this course, if you are prompted for a password while using Git operations on the command line, use your access token as the password.

- 2. 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:

OpenShift Details		
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	

workstation	active	ACTION	OPEN CONSOLE
classroom	active	ACTION	OPEN CONSOLE

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
```

- 2.1. 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 identifiers 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.

- 2.2. 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...

- 2.3. 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.

- 2.4. 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...
```

- 2.5. 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.
```

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

```
...output omitted...

Verifying your cluster configuration...

...output omitted...
```

- 2.7. 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.
```

- 2.8. 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.
- 3. Before starting any exercise, you need to fork this course's sample applications into your personal GitHub account. Perform the following steps:
- 3.1. 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.

The screenshot shows the GitHub interface for the repository 'RedHatTraining / DO288-apps'. At the top, there are navigation links for 'Why GitHub?', 'Enterprise', 'Explore', 'Marketplace', and 'Pricing'. A search bar and 'Sign in' and 'Sign up' buttons are also present. Below the header, the repository name 'RedHatTraining / DO288-apps' is displayed, along with statistics: 22 Watch, 0 Star, 3 Fork, and 3 Issues. A navigation bar below the repository name includes links for 'Code', 'Issues 0', 'Pull requests 0', 'Projects 0', 'Security', and 'Insights'.

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

The screenshot shows the GitHub sign-in page titled 'Sign in to GitHub'. It features a form with fields for 'Username or email address' containing 'yourgituser' and 'Password' (represented by a redacted field). There is a 'Forgot password?' link and a green 'Sign in' button.

- 3.3. Return to the RedHatTraining/D0288-apps repository and click **Fork** in the upper-right corner.

The screenshot shows the GitHub interface for the 'RedHatTraining / DO288-apps' repository. The 'Fork' button in the top right corner is highlighted. The repository statistics remain the same: 22 Watch, 0 Star, 3 Fork, and 3 Issues.

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

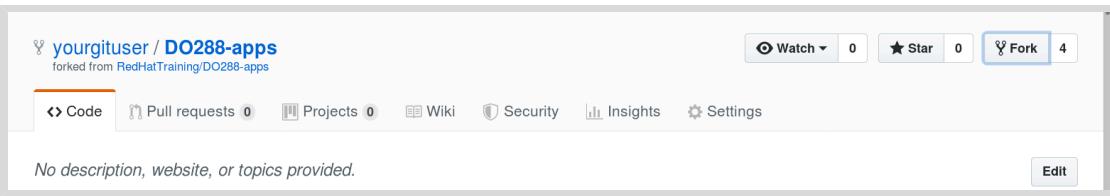
The screenshot shows a modal window titled 'Fork DO288-apps' asking 'Where should we fork DO288-apps?'. It lists a single option: 'yourgituser'. The background shows the original repository page.



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 D0288-apps when your fork the repository.

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



- 4. 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:
- 4.1. 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...
```

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

```
[student@workstation ~]$ cd DO288-apps
[student@workstation DO288-apps]$ git status
# On branch main

nothing to commit, working directory clean
```

- 4.3. 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 ~]$
```

- 5. 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 **main** 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 **main** 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*.

- 6. 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 4.6 the `oc new-app` command now produces Deployment resources instead of DeploymentConfig resources by default. This version of DO288 uses DeploymentConfigs only when it is required. To create DeploymentConfig resources, you can pass the `--as-deployment-config` flag when invoking `oc new-app`. For more information, see Understanding Deployments and DeploymentConfigs [<https://docs.openshift.com/container-platform/4.6/applications/deployments/what-deployments-are.html>].

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 resource 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 \
  https://github.com/RedHatTraining/D0288/tree/main/apps/apache-httpd
```

If the URL references a Git repository, you can optionally specify a branch name by using the octothorpe (#) delimiter. For example, the following URL indicates to use the branch named `my-branch`: `https://github.com/RedHatTraining/D0288-apps#my-branch`.

- If you have an existing containerized application that you want to deploy to OpenShift, use the `oc new-app` command to create a deployment resource 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 --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
<code>--as-deployment-config</code>	Configures the <code>oc new-app</code> to create a <code>DeploymentConfig</code> resource instead of a <code>Deployment</code> .
<code>--image-stream -i</code>	Provides the image stream to be used as either the S2I builder image for an S2I build or to deploy a container image.
<code>--strategy</code>	<code>docker</code> or <code>pipeline</code> or <code>source</code>
<code>--code</code>	Provides the URL to a Git repository to be used as input to an S2I build.
<code>--docker-image</code>	Provides the URL to a container image to be deployed.
<code>--dry-run</code>	set to <code>true</code> to show the result of the operation without performing it.
<code>--context-dir</code>	Provides the path to a directory to treat as the root.

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 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 php~http://gitserver.example.com/mygitrepo
```

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

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

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

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

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

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

```
[user@host ~]$ oc new-app \
--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` 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 resource 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 \
mysql -e MYSQL_USER=user -e MYSQL_PASSWORD=pass \
-e 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:

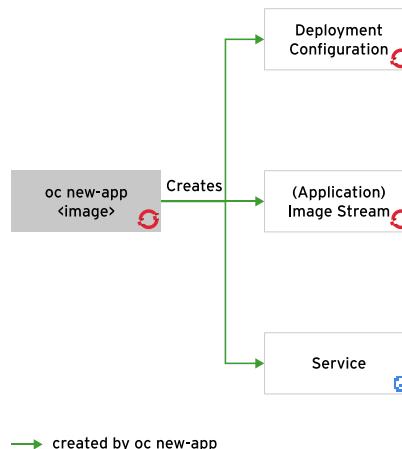


Figure 1.15: Resources created by the `oc new-app` command

The following command creates an application from source code in the PHP programming language, using a deployment config:

```
[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		
NAME	TYPE	FROM	STATUS	STARTED	DURATION	
builds/hello-1	Source	Git@3a0af02	Complete	About an hour ago	1m16s	2
NAME	DOCKER REPO		TAGS		UPDATED	
is/hello	docker-registry.default.svc:5000/test/hello		3			
NAME	REVISION	DESIRED	CURRENT	TRIGGERED BY		
dc/hello	1	1	1	config,image(hello:latest)	4	
NAME	DESIRED	CURRENT	READY	AGE		
rc/hello-1	1	1	1	3m	5	
NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE		
svc/hello	172.30.2.186	<none>	8080/TCP	2m31s	6	
NAME	READY	STATUS	RESTARTS	AGE		
po/hello-1-build	0/1	Completed	0	2m11s	7	
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.

The following command creates an application from source code in the PHP programming language, using a deployment instead of a deployment configuration:

```
[user@host ~]$ oc new-app \
--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	READY	STATUS	RESTARTS	AGE
pod/hello-1-build	0/1	Completed	0	2m11s ①
pod/hello-57f548f776-q86pg	1/1	Running	0	1m23s ②
NAME	READY	UP-TO-DATE	AVAILABLE	AGE
deployment.apps/hello	1/1	1	1	62s ③
NAME	DESIRED	CURRENT	READY	AGE
replicaset.apps/hello-57f548f776	1	1	1	3m ④
replicaset.apps/hello-875348fa8e	0	0	0	4m
NAME	TYPE	FROM	LATEST	
buildconfig.build.openshift.io/hello	Source	Git	3 ⑤	
NAME	TYPE	FROM	STATUS	STARTED
DURATION				
build.build.openshift.io/hello-1	Source	Git@3a0af02	Complete	About an hour ago ⑥ 1m16s
NAME	IMAGE REPOSITORY			
TAGS	UPDATED			
imagestream.image.openshift.io/hello	docker-registry.default.svc:5000/test/hello	latest	46 seconds ago ⑦	

- ① 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.
- ② The application pod created by the first deployment.
- ③ The deployment configuration created by the `oc new-app` command.
- ④ The replication controller configuration created by the first deployment. Subsequent deployments might also create deployer pods.
- ⑤ The build configuration created by the `oc new-app` command.
- ⑥ The first build is triggered by the `oc new-app` command.
- ⑦ The image stream created by the `oc new-app` command. It points to the container image created by the S2I process.

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 \
--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 \
-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 4.6 at
https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6/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
```

Instructions

► 1. Inspect the Dockerfile for the sample application.

- 1.1. Enter your local clone of the `D0288-apps` Git repository and checkout the `main` 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 main
...output omitted...
```

- 1.2. 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.
```

- 1.3. 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 --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
deployment.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 temp.builder.openshift.io/youruser-docker-build/echo-1:... 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

deployment/echo deploys istag/echo:latest <-
bc/echo docker builds https://github.com/youruser/D0288-apps#docker-build on
istag/ubi:8.0
deployment #2 running for 20 minutes - 1 pod
deployment #1 deployed 21 minutes ago
...output omitted...
```

The first deployment is for the builder pod. The second deployment is your running application pod.

- 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-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-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
...
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:

```
[student@workstation D0288-apps]$ oc describe deployment echo
Name:           echo
...
Labels:         app=echo
Annotations:   deployment.kubernetes.io/revision: 2
               image.openshift.io/triggers: ①
               [{"from":{"kind":"ImageStreamTag","name":"echo:latest"}, "imagePath":"spec.template.spec.containers[?(.name==\"echo
\")]".image"}]
...
Pod Template:
...
Containers:
  echo:
    Image:      image-registry.openshift-image-registry.svc:5000/youruser-docker-build/echo@sha256:5bbf...ef0b ②
...
Deployment #1 (latest):
...
output omitted...
```

- ① The deployment 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 deploy istag/echo:latest <-
bc/echo docker builds https://github.com/youruser/D0288-apps#docker-build on
istag/ubi:8.0
deployment #3 running for 43 seconds - 1 pod
deployment #2 deployed 26 minutes ago
deployment #1 deployed 27 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-2-build 0/1     Completed  0          1m
echo-p1hg     1/1     Running   0          1m
...output omitted...
```

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-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
deployment.apps "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:

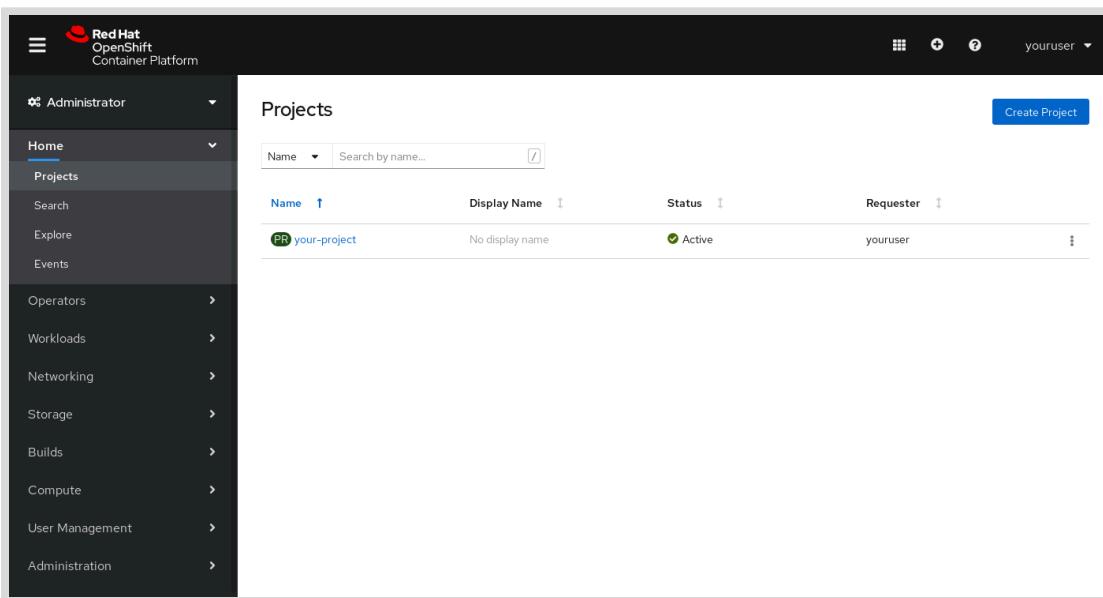
- Deploy an application from a binary image and manage its resources.
- View pod and build logs.
- Edit resource definitions.

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. For clusters with a single control plane node, this is usually an HTTPS URL to the public host name of that node.

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.



The screenshot shows the Red Hat OpenShift Container Platform web console. At the top, there is a navigation bar with the Red Hat logo, the text "Red Hat OpenShift Container Platform", and a user dropdown set to "youruser". Below the navigation bar is a sidebar on the left with a dark background and white text. The sidebar has a "Administrator" dropdown, followed by a "Home" dropdown which is currently selected. Under "Home", there are links for "Projects", "Search", "Explore", and "Events". Further down the sidebar are links for "Operators", "Workloads", "Networking", "Storage", "Builds", "Compute", "User Management", and "Administration", each preceded by a right-pointing arrow. To the right of the sidebar is the main content area. The title "Projects" is at the top of the content area. Below it is a search bar with the placeholder "Search by name..." and a magnifying glass icon. A blue button labeled "Create Project" is located in the top right corner of the content area. The main content area displays a table with four columns: "Name", "Display Name", "Status", and "Requester". The first row in the table shows a project named "your-project" with a "No display name", "Active" status, and requester "youruser".

Figure 1.16: 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.

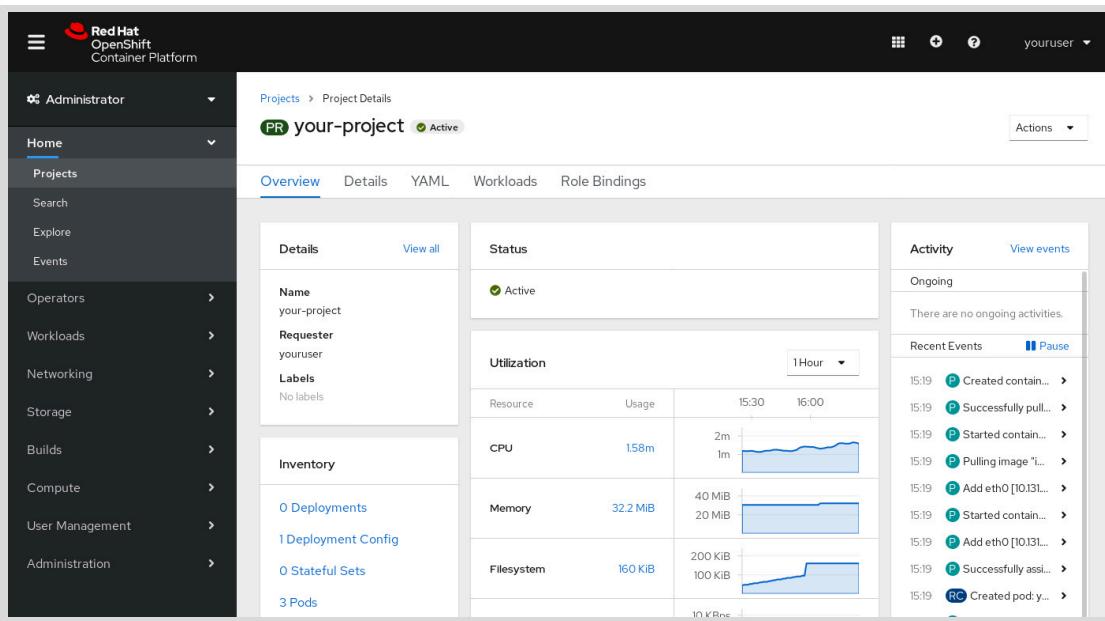


Figure 1.17: Web console project overview

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.

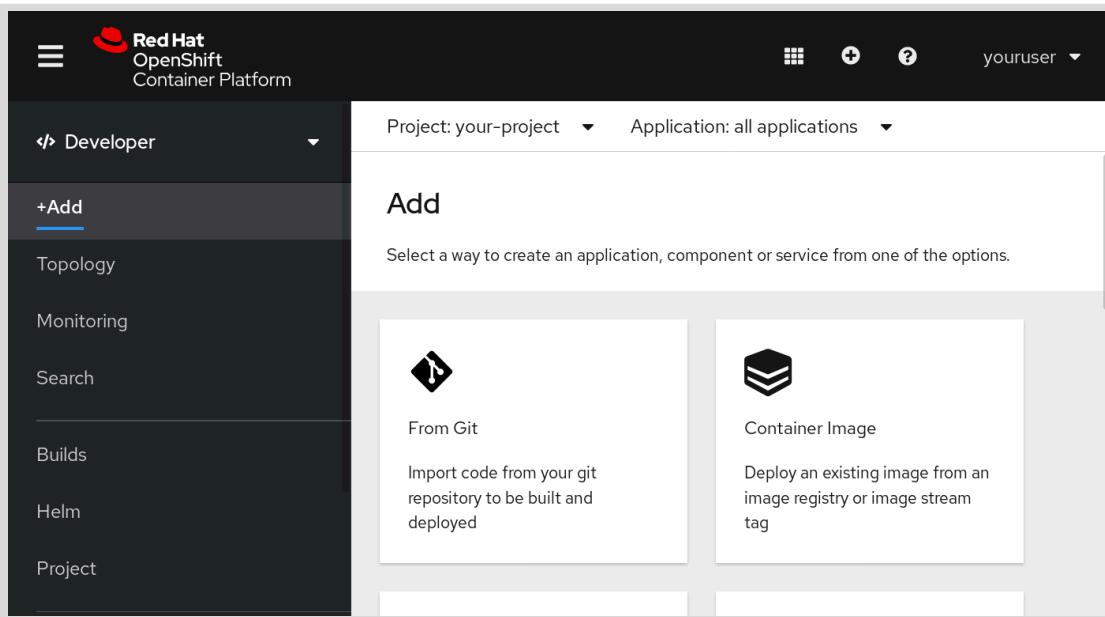


Figure 1.18: Actions available from the +Add section

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 have the deployment label with the name of the deployment.

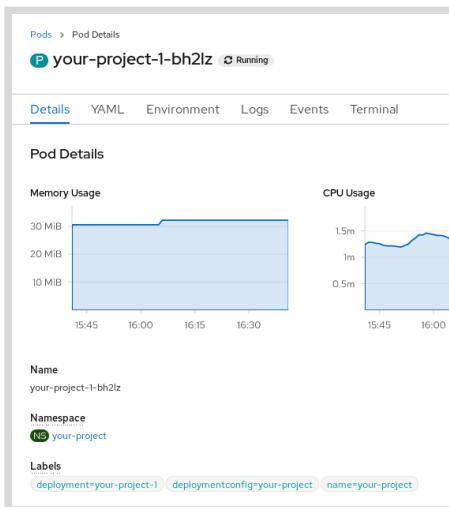


Figure 1.19: Labels at the bottom of a pod details page

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' (status: Complete). The page has tabs for 'Details', 'YAML', 'Environment', 'Logs' (which is selected), and 'Events'. Below the tabs, a message says 'Log stream ended.' with a timestamp '56 lines'. The log content is a multi-line text showing the build process, including copying config, writing manifest, storing signatures, pushing the image to the registry, and successfully pushing the image. At the bottom right of the log area are 'Download' and 'Expand' buttons.

```

Log stream ended.
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 sha256:02cb2f8840727e51a3c232637b7496e15a9a5671885a64f0d5e4a9dc2c1674fd
Copying blob sha256:9e7a6dc796f0a75c560158a9f9e30fb8b5a90cb53edce9ffbd57784064de39
Copying blob sha256:fc5b206e9329a1674dd9e8cfbee45c9be2bdd0dcbabb3c6bb67a2f22cfcf2a
Copying blob sha256:e7021e0589e97471d999c4265b7c8e64da328e48f116b5f260353b2e0a2ad5373
Copying blob sha256:a0a629e11c896fb5c92c3ede13fa50843042f978963a2a9e31288d1a7d5489
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 1.20: 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 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 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 provides significantly more functionality, including actions to customize various aspects of a Deployment, 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 4.6 at https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6/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
```

Instructions

- 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
```

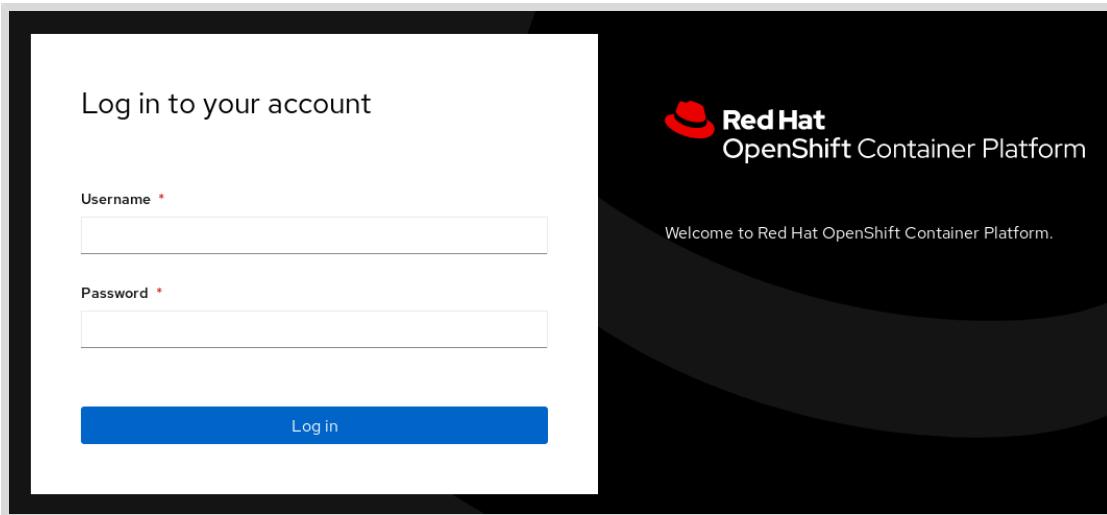
Another way to find the host name of your OpenShift web console is inspecting the routes on the `openshift-console` project. You must be logged into OpenShift before.

```
[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 ...
```

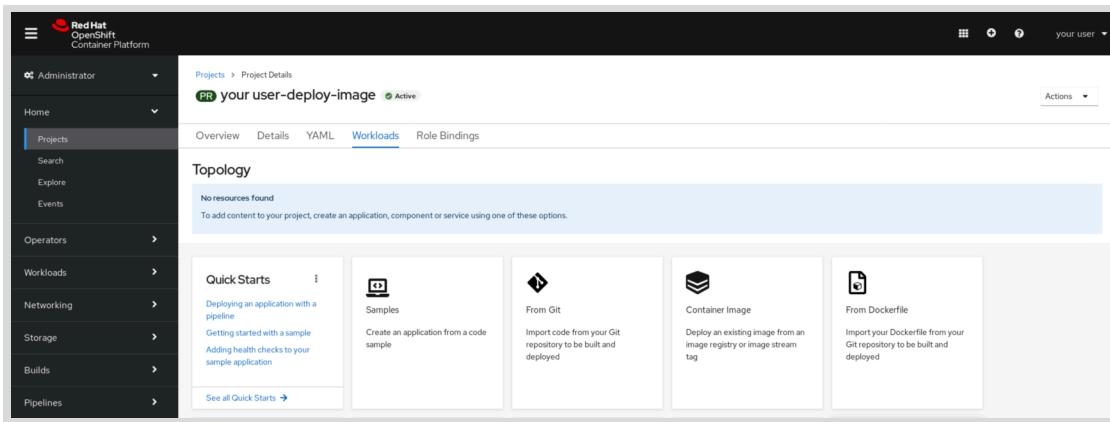
**Note**

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.

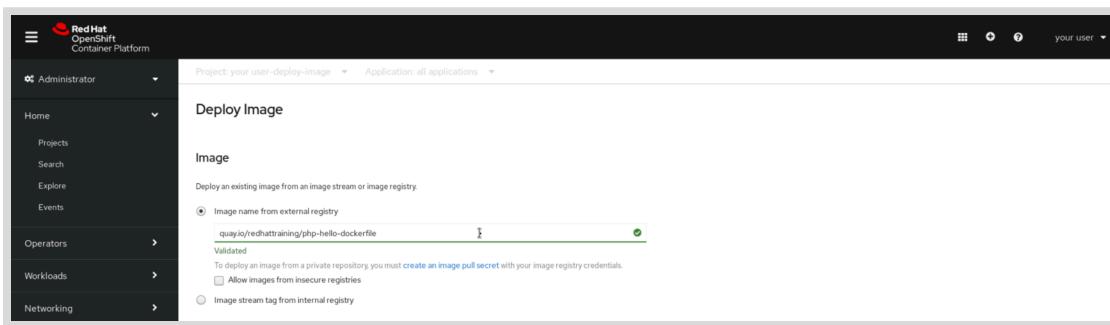
- 1.2. Open a web browser and navigate to `https://console-openshift-console.apps.cluster.domain.example.com` to access the OpenShift web console. Replace `apps.cluster.domain.example.com` with the value you got from the previous step. You should see the login page for the web console.
- 1.3. Log in 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.



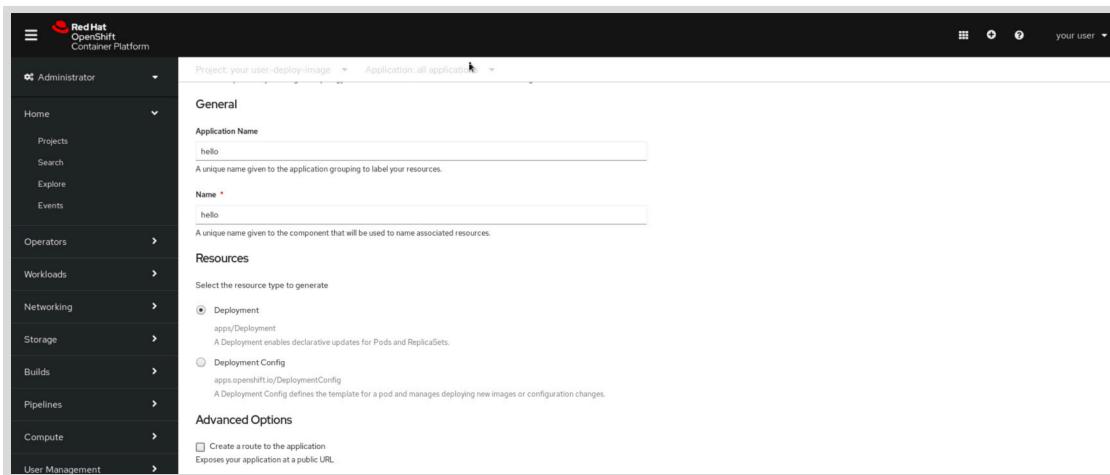
- 1.4. Navigate to the Administrator perspective, then select **Home** → **Projects** from the left side menu. First time users are placed in that page initially, Click **Create Project**. In the **Create Project** dialog box, enter `youruser-deploy-image` in the **Name** field. Replace `youruser` with 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 **Workloads** tab, click the **Container Image** button.



- 2.2. 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.

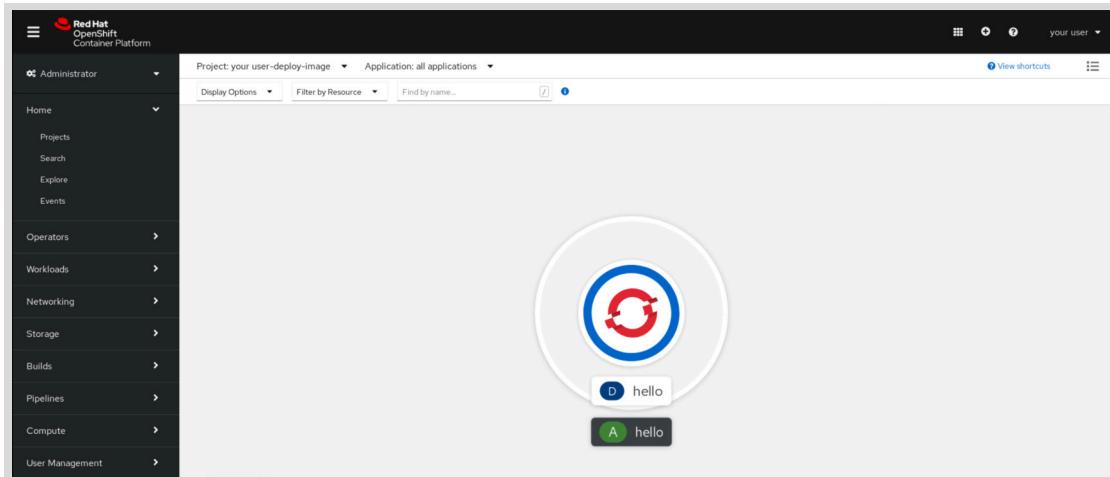


- 2.3. Scroll down to see information about the image and replace `php-hello-dockerfile` with `hello` in the **Name** and **Application Name** fields.

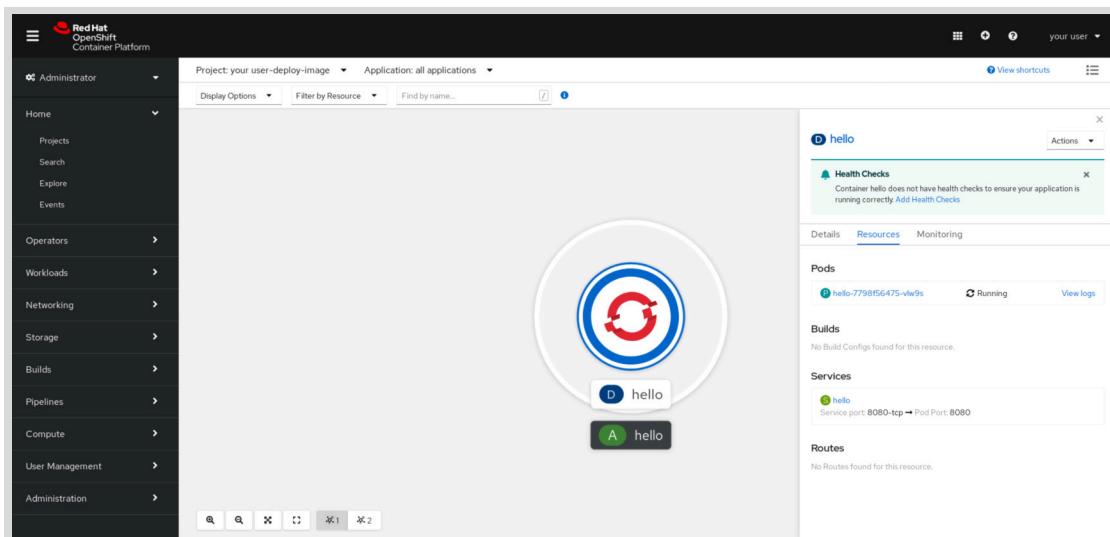


Uncheck the **Create a route to the application** option in the **Advanced Options** section at the bottom of the page.

- 2.4. 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.

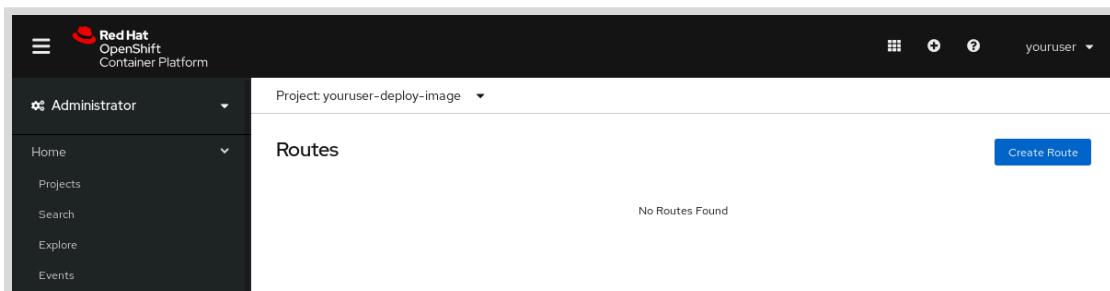


- 2.5. Click the icon above the deployment name to expand the deployment details and to view the number of running pods. Wait until the overview page displays a successful deployment with one pod:



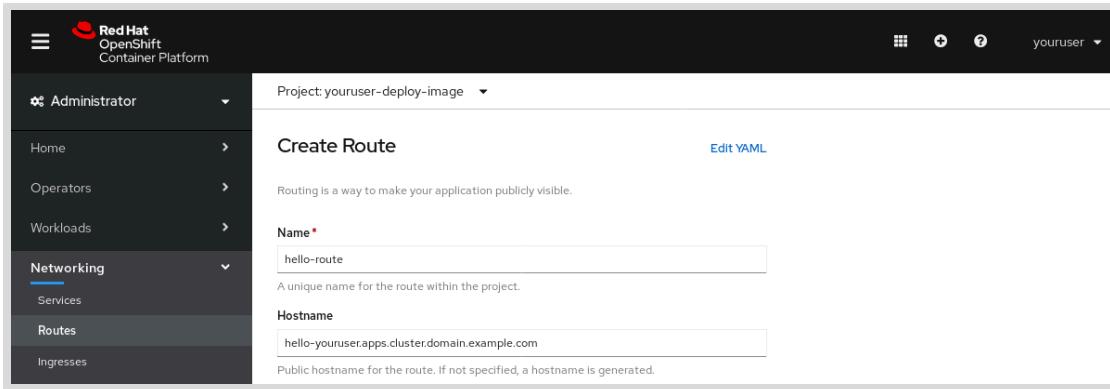
- 2.6. In the navigation bar, click **Networking → Routes**.

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

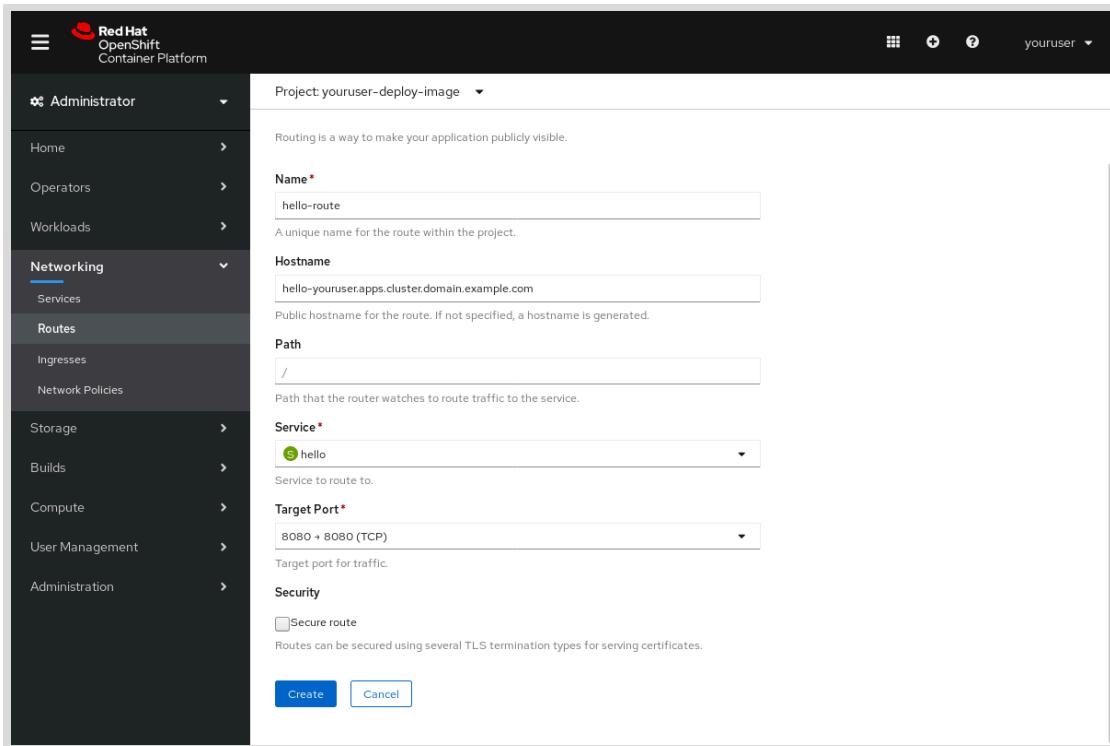


Enter **hello-route** in the **Name** field. Enter **hello-youruser.apps.cluster.domain.example.com** in the **Hostname** field.

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.



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**.



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

The screenshot shows the Red Hat OpenShift Container Platform web interface. The left sidebar is titled 'Administrator' and includes sections for Home, Operators, Workloads, Networking (selected), Services, Routes (selected), Ingresses, Network Policies, Storage, and Builds. The main content area is titled 'Project: youruser-deploy-image' and shows 'Routes > Route Details'. A route named 'hello-route' is listed with the status 'Accepted'. The 'Details' tab is selected. Route details include: Name: hello-route, Namespace: youruser-deploy-image, Labels: app=hello, app.kubernetes.io/component=hello, app.kubernetes.io/instance=hello, app.kubernetes.io/part-of=hello, app.openshift.io/runtime-version=latest, Location: http://hello-youruser.apps.cluster.domain.example.com, Status: Accepted, Host: hello-youruser.apps.cluster.domain.example.com, and Path: -.

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.

The screenshot shows the Red Hat OpenShift Container Platform web interface. The left sidebar is titled 'Administrator' and includes sections for Home, Operators, Workloads (selected), and Pods. The main content area is titled 'Project: your user-deploy-image' and shows 'Pods'. A single pod named 'hello-7798f56475-vlw9s' is listed with a status of 'Running'. The pod has 1 ready container, 0 restarts, and was created on Jun 8, 2:07 pm.

Click the application pod name, such as `hello-7798f56475-vlw9s` 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.

The screenshot shows the Red Hat OpenShift Container Platform web interface. The left sidebar is titled 'Administrator' and includes sections for Home, Projects, Search, Explore, Events, Operators, Workloads (selected), and Pods. The main content area is titled 'Project: your user-deploy-image' and shows 'Pods'. A pod named 'hello-7798f56475-vlw9s' is selected. The 'Logs' tab is selected, showing 4 lines of log output:

```
[08-Jun-2021 18:07:23] NOTICE: [pool www] 'user' directive is ignored when FPM is not running as root
[08-Jun-2021 18:07:23] NOTICE: [pool www] 'group' directive is ignored when FPM is not running as root
AH00058: httpd: could not reliably determine the server's fully qualified domain name, using 10.131.0.245. Set the 'ServerName' directive globally to suppress this message
```

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.

Chapter1 | Deploying and Managing Applications on an OpenShift Cluster

```
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.

```

1  kind: Pod
2  apiVersion: v1
3  metadata:
4    generateName: hello-7798f56475-
5    annotations:
6      k8s.v1.cni.cncf.io/network-status: |-
7        [
8          {
9            "name": "*",
10           "interface": "eth0",
11           "ip": "10.131.0.245"
12         }
13       "default": true,
14       "dns": {}
15     ]
16   k8s.v1.cni.cncf.io/networks-status: |-
17     [
18       {
19         "name": "*",
20         "interface": "eth0",
21         "ips": [
22           "10.131.0.245"
23         ]
24       }
25     ]
26   kubernetes.io/limit-ranger: >
27     [{"type": "Resource", "limits": {"cpu": "100m", "memory": "128Mi"}, "warnings": []}]

```

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** to view the **Deployment** page. Click **hello** to view the deployment details page.
In the upper-right corner, click **Actions** → **Delete Deployment**. In the confirmation dialog box, leave the checkbox checked and click **Delete** to delete the deployment.
- 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-zvjhb:/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_MIRROR` 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_MIRROR` 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 4.6 at

https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6/html-single/images/index#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
```

Instructions

► 1. Review the Quotes application source code.

- 1.1. Enter your local clone of the D0288-apps Git repository and check out the `main` 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 main
...output omitted...
```

- 1.2. 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.

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

- 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": "build.openshift.io/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:7.3-ubi8",
      }
    }
  }
...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, and omits the service and image stream resources:

```
...output omitted...
{
  "apiVersion": "apps/v1",
  "kind": "Deployment", ❶
...output omitted...
  "containers": [
...output omitted...
    "name": "mysql",
    "ports": [
      {
        "containerPort": 3306
      }
    ]
...output omitted...
    "volumes": [
      ...
    ]
  ]
}
```

```

{
    "emptyDir": {}, ❷
    "name": "data"
...output omitted...
    "triggers": [
...output omitted...
        "env": {
...output omitted...
            "image": "mysql:5.7", ❸
...output omitted...

```

- ❶ A deployment deploys a MySQL database container. The deployment and associated resources are the same as those that would be created by the oc new-app command on the database image.
- ❷ The database is not backed by persistent storage. All data would be lost if the database pod is restarted.
- ❸ 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", ❶
    "displayName": "Git Repository URL",
    "description": "The URL of the repository with your application source
code.",
...output omitted...
{
    "name": "DATABASE_USER", ❷
    "displayName": "Database User",
...output omitted...

```

- ❶ Parameters provide application-specific configurations, such as the Git repository URL.
- ❷ 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 D0288-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 --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-7d76ff58f8-6j2gx   1/1     Running    0          28s
quotesdb-6b7ffcc649-ds1pq   1/1     Running    0          80s
```

Make a note of the application and database pod names (quotesapi-7d76ff58f8-6j2gx and quotesdb-6b7ffcc649-ds1pq 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-6b7ffcc649-ds1pq | grep IP
IP: 10.129.0.142
```

- 5.2. Verify the database pod login credentials:

```
[student@workstation D0288-apps]$ oc describe pod quotesapi-7d76ff58f8-6j2gx \
| grep -A 4 Environment
Environment:
  MYSQL_USER: <set to the key 'database-user' in secret 'quotesapi'>
  MYSQL_ROOT_PASSWORD: <set to the key 'database-password' 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-7d76ff58f8-6j2gx \
| 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 `echo` command provides useful output:

```
[student@workstation D0288-apps]$ oc rsh quotesapi-7d76ff58f8-6j2gx 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-7d76ff58f8-6j2gx
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-6b7ffcc649-dslpq:/tmp/quote.sql
```

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

```
[student@workstation D0288-apps]$ oc rsh -t quotesdb-6b7ffcc649-dslpq
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

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.

Instructions

1. Navigate to your local clone of the D0288-apps Git repository and create a new branch named `source-build` from the main 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 `python3 -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

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.

Instructions

1. Navigate to your local clone of the D0288-apps Git repository and create a new branch named `source-build` from the main 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 `main` 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 main
...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` 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 --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
  deployment.apps "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 `python3 -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 `python3 -m json.tool` command to validate the JSON file:

```
[student@workstation D0288-apps]$ python3 -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
  deployment/greet deploys istag/greet:latest <-
    bc/greet source builds https://github.com/yourgithubuser/D0288-apps#source-
build on openshift/nodejs:10
  deployment #2 running for 23 seconds - 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-2-build  0/1     Completed  0          2m
greet-594d667bc4-2b9ch 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:

- RHOPC 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 Containerfile Instructions (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/>]

Using the Red Hat Container Catalog

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 4.6 at

https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6/html-single/images/index#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 Containerfile 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 Containerfile 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:

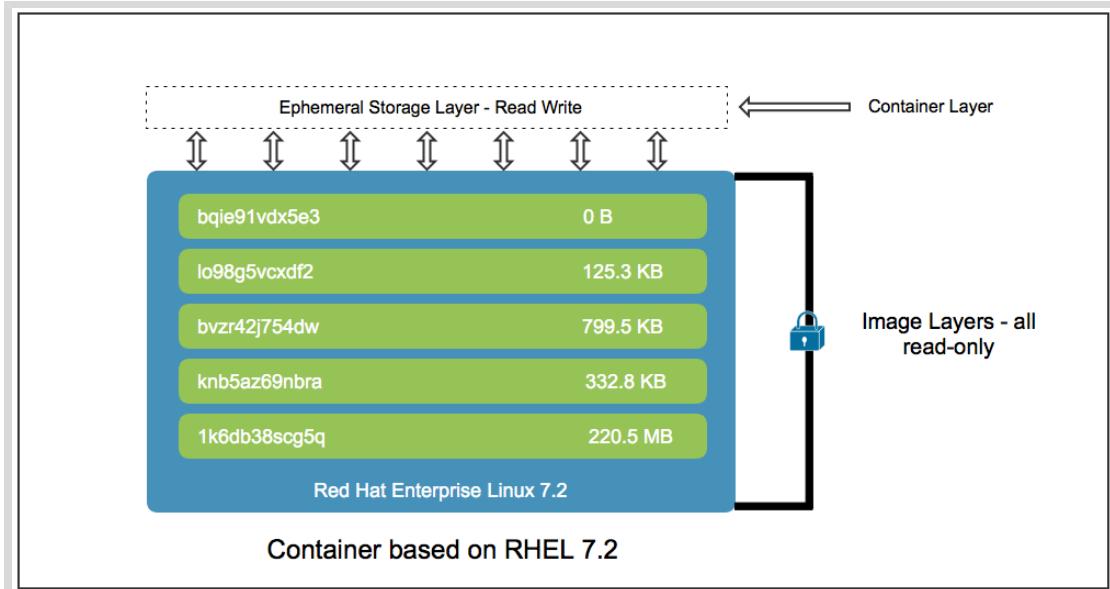
- **Minimal size:** The Universal Base Image is a relatively minimal (approximately 90-200 MB) base container image with fast startup times.
- **Security:** Provenance is a huge concern when using container base images. You must use a trusted image, from a trusted source. Language runtimes, web servers, 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 Containerfile Instructions

A Containerfile automates the building of container images. A Containerfile 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 Containerfile 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 Containerfile. The container build process uses `/bin/sh` to execute commands.

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

Each instruction in a Containerfile results in a new layer for the final container image. Therefore, having too many instructions in a Containerfile creates too many layers, resulting in images that are unnecessarily large. For example, consider the following RUN instruction in a Containerfile:

```
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 Containerfiles, 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 Podman, 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
<code>io.openshift.tags</code>	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.
<code>io.k8s.description</code>	This label provides consumers of the container image more detailed information about the service or functionality that the image provides.
<code>io.openshift.expose-services</code>	This label contains a list of service ports that match the EXPOSE instructions in the Containerfile and provides more descriptive information about what actual service is provided to consumers. The format is <code>PORT [/PROTO] :NAME</code> where <code>[/PROTO]</code> is optional and it defaults to <code>tcp</code> 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 Containerfile.

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 Containerfile. 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 Containerfile instructions. It is useful to store information such as software version numbers, and also to append directories to the `PATH` environment variable.

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 Containerfile. 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 `Containerfile` 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 `Containerfile` 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 `Containerfile` 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 `Containerfile` 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 `Containerfile`. The parent `Containerfile` is shown below:

```

FROM registry.access.redhat.com/rhscl/nodejs-6-rhel7
EXPOSE 3000
# Make all Node.js apps use /opt/app-root 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 Containerfile 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 Containerfile.

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 Containerfiles for OpenShift

When you write or change a Containerfile 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 Containerfile 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 Containerfiles 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-single/images/creating-images

► Guided Exercise

Building Container Images with Advanced Containerfile Instructions

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

Outcomes

You should be able to:

- Create an Apache HTTP Server container image using a Containerfile and deploy it to an OpenShift cluster.
- Create a child container image by extending the parent Apache HTTP Server image.
- Change the Containerfile 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).
- The Containerfile 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
```

Instructions

► 1. Review the Apache HTTP Server parent Containerfile.

A prebuilt Apache HTTP Server parent container image is provided in the Quay.io public registry at quay.io/redhattraining/httpd-parent. Briefly review the Containerfile for this parent image located at ~/D0288/labs/container-build/httpd-parent/Containerfile:

```
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 && \
    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 Containerfile.

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 Containerfile for the child container image is stored in the classroom Git repository server. To view the Containerfile, 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 DO288-apps Git repository and checkout the main 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 main
...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/Containerfile file. The Containerfile 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 Containerfile.

- 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 the Apache HTTP Server child image:

```
[student@workstation D0288-apps]$ podman build --layers=false \
-t do288-apache ./container-build
STEP 1: FROM quay.io/redhattraining/httpd-parent ①
...output omitted...
STEP 2: COPY src/ ${DOCROOT}/ ②
STEP 3: COMMIT do288-apache
...output omitted...
Storing signatures
...output omitted...
```

- ① The Containerfile is automatically identified and podman pull the parent image.
- ② The ONBUILD instruction in the parent Containerfile triggers the copying of the child's `index.html` file, which overwrites the parent index file.



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
```

3.4. View the the images:

```
[student@workstation D0288-apps]$ podman images
localhost/do288-apache          latest fc114a884288  9 minutes ago  236 MB
quay.io/redhattraining/httpd-parent latest 4346d3cace25  2 years ago   236 MB
```

3.5. Tag and push the image to Quay.io

```
[student@workstation D0288-apps]$ podman tag do288-apache \
quay.io/${RHT_OCP4_QUAY_USER}/do288-apache
[student@workstation D0288-apps]$ podman login quay.io -u ${RHT_OCP4_QUAY_USER}
Password:
Login Succeeded!
[student@workstation D0288-apps]$ podman push \
quay.io/${RHT_OCP4_QUAY_USER}/do288-apache
...output omitted...
Storing signatures
[student@workstation D0288-apps]$
```

3.6. Log into Quay.io [<https://quay.io>] and make the new image public

3.7. Deploy the Apache HTTP Server child image:

```
[student@workstation D0288-apps]$ oc new-app --name hola \
quay.io/${RHT_OCP4_QUAY_USER}/do288-apache
--> Found Container image 1d6f8d3 (11 minutes old) from quay.io for "quay.io/
youruser/do288-apache"
```

```

Red{nbsp}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{nbsp}Hat only supports Red{nbsp}Hat
technologies through subscriptions for Red{nbsp}Hat products. This image is
maintained by Red{nbsp}Hat and updated regularly.

Tags: base rhel8

...output omitted...
--> Success
Application is not exposed. You can expose services to the outside world by
executing one or more of the commands below:
'oc expose svc/hola'
Run 'oc status' to view your app.

```

- ▶ 4. 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-13p75f5  0/1     CrashLoopBackOff    0          12s
```

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

```
[student@workstation D0288-apps]$ oc logs hola-13p75f5
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 Because OpenShift runs containers using a random userid, ports below 1024 are privileged and can only be run as `root`.
- ② 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.

- 6. Delete all resources from the OpenShift project. The next step changes the Containerfile to follow Red Hat recommendations for OpenShift.

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

```
[student@workstation D0288-apps]$ oc delete all -l app=hola
service "hola" deleted
deployment.apps "hola" deleted
build.build.openshift.io "hola-1" deleted
```

- 7. Change the Containerfile for the child container to run on an OpenShift cluster by updating the Apache HTTP Server process to run as a random, unprivileged user.

- 7.1. Edit the ~/D0288-apps/container-build/Containerfile file and perform the following steps. You can also copy these instructions from the provided ~/D0288/solutions/container-build/Containerfile file.
- 7.2. Override the EXPOSE instruction from the parent image and change the port to 8080.

```
EXPOSE 8080
```

- 7.3. 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"
```

- 7.4. 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
RUN sed -i "s/#ServerName www.example.com:80/ServerName 0.0.0.0:8080/g" /etc/
httpd/conf/httpd.conf
```

- 7.5. 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
```

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

USER 1001

- 7.7. Save the Containerfile 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 the Containerfile to enable running as a random uid on OpenShift"
...output omitted...
[student@workstation container-build]$ git push
...output omitted...
[student@workstation container-build]$ cd ..
```

- 8. Rebuild and redeploy the Apache HTTP Server child container image.

- 8.1. Remove old images

```
[student@workstation ~]$ podman rmi -a --force
...output omitted...
```

- 8.2. Re-create the application using the new Containerfile:

```
[student@workstation ~]$ podman build --layers=false \
-t do288-apache ./container-build
STEP 1: FROM quay.io/redhattraining/httpd-parent
...output omitted...
STEP 2: COPY src/ ${DOCROOT}/
STEP 3: EXPOSE 8080
STEP 4: 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"
STEP 5: RUN sed -i "s/Listen 80/Listen 8080/g" /etc/httpd/conf/httpd.conf
STEP 6: RUN chgrp -R 0 /var/log/httpd /var/run/httpd &&      chmod -R g=u /var/log/
httpd /var/run/httpd
STEP 7: USER 1001
STEP 8: COMMIT do288-apache
...output omitted...
```

- 8.3. Tag the new image and replace the image in Quay.io

```
[student@workstation D0288-apps]$ podman tag do288-apache \
quay.io/${RHT_OCP4_QUAY_USER}/do288-apache
[student@workstation D0288-apps]$ podman push \
quay.io/${RHT_OCP4_QUAY_USER}/do288-apache
...output omitted...
Storing signatures
[student@workstation D0288-apps]$
```

- 8.4. Redeploy the Apache HTTP Server child container image.

```
[student@workstation ~]$ oc new-app --name hola \
quay.io/${RHT_OCP4_QUAY_USER}/do288-apache
--> Found container image fe746d5 (7 minutes old) from quay.io for "quay.io/
youruser/do288-apache"
...output omitted...
--> Success
Application is not exposed. You can expose services to the outside world by
executing one or more of the commands below:
'oc expose service/hola'
Run 'oc status' to view your app.
```

8.5. Wait until the pod is ready and running. View the status of the application pod:

```
[student@workstation ~]$ oc get pods
NAME      READY   STATUS    RESTARTS   AGE
hola-1775gkw   1/1     Running   0          5s
```

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

- 9. Create an OpenShift route to expose the application to external access:

```
[student@workstation ~]$ oc expose --port='8080' svc/hola
route.route.openshift.io/hola exposed
```

- 10. 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
```

- 11. 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...
```

- 12. 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	your-project	kubernetes.io/dockercfg	1	Aug 3, 12:05 am	...
builder-token-cs2r4	your-project	kubernetes.io/service-account-token	4	Aug 3, 12:05 am	...
builder-token-hxj2h	your-project	kubernetes.io/service-account-token	4	Aug 3, 12:05 am	...
default-dockercfg-97qrn	your-project	kubernetes.io/dockercfg	1	Aug 3, 12:05 am	...
default-token-c892t	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-l-ca	NS your-project	1	Aug 3, 3:18 pm
CM your-project-l-global-ca	NS your-project	1	Aug 3, 3:18 pm
CM your-project-l-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: value1 ①②
  key2: value2 ③④
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: cm9vdAo= 1 2
  password: c2VjcmV0Cg== 3 4
kind: Secret 5
metadata:
  name: mysecret 6
  type: Opaque
```

- 1** 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.
- 2** The value stored for the first key, in base64-encoded format.
- 3** The name of the second key.
- 4** The value stored for the second key, in base64-encoded format.
- 5** The OpenShift resource type; in this case, a secret.
- 6** 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  
bmV3cGFzc3dvcnQK
```

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":"bmV3cGFzc3dvcnQK"}}'
```

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, use the `oc set env` command:

```
[user@host ~]$ oc set env deployment/mydcname \
--from configmap/myconf
```

To mount all keys from a configuration map as files from a volume inside pods created from a deployment, use the `oc set volume` command:

```
[user@host ~]$ oc set volume deployment/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, use the `oc set env` command:

```
[user@host ~]$ oc set env deployment/mydcname \
--from secret/mysecret
```

To mount data from a secret resource as a volume inside pods created from a deployment, use the `oc set volume` command:

```
[user@host ~]$ oc set volume deployment/mydcname --add \
-t secret -m /path/to/mount/volume \
--name myvol --secret-name mysecret
```

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 deployment/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 deployment/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 4.6 at https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6/html-single/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
```

Instructions

► 1. Review the application source code.

- 1.1. Enter your local clone of the `D0288-apps` Git repository and check out the `main` 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 main
...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 --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
deployment.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-build  0/1     Completed  0          65s
myapp-597fdb8cc9-z6t86  1/1     Running   0          29s
```

- 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": "dXNlcm5hbWU9dXNlcjEKcGFzc3dvcmQ9cGFyczEKc2..."
  },
  "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 deployment/myapp \
--from configmap/myappconf
deployment.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 deployment/myapp --add \
-t secret -m /opt/app-root/secure \
--name myappsec-vol --secret-name myappfilesec
deployment.apps/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 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)
deployment/myapp deploys istag/myapp:latest <-
bc/myapp source builds https://github.com/youruser/D0288-apps#app-config on
openshift/nodejs:12
deployment #4 running for 10 seconds - 1 pod
deployment #3 deployed 44 seconds ago
deployment #2 deployed 3 minutes ago
deployment #1 deployed 4 minutes ago
```

**Note**

You should see the application redeployed twice, due to the two `oc set env` commands that change the deployment.

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          8m12s
myapp-ddffbc7f9-ntsjq  1/1     Running   0          3m53s
```

- 6.3. Use the `oc rsh` command to inspect the environment variables in the container:

```
[student@workstation D0288-apps]$ oc rsh myapp-ddffbc7f9-ntsjq 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 delete pod` command to trigger a new deployment. After the pod is deleted, the replication controller automatically deploys a new pod. This ensures that the application picks up the changed values in the configuration map:

```
[student@workstation D0288-apps]$ oc delete pod myapp-ddffbc7f9-ntsjq
pod "myapp-ddffbc7f9-ntsjq" deleted
```

- 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
myapp-1-build 0/1     Completed  0          16m
myapp-ddffbc7f9-5wls6 1/1     Running   0          2m37s
```

- 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 DO288-apps]$ lab app-config finish
```

This concludes the guided exercise.

► Lab

Designing Containerized Applications for OpenShift

In this lab, you will fix the Containerfile 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 Containerfile 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>

Instructions

1. Navigate to your local clone of the D0288-apps Git repository and create a new branch named `design-container` from the `main` branch. Briefly review the `Containerfile` 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. 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 `Containerfile` 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 `Containerfile` 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

In this lab, you will fix the Containerfile 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 Containerfile 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`

Instructions

1. Navigate to your local clone of the D0288-apps Git repository and create a new branch named `design-container` from the `main` branch. Briefly review the `Containerfile` for the application in the `/home/student/D0288-apps/hello-java/` directory.
 - 1.1. Check out the `main` branch of the Git repository.

```
[student@workstation ~]$ cd D0288-apps  
[student@workstation D0288-apps]$ git checkout main  
...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/Containerfile` 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 image from the Containerfile.

```
[student@workstation D0288-apps]$ podman build --layers=false \
-t do288-hello-java ./hello-java
STEP 1: FROM registry.access.redhat.com/ubi8/ubi:8.0
...output omitted...
STEP 11: COMMIT do288-hello-java
...output omitted...
```

- 2.5. Tag the new image and push it to Quay.io

```
[student@workstation D0288-apps]$ podman tag do288-hello-java \
quay.io/${RHT_OCP4_QUAY_USER}/do288-hello-java
[student@workstation D0288-apps]$ podman login quay.io -u ${RHT_OCP4_QUAY_USER}
Password:
Login Succeeded!
[student@workstation D0288-apps]$ podman push \
quay.io/${RHT_OCP4_QUAY_USER}/do288-hello-java
...output omitted...
Storing signatures
```

- 2.6. Log into Quay.io [http://quay.io], and make the newly added image public. The new-app command will fail without this step.

- 2.7. Use the image in the repository to create a new application in OpenShift named elvis

```
[student@workstation D0288-apps]$ oc new-app --name elvis \
quay.io/${RHT_OCP4_QUAY_USER}/do288-hello-java
...output omitted...
--> Creating resources ...
imagestream.image.openshift.io "elvis" created
deployment.apps "elvis" created
service "elvis" created
--> Success
...output omitted...
```

3. 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. 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
elvis-799f8df69b-vh6fr   0/1     Error      2          34s
```

- 3.2. View the logs for the application pod and investigate why the application pod failed to start.

```
[student@workstation D0288-apps]$ oc logs elvis-799f8df69b-vh6fr
/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 Containerfile.

4. Edit the Containerfile 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 Containerfile at `/home/student/D0288-apps/hello-java/Containerfile` 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/Containerfile`.

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 Containerfile 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 Containerfile to run with random user id on OpenShift"
[student@workstation hello-java]$ git push
[student@workstation hello-java]$ cd ..
[student@workstation D0288-apps]$
```

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 D0288-apps]$ podman rmi -a --force
...output omitted...
[student@workstation D0288-apps]$ podman build --layers=false \
-t do288-hello-java ./hello-java
STEP 1: FROM registry.access.redhat.com/ubi8/ubi:8.0
...output omitted...
STEP 7: RUN chgrp -R 0 /opt/app-root && chmod -R g=u /opt/app-root
...output omitted...
STEP 9: USER 1001
...output omitted...
```

6.2. Tag the new image and push it to Quay.io

```
[student@workstation D0288-apps]$ podman tag do288-hello-java \
quay.io/${RHT_OCP4_QUAY_USER}/do288-hello-java
[student@workstation D0288-apps]$ podman push \
quay.io/${RHT_OCP4_QUAY_USER}/do288-hello-java
...output omitted...
Storing signatures
```

6.3. Remove the old project and re-create it.

```
[student@workstation D0288-apps]$ oc delete project \
${RHT_OCP4_DEV_USER}-design-container
...output omitted...
[student@workstation D0288-apps]$ oc new-project \
${RHT_OCP4_DEV_USER}-design-container
```

6.4. Use the updated image in the repository to create a new application in the OpenShift cluster named elvis

```
[student@workstation D0288-apps]$ oc new-app --name elvis \
quay.io/${RHT_OCP4_QUAY_USER}/do288-hello-java
...output omitted...
--> Creating resources ...
imagestream.image.openshift.io "elvis" created
deployment.apps "elvis" created
service "elvis" created
--> Success
...output omitted...
```

6.5. Wait for the application pod to be ready and running.

NAME	READY	STATUS	RESTARTS	AGE
elvis-6d4fc74867-b6z2h	1/1	Running	0	29s

- 6.6. View the logs for the application pod and verify that there are no errors during startup:

```
[student@workstation ~]$ oc logs elvis-6d4fc74867-b6z2h
Starting hello-java app...
JVM options => -Xmx512m
...output omitted...
2021-06-22 17:43:56,211 INFO [org.wildfly.extension.undertow] (MSC service thread
1-2) WFLYUT0018: Host default-host starting
...output omitted...
2021-06-22 17:43:56,484 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-6d4fc74867-b6z2h
```

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
Events: <none>
```

- 8.3. Use the `oc set env` command to add the configuration map to the deployment configuration:

```
[student@workstation ~]$ oc set env deployment/elvis --from cm/appconfig
deployment.apps/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...
deployment/elvis deploys istag/elvis:latest
  deployment #3 running for 21 seconds - 1 pod
  deployment #2 deployed 16 minutes ago
  deployment #1 deployed 16 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 D0288-apps]$ oc get pods
NAME           READY   STATUS    RESTARTS   AGE
elvis-66c7f6d47f-ll2jq   1/1     Running   0          2m36s
```

- 9.3. Verify that the APP_MSG key is injected into the container as an environment variable:

```
[student@workstation ~]$ oc rsh elvis-66c7f6d47f-ll2jq 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-66c7f6d47f-ll2jq]. 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 Containerfile, 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.

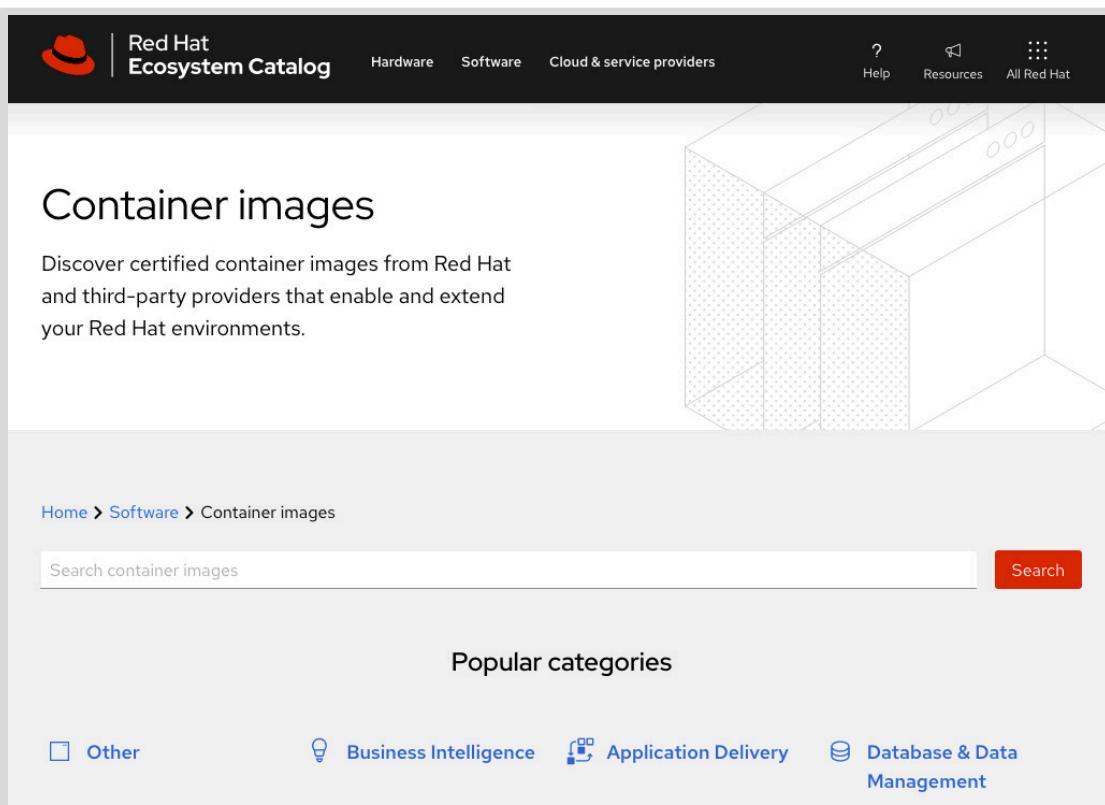


Figure 3.1: Red Hat Container Catalog

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.

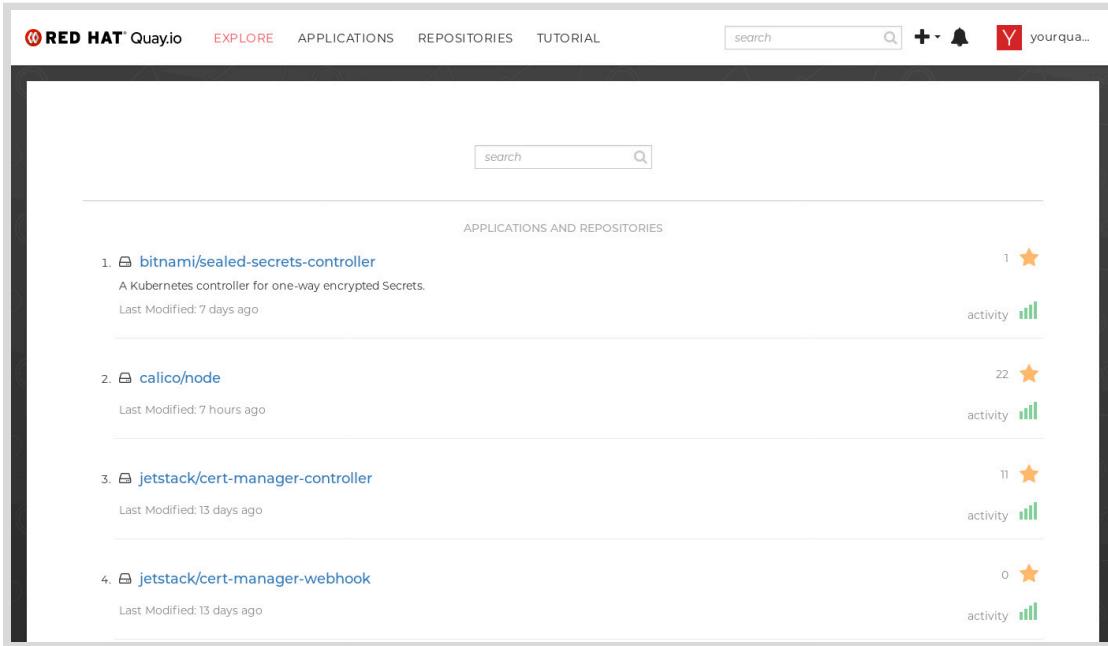


Figure 3.2: Quay.io web page

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 URLs 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
```

Instructions

- 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. Do not make this repository public. This repository must be kept private for this lab.

```
[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.

```
[student@workstation ~]$ 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 ~]$ 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 ~]$ podman logs sleep
...output omitted...
sleeping
sleeping
```

- 2.4. Stop and remove the test container:

```
[student@workstation ~]$ podman stop sleep
...output omitted...
[student@workstation ~]$ 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 --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
```

- 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 --name sleep \
--docker-image quay.io/${RHT_OCP4_QUAY_USER}/ubi-sleep:1.0
...output omitted...
--> Creating resources ...
imagestream.image.openshift.io "sleep" created
deployment.apps "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-7bf77b7596-ldrsv   1/1     Running   0          95s
```

- 3.7. Verify that the pod produces log output:

```
[student@workstation ~]$ oc logs sleep-7bf77b7596-ldrsv
...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:

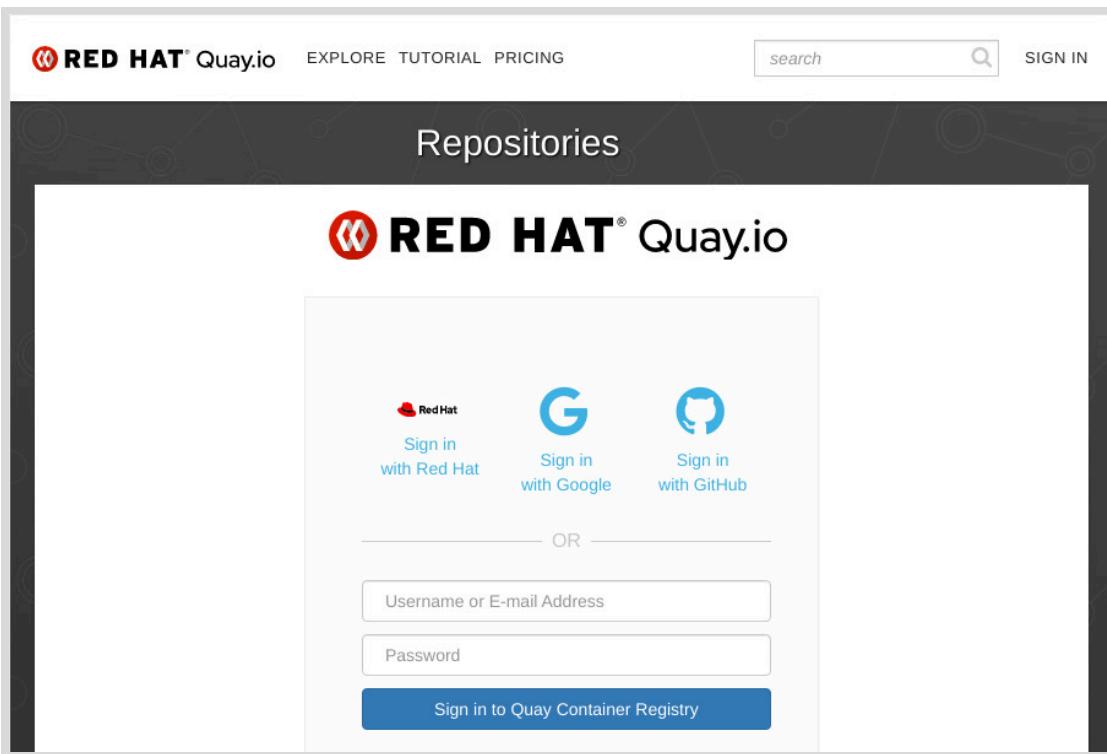
```
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-external-registry
project.project.openshift.io "external-registry" deleted
```

- 4.2. Delete the container image from the external registry:

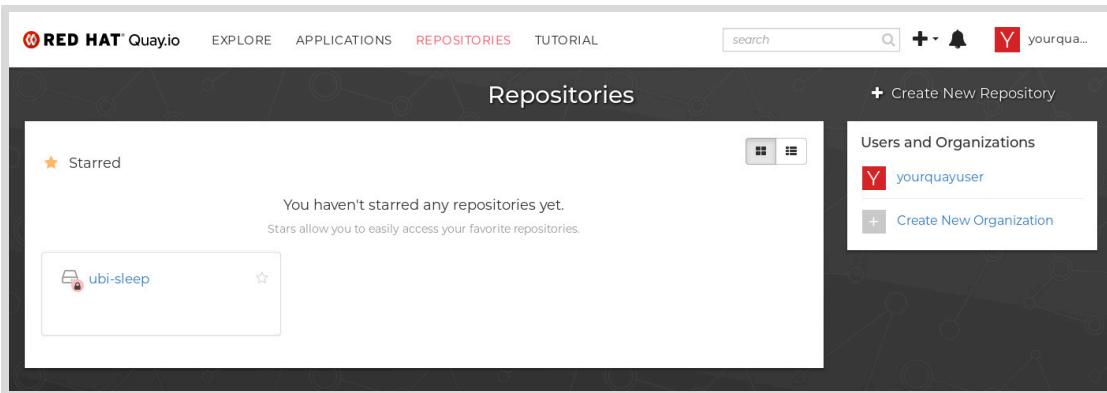
```
[student@workstation ~]$ skopeo delete \
docker://quay.io/${RHT_OCP4_QUAY_USER}/ubi-sleep:1.0
```

- 4.3. Log in to Quay.io using your personal free account.

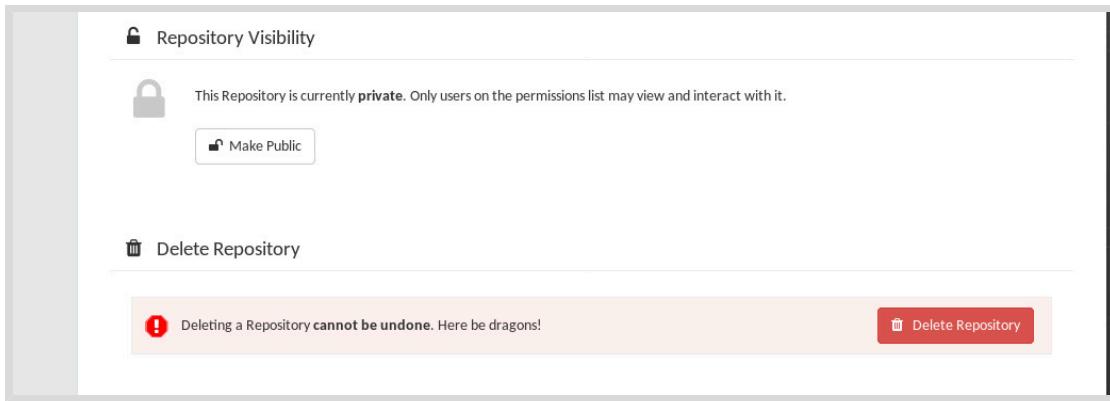
Navigate to <http://quay.io> and click **Sign In** to provide your user credentials. Click **Sign in to Quay Container Registry** to log in to Quay.io.



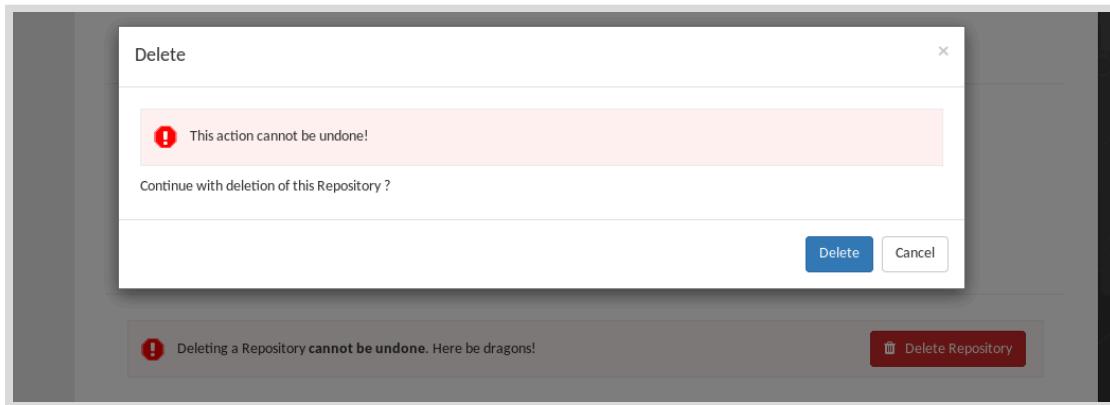
- 4.4. On the Quay.io main menu, click **Repositories** and look for **ubi-sleep**. The lock icon indicates that it is a private repository that requires authentication for both pulls and pushes. Click **ubi-sleep** to display the **Repository Activity** page.



- 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 `Containerfile` 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.imageregistry 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 4.6 at
https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6/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 4.6 at
https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6/html-single/registry/index#accessing-the-registry

► Guided Exercise

Allowing Access to 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
```

Instructions

► 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 ~]$ 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 ~]$ 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 ~]$ 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 ~]$ podman rm info
...output omitted...
[student@workstation ~]$ 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. While it is possible to also use Image Streams with Kubernetes Deployments, this is not typically needed and is outside the scope of this course.

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.

Chapter 3 | Publishing Enterprise Container Images

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.

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 ...    6 days ago
php:7.3      image-registry ...    6 days ago
php:latest   image-registry ...    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 -i myis` command. You can also use that image stream as a builder image in the `oc new-app 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 \ ①
--from registry.example.com/myorg/myimage
[user@host ~]$ oc policy add-role-to-group system:image-puller \
system:serviceaccounts:myapp ②
[user@host ~]$ oc project myapp
[user@host ~]$ oc new-app -i shared/myis
```

- ① 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.
- ② The `system:image-puller` role allows a service account to pull the image layers that the image stream cached in the internal registry.
- ③ 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 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 4.6 at https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6/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
```

Instructions

- 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...acc1②
  2 minutes ago

...output omitted...
```

- ① The image stream tag `hello-world:latest` references an image from Quay.io.
- ② 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 --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
deployment.apps "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-6599bb7b9c-zk58m 1/1     Running   0          40s
```

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

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.

Instructions

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

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.

Instructions

1. Push the `php-info` OCI-formatted container image to Quay.io.

Chapter 3 | Publishing Enterprise Container Images

- 1.1. 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
```

- 1.2. 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
```

- 1.3. Use the `podman` command to log in to Quay.io

```
[student@workstation ~]$ podman login -u ${RHT_OCP4_QUAY_USER} quay.io  
Password:  
Login Succeeded!
```

- 1.4. 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
```

- 1.5. 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...
```

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.

- 2.1. 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 --name info \
-i ${RHT_OCP4_DEV_USER}-common/php-info
...output omitted...
--> Creating resources ...
imagestreamtag.image.openshift.io "info:latest" created
deployment.apps "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-5c687bc4bc-j4sdz   1/1     Running   0          26s
```

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:

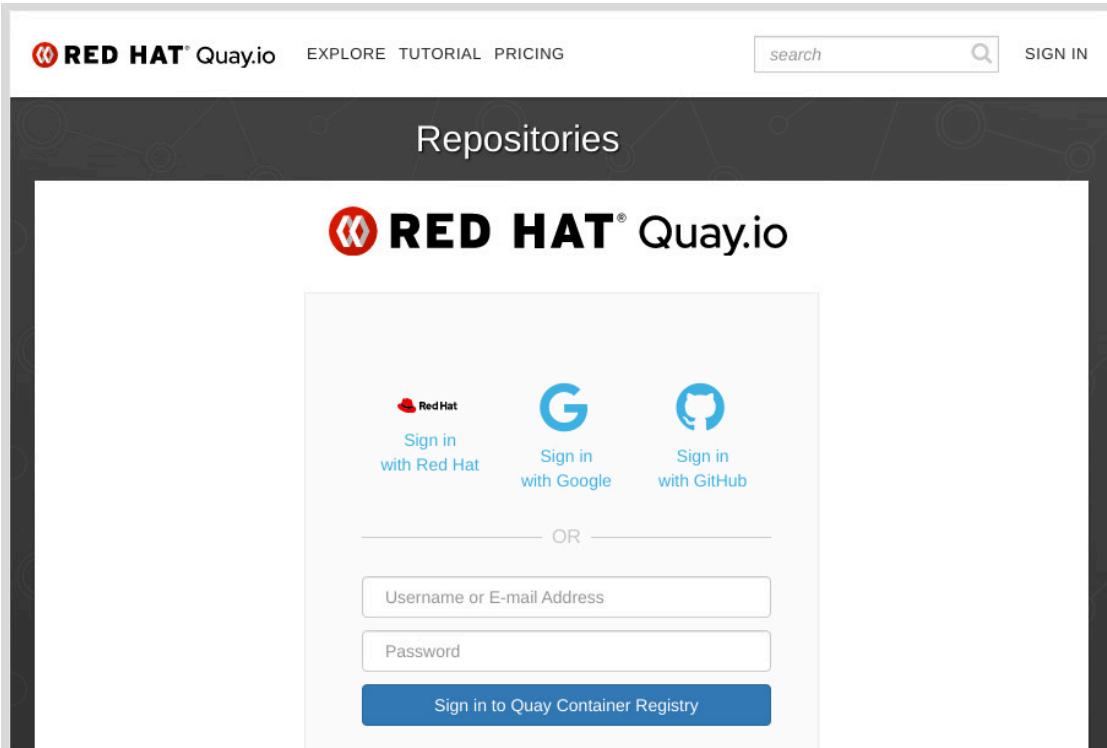
```
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-common
project.project.openshift.io "youruser-common" deleted
```

6.3. Delete the container image from the external registry:

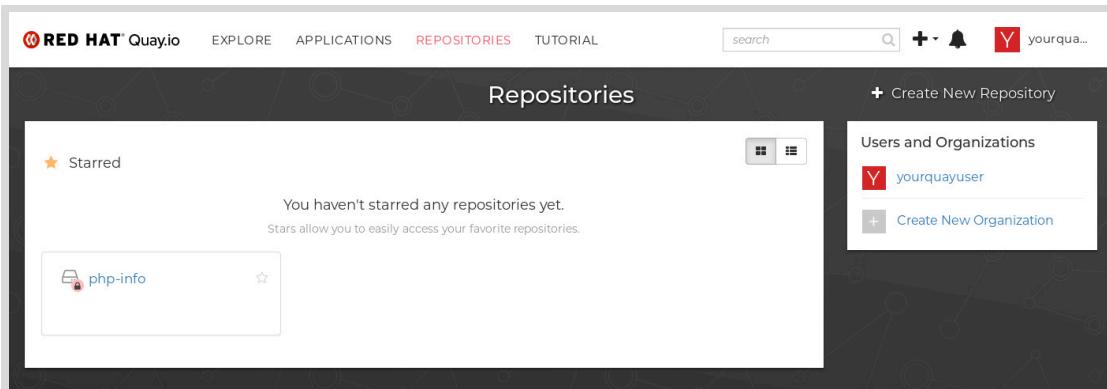
```
[student@workstation ~]$ skopeo delete \
docker://quay.io/${RHT_OCP4_QUAY_USER}/php-info:latest
```

6.4. Log in to Quay.io using your personal free account.

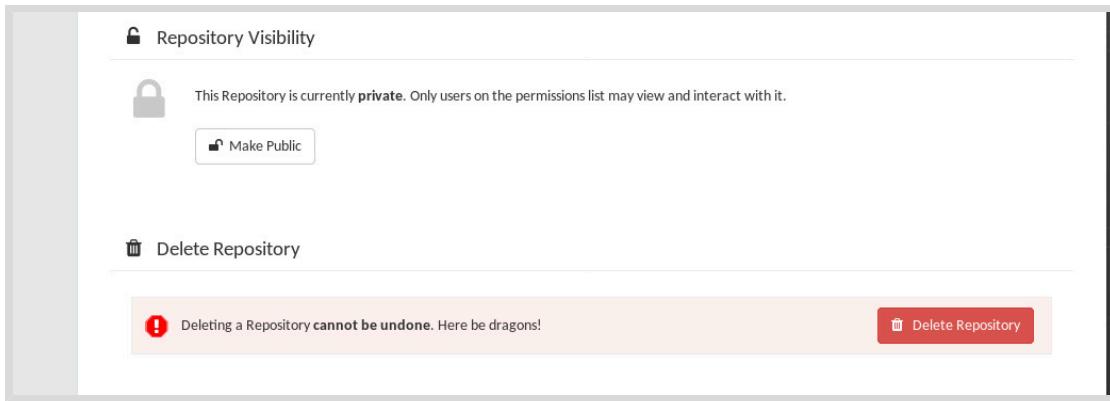
Navigate to <http://quay.io> and click **Sign In** to provide your user credentials. Click **Sign in to Quay Container Registry** to log in on Quay.io.



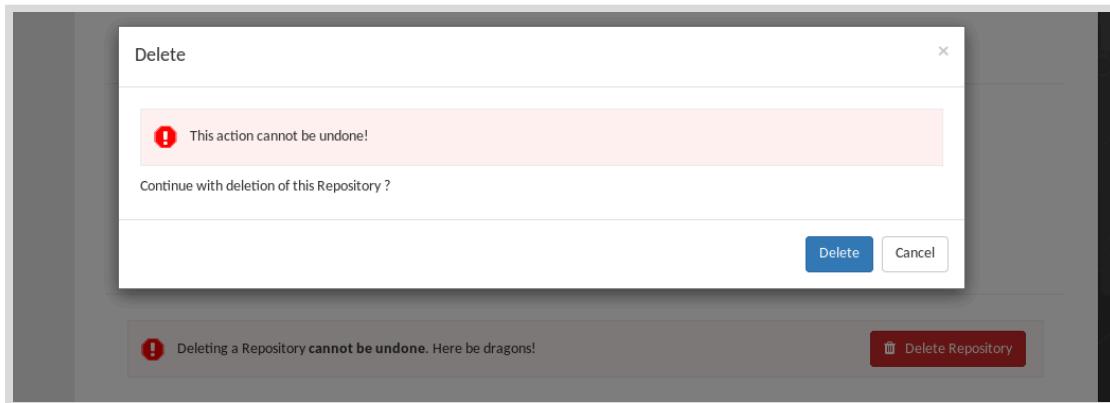
- 6.5. On the Quay.io main menu, click **Repositories** and look for **php-info**. Click **php-info** to open the **Repository Activity** page.



- 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

Managing Builds on OpenShift

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

Managing Builds on OpenShift

Describing the Red Hat OpenShift Build Process

Objectives

After completing this section, you should be able to describe the Red Hat 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, Red Hat 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 Red Hat OpenShift:

- Source-to-image (S2I) build
- Docker build
- Custom build

Each strategy requires a container image.

Source-to-image (S2I) build

The `source-to-image` strategy creates a new container image based on application source code or application binaries. Red Hat 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 `Containerfiles`.

Red Hat OpenShift bases the `source-to-image` strategy upon the source-to-image (S2I) process.

Docker build

The `docker` `build` strategy uses the `buildah` command to build a new container image given a `Containerfile` file. The `docker` strategy can retrieve the `Containerfile` and the artifacts to build the container image from a Git repository, or can use a `Containerfile` provided inline in the build configuration as a build source.

The Docker build runs as a pod inside the Red Hat OpenShift cluster. Developers do not need to have Docker tooling on their workstation.

**Note**

Docker builds require elevated privileges and the Red Hat OpenShift cluster administrator might deny some or all users the right to start Docker builds.

Custom build

The `custom` build 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. Red Hat OpenShift supports the following six types of input sources, listed in order of precedence:

- `Containerfile`: Specifies the Containerfile inline to build an image.
- `Image`: You can provide additional files to the build process when you build from images.
- `Git`: Red Hat 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 `Containerfile` takes precedence, it overrides any other `Containerfile` provided by another input. Additionally, binary input and Git repositories are mutually exclusive inputs.

**Note**

Although Red Hat 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 Red Hat OpenShift must create a new build.

Red Hat 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
    ...output omitted...
  },
  "spec": {
    "triggers": [ ②
      {
        "type": "GitHub",
        "github": {
          "secret": "gukAWHzq1On4AJlMjvjS" ③
        }
      },
      ...output omitted...
    ],
    "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"
      }
    },
    ...output omitted...
  },
  ...output omitted...
}
```

- 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 Red Hat 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.

- ➅ The `source` attribute is responsible for defining the input source of the build. In this example, it uses a Git repository.
- ➆ Defines the build strategy to build the final container image. In this example, it uses the `Source` strategy.
- ➇ 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 4.6 at
https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6/html-single/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. Source-to-image
- 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. Containerfile

► 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. Containerfile
 - 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 Containerfile 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. Source-to-image
 - 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. Containerfile

► 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. Containerfile
- 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 Containerfile 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 Red Hat OpenShift using the `oc create -f` command.

Managing Application Builds Using CLI

Red Hat 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 Red Hat 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. Recall that bc is shorthand notation for build configuration.

```
[user@host ~]$ oc delete bc/name
```

The following 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, the five most recent builds that have completed are persisted. 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
  ...contents omitted...

```

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

Red Hat OpenShift administrators can manually prune builds using the `oc adm object pruning` command.

Log Verbosity

Red Hat 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.

To override the administrator's default setting, a developer may 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. Red Hat 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 4.6 at

https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6/html-single/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:1.8`).
- The sample application in the Git repository, in the `java-serverhost` directory.

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
```

Instructions

► 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 `main` 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 main
...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 `java-serverhost` folder.

Examine the /home/student/D0288-apps/java-serverhost/src/main/java/com/redhat/training/example/javaserverhost/rest/ServerHostEndPoint.java file:

```
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. Source your classroom environment configuration:

```
[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 following parameters for the build:

- Application name: jhost
- Branch: manage-builds
- Build environment variable: MAVEN_MIRROR_URL=http://\${RHT_OCP4_NEXUS_SERVER}/repository/java
- Image stream: redhat-openjdk18-openshift:1.8
- Build directory: java-serverhost

You can execute the /home/student/D0288/labs/manage-builds/oc-new-app.sh script, or execute the command manually:

```
[student@workstation D0288-apps]$ oc new-app --name jhost \
--build-env MAVEN_MIRROR_URL=http://${RHT_OCP4_NEXUS_SERVER}/repository/java \
-i redhat-openjdk18-openshift:1.8 \
https://github.com/${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
deployment.apps "jhost" created
service "jhost" created
...output omitted...
```



Note

Do not put a space after the environment variable MAVEN_MIRROR_URL=. The environment variable follows the *NAME=VALUE* format.

3.2. Use the `oc logs` command to check the build logs from the jhost build:

```
[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:

NAME	READY	STATUS	RESTARTS	AGE
jhost-1-build	0/1	Completed	0	32m
jhost-7d9b748448-qwtqs	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 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

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:

```
[student@workstation D0288-apps]$ oc get bc  
NAME      TYPE      FROM          LATEST  
jhost     Source    Git@manage-builds  1
```

The `oc new-app` command created the `jhost` build configuration resource with the `Source` strategy and 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 `/home/student/D0288-apps/java-serverhost/src/main/java/com/redhat/training/example/jaserverhost/rest/ServerHostEndPoint.java` file and update to version 2.0:

```
... code omitted ...  
String msg = "I am running on server "+host+" Version `2.0 \n";`  
return Response.ok(msg).build();  
... code omitted ...
```

- 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(-)
[student@workstation java-serverhost]$ git push
[student@workstation java-serverhost]$ cd ..
```

5.3. Start a new build with the `oc start-build` command:

```
[student@workstation D0288-apps]$ oc start-build bc/jhost
build.build.openshift.io/jhost-2 started
```

5.4. The application does not contain the changes that you committed to your local Git repository. The application uses the remote Git repository URL as the source. You must push the changes to the remote Git repository before you start the OpenShift build.

Cancel the build to avoid a new deployment using the older version:

```
[student@workstation D0288-apps]$ 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 D0288-apps]$ oc get builds
NAME      TYPE      FROM      STATUS      ...
jhost-1   Source    Git@cb73a3d Complete   ...
jhost-2   Source    Git@cb73a3d Cancelled (CancelledBuild) ...
```

If you did not cancel the build before it completed, you will see the following output:

```
[student@workstation D0288-apps]$ oc get builds
NAME      TYPE      FROM      STATUS      ...
jhost-1   Source    Git@cb73a3d Complete   ...
jhost-2   Source    Git@20d7733 Complete   ...
```

Continue with the following steps.

5.6. Push the updated source code to the Git server:

```
[student@workstation D0288-apps]$ 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-5978d79042-kqrwd 1/1     Running   0          4m40s
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
```



Note

If the previous command returns an HTML page with the text `Application is not available`, wait a couple of seconds and try again.

► 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`, execute 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 Red Hat 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. Red Hat 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

Red Hat OpenShift webhook triggers are HTTP API endpoints that start a new build. Use a webhook to integrate Red Hat 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 Red Hat 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 Containerfile.

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

Red Hat OpenShift webhook triggers are HTTP API endpoints that start new builds. Use a webhook to integrate Red Hat 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 Red Hat OpenShift with the latest code.

Software does not have to be a VCS to use these API endpoints, but Red Hat 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 Red Hat OpenShift. A generic webhook can be used by any software to start an Red Hat 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 Red Hat 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 4.6 at https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6/html-single/builds/triggering-builds-build-hooks

► Guided Exercise

Triggering Builds

In this exercise, you will trigger a new build of an application in Red Hat 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 Red Hat OpenShift cluster.
- The original version of the PHP S2I builder image (`php-73-ubi8-original.tar.gz`)
- The new version of the PHP S2I builder image (`php-73-ubi8-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
```

Instructions

► 1. Prepare the lab environment.

- 1.1. Source the classroom environment configuration:

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

- 1.2. Log in to Red Hat OpenShift by 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 7.3 builder image to your Quay.io registry.

You can copy or execute the command by using the /home/student/D0288/labs/trigger-builds/push-original.sh script:

```
[student@workstation ~]$ cd /home/student/D0288/labs/trigger-builds
[student@workstation trigger-builds]$ skopeo copy \
oci-archive:php-73-ubi8-original.tar.gz \
docker://quay.io/${RHT_OCP4_QUAY_USER}/php-73-ubi8: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 must add a secret to 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 Red Hat OpenShift about image changes.

```
[student@workstation trigger-builds]$ oc import-image php \
--from quay.io/${RHT_OCP4_QUAY_USER}/php-73-ubi8 --confirm
imagestream.image.openshift.io/php-73-ubi8 imported

Name:          php
...output omitted...
latest
tagged from quay.io/youruser/php-73-ubi8
```

```
* quay.io/youruser/php-73-ubi8@sha256:c919...8cb2
  Less than a second ago
  ...output omitted...
```

► 3. Create a new application.

3.1. Create a new application build with the following parameters:

- Name: trigger
- Builder image: php
- Directory of the application: trigger-builds

You can copy or execute the complete command by using the /home/student/D0288/labs/trigger-builds/oc-new-app.sh script:

```
[student@workstation trigger-builds]$ oc new-app \
--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 buildconfig/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-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

You can copy or execute the complete command by using the /home/student/DO288/labs/trigger-builds/push-newer.sh script:

```
[student@workstation trigger-builds]$ skopeo copy \
oci-archive:php-73-ubi8-newer.tar.gz \
docker://quay.io/${RHT_OCP4_QUAY_USER}/php-73-ubi8: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-73-ubi8 imported

Name:          php
...output omitted...
latest
tagged from quay.io/youruser/php-73-ubi8

* quay.io/youruser/php-73-ubi8@sha256:3f5e...6ab7
  Less than a second ago
quay.io/youruser/php-73-ubi8@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.

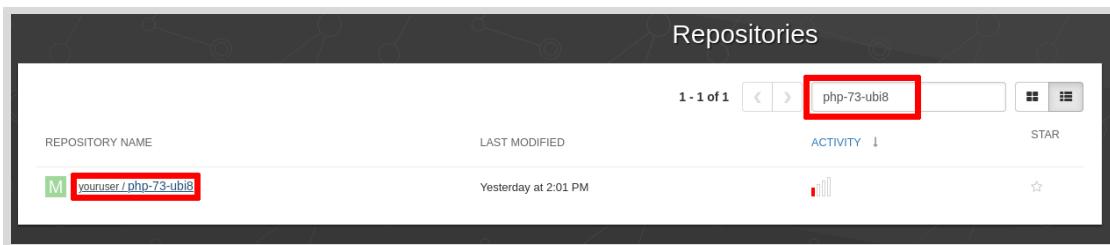
6.1. Change to the home directory.

```
[student@workstation trigger-builds]$ cd
```

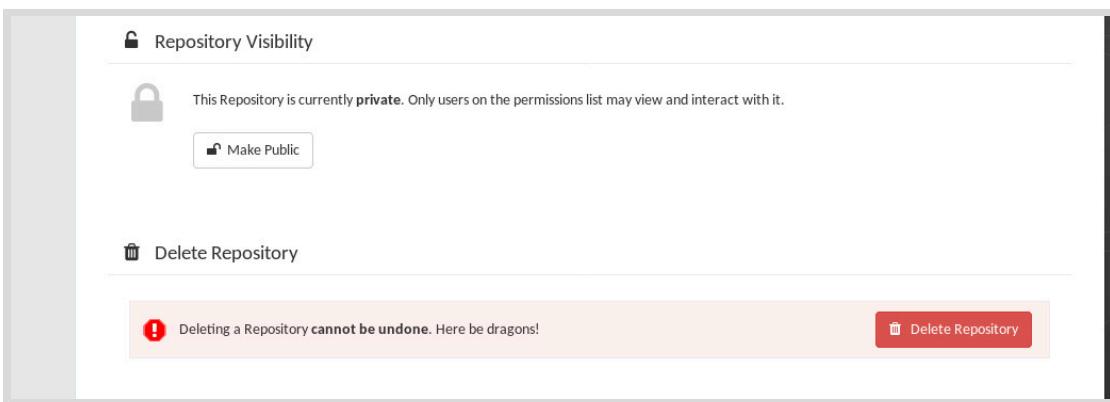
6.2. Delete the container image from the external registry:

```
[student@workstation ~]$ skopeo delete \
docker://quay.io/${RHT_OCP4_QUAY_USER}/php-73-ubi8
```

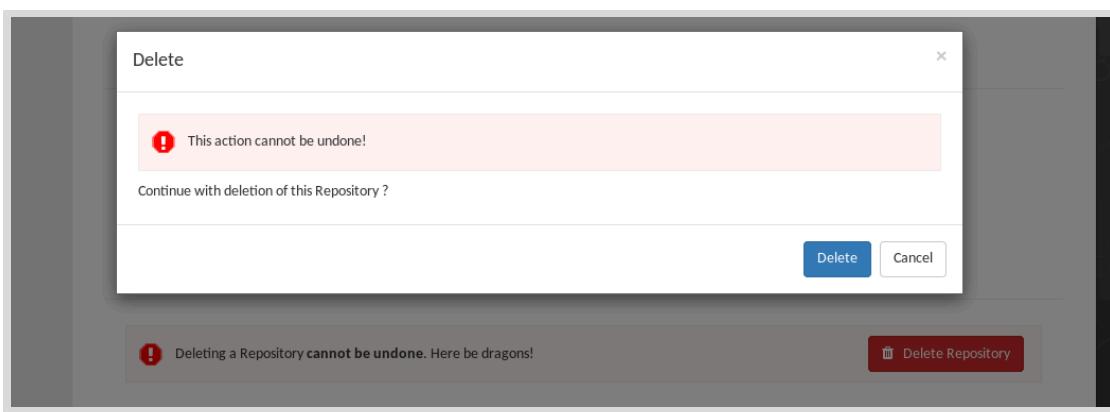
- 6.3. Log in on Quay.io using your personal free account.
- 6.4. On Quay.io's top-level menu, click **Repositories** and filter repositories with the **php-73-ubi8** name. Then, click the **php-73-ubi8** repository.



- 6.5. On the **Repository Activity** page for the **php-73-ubi8** repository, click the gear icon. Scroll down until you find the **Delete Repository** button and click it.



- 6.6. In the **Delete** window, click **Delete** to confirm you want to delete the **php-73-ubi8** repository.



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

Red Hat OpenShift Container Platform (RHOC) 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, RHOC either will or will not push the image to the registry. If the command returns a non-zero exit code, indicating failure, RHOC does not push the image and marks the build as failed. If the command succeeds, RHOC 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
```



Note

Post-commit hooks are only intended to validate an image, never to modify it.

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 RHOC 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 RHOCP 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 4.6 at
https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6/html-single/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 Red Hat 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
```

Instructions

► 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 Red Hat 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)
deployment/builds-for-managers deploys istag/builds-for-managers:latest
  deployment #1 deployed 5 minutes ago - 1 pod
...output omitted...
```

▶ 2. Create a new application.

- 2.1. 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 --name hook --context-dir post-commit \
php:7.3-http://github.com/${RHT_OCP4_GITHUB_USER}/D0288-apps
```

- 2.2. Wait for the build to finish:

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

- 2.3. Wait for the application to be ready and running:

NAME	READY	STATUS	RESTARTS	AGE
builds-for-managers-1-w1s8f	1/1	Running	0	8m
hook-1-build	0/1	Completed	0	1m
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. Open a web browser to access `http://builds-for-managers-youruser-post-commit.apps.cluster.domain.example.com`.

```
[student@workstation ~]$ firefox $(oc get route/builds-for-managers \
-o jsonpath='{.spec.host}') &
```

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 Red Hat 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 for OpenShift

In this lab, you will use Red Hat 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 Red Hat 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 a Red Hat 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.
```

Instructions

1. Create the `youruser-build-app` project.
2. Execute the `/home/student/D0288/labs/build-app/oc-new-app.sh` script 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.

3. Verify that the application build fails and fix the problem.
4. Expose the application service for external access and obtain the route URL.
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.
6. Use the generic webhook for the build configuration to start a new application build.
7. Grade your work:

```
[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.

► Solution

Building Applications for OpenShift

In this lab, you will use Red Hat 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 Red Hat 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 a Red Hat 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`.

Instructions

1. Create the `youruser-build-app` project.

- 1.1. Load your classroom environment configuration.

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

- 1.2. Log in to Red Hat 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 to host the application:

```
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-build-app
```

2. Execute the `/home/student/D0288/labs/build-app/oc-new-app.sh` script 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.

- 2.1. Review the script that creates the application:

```
[student@workstation ~]$ cat ~/D0288/labs/build-app/oc-new-app.sh
...output omitted...
oc new-app --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
```

- 2.2. Execute the `oc-new-app.sh` script to create the application:

```
[student@workstation ~]$ ~/D0288/labs/build-app/oc-new-app.sh
...output omitted...
--> Creating resources ...
...output omitted...
--> Success
...output omitted...
```

3. Verify that the application build fails and 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...
error: build error: error building at STEP "RUN /usr/libexec/s2i/assemble": error
while running runtime: exit status 1
...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:

```
[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
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-2-build 0/1     Completed  0          4m5s
simple-964487d7b-fs9gr 1/1     Running   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 ~]$ SECRET=$(oc get bc simple \
-o jsonpath='{.spec.triggers[*].generic.secret}{"\n"}')
```

- 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/$SECRET/generic
...output omitted...
"status": "Success",
...output omitted...
```

**Note**

The preceding command contains no spaces after the first line. If the curl command fails, verify that:

- Your URL contains no spaces.
- The RHT_OCP4_MASTER_API variable does not end in a slash (/). If it does, then use \${RHT_OCP4_MASTER_API%} in the preceding command to strip the slash.
- Your \$SECRET variable contains the correct secret.

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         94m
simple-2-build 0/1    Completed  0         36m
simple-3-build 0/1    Completed  0         19m
simple-7c6995cdcf-phvk5 1/1    Running   0         16m
```

7. Grade your work:

```
[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 Base 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

Red Hat OpenShift (RHOCP) 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 Red Hat 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 RHOCP 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

RHOCP 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:

1. RHOCP accesses the URL as a container registry URL. If this succeeds, there is no need to create a build configuration. RHOCP creates the deployment and other resources required to deploy a container image.
2. If the URL points to a Git repository, RHOCP 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. RHOCP 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, RHOCP 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 RHOCP 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 RHOCP 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 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 RHOCP 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 https://github.com/RedHatTraining/D0288-apps/ubi-echo
```

If your RHOCP cluster finds one of the files mentioned in the previous table in the root directory of the repository, then the RHOCP 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 -i php:7.3 \
https://github.com/RedHatTraining/D0288-apps/php-helloworld
```

The RHOCP 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 RHOCP 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 <code>assemble</code> script builds the application from source and places it into appropriate directories inside the image.

Script	Mandatory?	Description
run	Yes	The <code>run</code> script executes your application. It is recommended to use the <code>exec</code> command when running any container processes in your <code>run</code> script. This ensures signal propagation and graceful shutdown of any process launched by the <code>run</code> script.
save-artifacts	No	The <code>save-artifacts</code> 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 <code>usage</code> script provides a description of how to properly use your image.
test/run	No	The <code>test/run</code> 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. RHOCP instantiates a container based on the S2I builder image, and then creates a tar file containing the source code and the S2I scripts. RHOCP then streams the tar file into the container.
2. Before running the `assemble` script, RHOCP 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 RHOCP sets the `run` script as the CMD instruction for the final image.

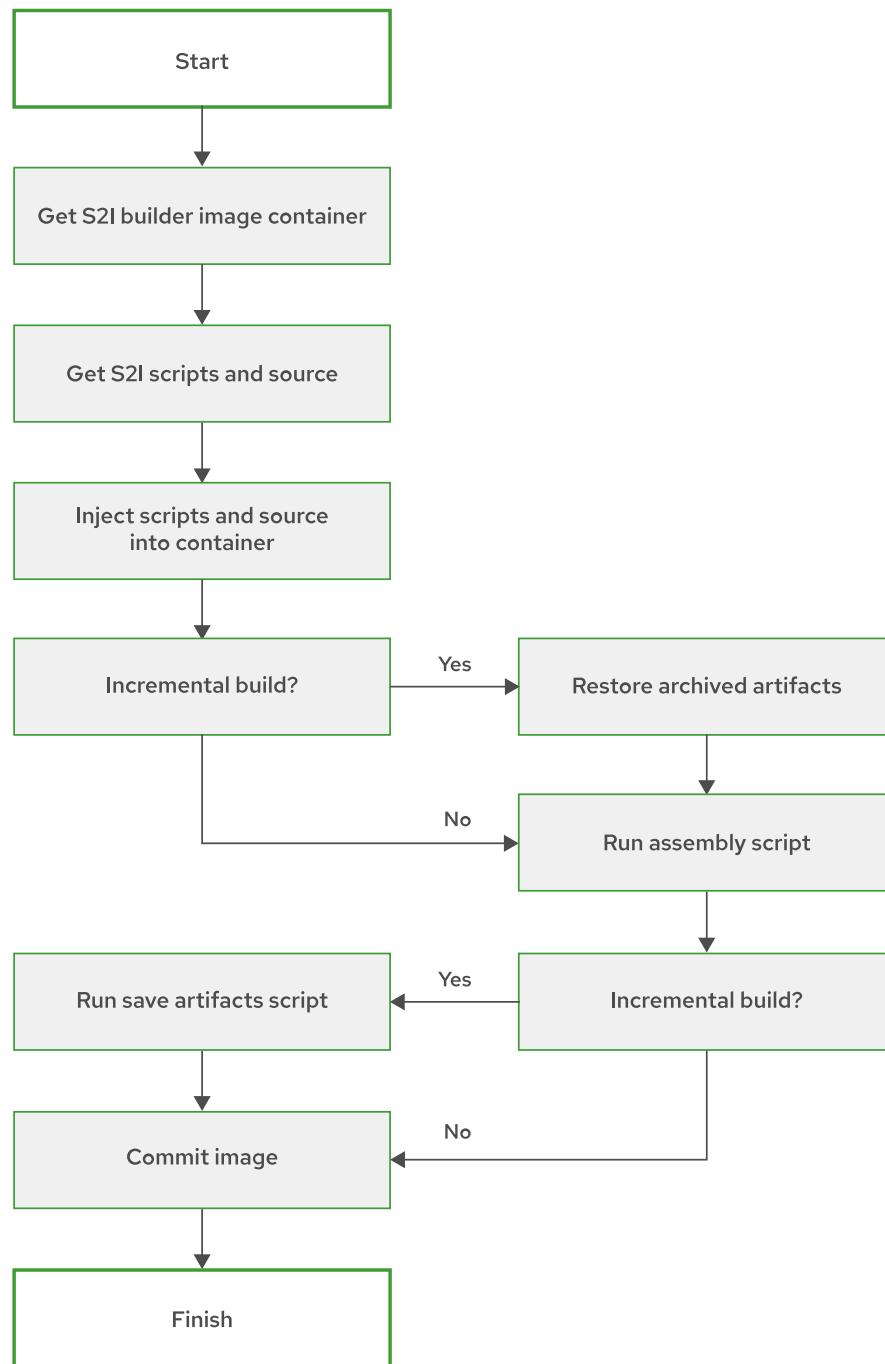


Figure 5.1: The S2I build workflow

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" \
      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 && \
    yum install -y --nodos nginx && \
    yum clean all

EXPOSE 8080④

COPY ./s2i/bin/ /usr/libexec/s2i⑤

```

- ① Set the labels that are used for RHOCP 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 RHOCP starts the S2I process, it inspects the source code folder, the custom S2I scripts, and the application source code. RHOCP includes all of these files in the tar file injected into the S2I builder image. RHOCP 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 RHOCP S2I builder images is available in the *Images* guide, in the documentation for RHOCP 4.6 at

https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6/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 container registry to the local system. Use the `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 ~]$ podman pull \
myregistry.example.com/rhscl/php-73-rhel7
...output omitted...
Digest: sha256:...
[user@host ~]$ 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 a Red Hat OpenShift (RHOCP) cluster using S2I, it is 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
...output omitted...
if [ -d /tmp/artifacts/.m2 ]; then
    echo "---> Restoring maven artifacts..."
    mv /tmp/artifacts/.m2 ${HOME}/
fi
...output omitted...
```


References

Further information is available in the *Creating Images* chapter of the *Images* guide for RHOC 4.6 at
https://access.redhat.com/documentation/en-us/openShift_container_platform/4.6/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 Red Hat OpenShift (RHOCUP) 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
```

Instructions

- 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 ~]$ 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 main 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 main
...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 a RHOCUP 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 RHOCP 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 \
--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
deployment.apps "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 RHOCP 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          105s
bonjour-7bc86dd97-wjvhx  1/1     Running    0          72s
```

- 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, 2021 @ 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 deployment/bonjour
...output omitted...
[Fri Nov 03 16:12:21.690941 2021] [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 2021] [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 2021] [ssl:debug] [pid 9] ssl_engine_init.c(270):
AH01886: SSL FIPS mode disabled
...output omitted...
[Fri Nov 03 16:12:21.745520 2021] [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 2021] [core:notice] [pid 9] AH00094: Command line:
'httdp -D FOREGROUND -e debug'
...output omitted...
10.131.0.16 - - [03/Nov/2021:16:28:53 +0000] "GET / HTTP/1.1" 200 28 "-"
"curl/7.29.0"
10.128.2.216 - - [03/Nov/2021:16:29:03 +0000] "GET /info.html HTTP/1.1" 200 87 "-"
"curl/7.29.0"
```

- ▶ 5. Cleanup. Delete the project:

```
[student@workstation D0288-apps]$ cd ~
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-s2i-scripts
project.project.openshift.io "youruser-redhat-com-s2i-scripts" deleted
```

- ▶ 6. Remove the test container you created earlier to view the default S2I scripts.

```
[student@workstation ~]$ podman rm test
925b932059df9e327bfffffe1964c7a1a5a45d13d872b2fb67e333b63166923ce3
```

Your ID value will differ from above.

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

Completing Guided Exercise: Customizing S2I Builds

· Log in on OpenShift..... SUCCESS
· Git repo '/home/student/D0288-apps' has no pending changes.. SUCCESS

Please use start if you wish to do the exercise again.
```

This concludes the guided exercise.

Creating an S2I Base 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 Red Hat OpenShift Container Platform (RHOCP) cluster.

Before deploying the S2I builder image to an RHOCP 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 RHOCP cluster.

Installing the S2I Tool

Since RHEL 7, 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` or `rhel-server-rhscl-8-rpms` yum repository depending on your RHEL version.

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 RHOCP 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 ~]$ 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 RHOCP 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 RHOCP 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 RHOCP 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 RHOCP, 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 RHOCOP 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 RHOCOP 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 RHOCOP.

Building an Nginx S2I Builder Image

To create an Nginx S2I builder image based on RHEL 8, 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 \
    && 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 RHOCP 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 ~]$ 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 ~]$ 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 RHOCP cluster:

```
[user@host ~]$ 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 RHOCP

Use the `skopeo` command to push the S2I builder image to a container registry, and create an image stream in RHOCP 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 ~]$ skopeo copy containers-storage:localhost/s2i-do288-httd \
docker://quay.io/${RHT_OCP4_QUAY_USER}/s2i-do288-httd
```

- 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 RHOCP `oc new-app` command:

```
[user@host ~]$ oc new-app --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 RHOCP 4.6 at
https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6/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 Base 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 a Red Hat OpenShift Container Platform (RHOCP) cluster using the S2I builder image.

Before You Begin

To perform this exercise, ensure you have access to:

- A running RHOCP cluster.
- The parent image (`ubi8/ubi`) for the sample application.
- The `html-helloworld` application in the `D0288-apps` 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
```

Instructions

- 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 v1.3.1
```

- 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-d0288-httpd s2i-d0288-httpd
```

- 2.2. Verify that the template files have been created. The `s2i create` command creates the following directory structure:

```
[student@workstation ~]$ tree 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
```



Note

The `s2i` command generates a *Containerfile* with the legacy name of *Dockerfile*. *Containerfile* is the preferred name to use.

- 3. Create the Apache HTTP server S2I builder image.

- 3.1. An example Containerfile for the Apache HTTP server builder image is provided for you at `~/D0288/labs/apache-s2i/Containerfile`. Briefly review this file:

```
[student@workstation ~]$ cat ~/D0288/labs/apache-s2i/Containerfile
FROM registry.access.redhat.com/ubi8/ubi:8.4 ①

# Generic labels
LABEL Component="httpd" \
      Name="s2i-do288-httdp" \
      Version="1.0" \
      Release="1"

# Labels consumed by RHOCP
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" ③

# This label is used to categorize this image as a builder image in the
# RHOCP 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 && \ ④
```

```

yum clean all --disableplugin=subscription-manager -y && \
echo "This is the default index page from the s2i-do288-httpd S2I builder
image." > ${DOCROOT}/index.html 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 RHOCP 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 Containerfile to the `~/s2i-do288-httpd` directory and overwrite the generated template Containerfile:

```

[student@workstation ~]$ rm ~/s2i-do288-httpd/Dockerfile
[student@workstation ~]$ cp ~/D0288/labs/apache-s2i/Containerfile
~/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/
'D0288/labs/apache-s2i/s2i/bin/assemble' -> 's2i-do288-httpd/s2i/bin/assemble'
'D0288/labs/apache-s2i/s2i/bin/usage' -> 's2i-do288-httpd/s2i/bin/usage'
'D0288/labs/apache-s2i/s2i/bin/run' -> 's2i-do288-httpd/s2i/bin/run'

```

- 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 Podman should use the Containerfile in the current directory to build the image:

```
[student@workstation ~]$ cd s2i-do288-httdp
[student@workstation s2i-do288-httdp]$ podman build -t s2i-do288-httdp .
STEP 1: FROM registry.access.redhat.com/ubi8/ubi:8.4
...output omitted...
STEP 14: COMMIT s2i-do288-httdp
...output omitted...
```

3.5. Verify that the builder image was created:

```
[student@workstation s2i-do288-httdp]$ podman images
REPOSITORY                      TAG      IMAGE ID      CREATED
localhost/s2i-do288-httdp        latest   82beb27428b7  9 seconds ago
registry.access.redhat.com/ubi8/ubi 8.4     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 Containerfile generated by the `s2i build` command.

```
[student@workstation s2i-do288-httdp]$ mkdir ~/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/Containerfile
Application dockerfile generated in /home/student/s2i-sample-app/Containerfile
```

- 4.4. Review the generated application directory.

```
[student@workstation s2i-do288-httdp]$ cd ~/s2i-sample-app
[student@workstation s2i-sample-app]$ tree .
.
├── Containerfile
└── downloads
    └── defaultScripts
```

```

|   └── scripts
└── upload
    ├── scripts
    └── src
        └── index.html

6 directories, 2 files

```

Observe the `index.html` file located in the `upload/src` 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 Containerfile.

```
[student@workstation s2i-sample-app]$ cat Containerfile
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 RHOCP 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 Containerfile.

```
[student@workstation s2i-sample-app]$ podman build \
-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 9: COMMIT s2i-sample-app
...output omitted...
```

4.7. Verify that the application container image was created:

```
[student@workstation s2i-sample-app]$ podman images
REPOSITORY           TAG      IMAGE ID      CREATED
localhost/s2i-sample-app    latest   3c8637c4372d  About an hour ago
localhost/s2i-do288-httdp    latest   d06040a9eeca  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 RHOCP cluster. You can copy the command, or run it directly from the `~/D0288/labs/apache-s2i/local-test.sh` script:

```
[student@workstation s2i-sample-app]$ 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]$ podman ps
CONTAINER ID   IMAGE               COMMAND ...
a5f4b3e5bfaa   localhost/s2i-sample-app:latest   /bin/sh -c /usr/lib...
```

- 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]$ 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]$ 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]$ 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 RHOCP 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".
...output omitted...
```

- 6.2. 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
```

- 6.3. Link the new secret to the `builder` service account.

```
[student@workstation s2i-sample-app]$ oc secrets link builder quayio
```

- 6.4. 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...
```

6.5. 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 RHOCP 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 --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 RHOCP 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          71s
hello-s2i-57cb8dc96c-rnkgt  1/1     Running    0          50s
```

- 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]$ export APP_URL=$( \
oc get route/hello-s2i \
-o jsonpath='{.spec.host}{"\n"}')
[student@workstation s2i-sample-app]$ echo ${APP_URL}
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 ${APP_URL}
<html>
  <body>
    <h1>Hello, World!</h1>
  </body>
</html>
```

► 8. Clean up.

- 8.1. Delete the apache-s2i project in RHOCP:

```
[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]$ podman rm test
a5f4b3e5bf ...output omitted...
```

- 8.3. Delete the container images from the workstation VM:

```
[student@workstation s2i-sample-app]$ podman rmi -f \
localhost/s2i-sample-app \
localhost/s2i-do288-httdp \
registry.access.redhat.com/ubi8/ubi:8.4
...output omitted...
Untagged: localhost/s2i-sample-app:latest
...output omitted...
Untagged: localhost/s2i-do288-httdp:latest
...output omitted...
Untagged: registry.access.redhat.com/ubi8/ubi:8.4
```

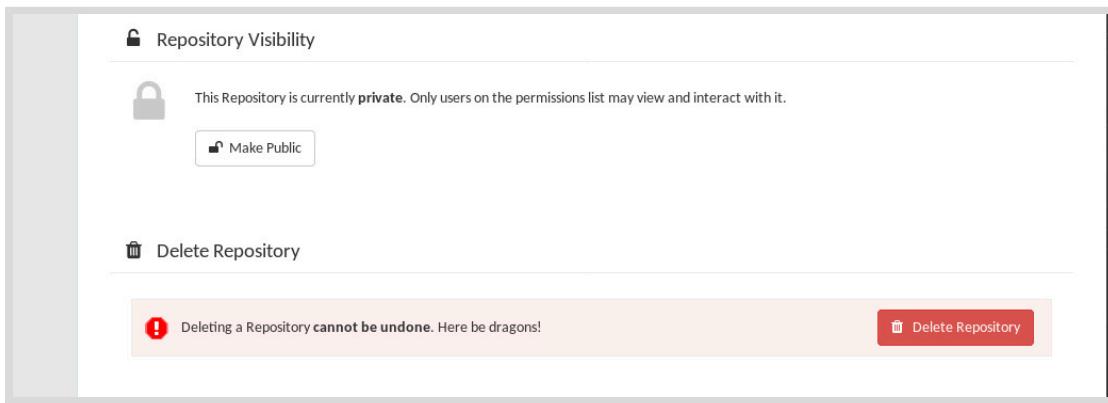
- 8.4. Delete the s2i-do288-httdp image from the external registry:

```
[student@workstation s2i-sample-app]$ skopeo delete \
docker://quay.io/${RHT_OCP4_QUAY_USER}/s2i-do288-httdp:latest
```

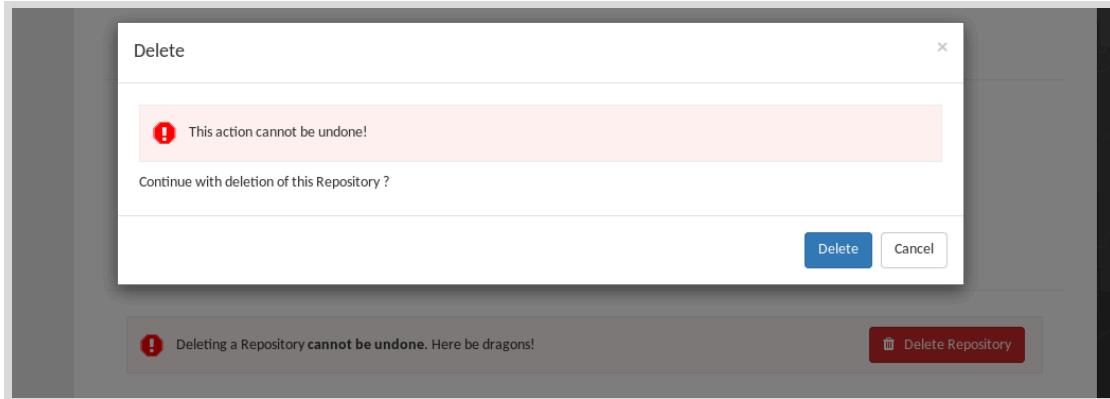
- 8.5. Log in to Quay.io using your personal free account.

 Navigate to <http://quay.io> and click **Sign In** to provide your user credentials.

- 8.6. On the Quay.io main menu, click **Repositories** and look for s2i-do288-httdp. The lock icon next to it indicates it is a private repository that requires authentication for both pulls and pushes. Click s2i-do288-httdp to display the **Repository Activity** page.
- 8.7. On the **Repository Activity** page for the s2i-do288-httdp 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-httdp 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

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 a Red Hat OpenShift Container Platform (RHOCP) 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 RHOCP 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 directory called `custom-s2i` inside the `/home/student/D0288/labs` directory. This directory contains the template directory 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 RHOCP 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 RHOCP 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` directory.
 - 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 RHOCP 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 RHOCP, 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!

Instructions

1. The S2I scripts for the builder image are provided in the `/home/student/D0288/labs/custom-s2i/s2i/bin` directory. Review the `assemble`, `run`, and `usage` scripts.
2. 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.
3. Build the S2I builder image. Name the image `s2i-do288-go`.
4. 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.
5. Push the `s2i-do288-go` S2I builder image to your personal Quay.io account.
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`.
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:
8. Create a new branch where you can save any changes you make during this exercise, and push it to Github:
9. Deploy and test the `go-hello` application from your personal GitHub fork of the `D0288-apps` repository to the classroom RHOC 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!

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.
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!

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:

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

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 a Red Hat OpenShift Container Platform (RHOCP) 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 RHOCP 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 directory called `custom-s2i` inside the `/home/student/D0288/labs` directory. This directory contains the template directory 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 RHOCP 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 RHOCP 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` directory.
 - 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 RHOCP 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 RHOCP, 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!
```

Instructions

1. The S2I scripts for the builder image are provided in the `/home/student/D0288/labs/custom-s2i/s2i/bin` directory. Review the `assemble`, `run`, and `usage` scripts.
2. 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.
 - 2.1. 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
```

3. Build the S2I builder image. Name the image `s2i-do288-go`.

- 3.1. Create the S2I builder image:

```
[student@workstation ~]$ cd ~/D0288/labs/custom-s2i
[student@workstation custom-s2i]$ 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
```

- 3.2. Verify that the builder image was created:

```
[student@workstation custom-s2i]$ podman images
REPOSITORY          TAG      IMAGE ID      ...
localhost/s2i-do288-go    latest   d1f856d10fa7  ...
...output omitted...
```

4. 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.

- 4.1. 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]$ 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
```

- 4.4. Verify that the application container image was created:

```
[student@workstation s2i-go-app]$ 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 RHOCP cluster. You can copy the command or run it directly from the `~/D0288/labs/custom-s2i/local-test.sh` script:

```
[student@workstation s2i-go-app]$ podman run --name go-test -u 1234 \
-p 8080:8080 -d s2i-go-app
18b903cfaae4...
```

- 4.6. Verify that the container started successfully:

```
[student@workstation s2i-go-app]$ 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]$ podman stop go-test
18b903cfaae4...
[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 ~]$ 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 ~]$ 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 RHOCP 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. If you are not already logged in, log in to your personal Quay.io account using Podman, so that you can export the `auth.json` file to use it in a pull secret in the next step.

```
[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 main
...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 RHOCP 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 --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
deployment.apps "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 RHOCP 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:

NAME	READY	STATUS	RESTARTS	AGE
greet-1-build	0/1	Completed	0	53s
greet-6986b8fcb-fn4rq	1/1	Running	0	14s

- 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 `D0288-apps/go-hello` directory. Create the directory:

```
[student@workstation ~]$ mkdir -p ~/D0288-apps/go-hello/.s2i/bin
```

- 10.2. Copy the `run` script in the S2I builder image from `~/D0288/labs/custom-s2i/s2i/bin/run`:

```
[student@workstation ~]$ cp ~/D0288/labs/custom-s2i/s2i/bin/run \
~/D0288-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 `~/D0288/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 RHOCUP 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
greet-1-build  0/1     Completed  0          36m
greet-2-build  0/1     Completed  0          50s
greet-6986b8fcf-fn4rq  1/1     Running   0          36m
...output omitted...
```

- 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 RHOCP.

```
[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 ~]$ podman rm go-test
```

- 13.3. Delete all the container images built during this lab on the **workstation** VM.

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

Deploying Multi-container Applications

Goal

Deploy multi-container applications using Helm charts and Kustomize.

Objectives

- Describe the elements of an OpenShift template.
- Build a multicontainer application using Helm Charts.
- Customize OpenShift deployments.

Sections

- Describing OpenShift Templates (and Quiz)
- Creating a Helm Chart (and Guided Exercise)
- Customizing Deployments with Kustomize (and Guided Exercise)

Lab

Deploying Multi-container Applications

Describing OpenShift Templates

Objectives

After completing this section, you should be able to convert a list of Red Hat OpenShift (RHOCP) resources into an RHOCP template.

Describing a Template

An RHOCP template is a YAML or JSON file consisting of a set of RHOCP resources. Templates define parameters that are used to customize the resource configuration. RHOCP processes templates by replacing parameter references with values and creating a customized set of resources.

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 RHOCP. The template contains configuration details about the containers that make up the product. They also contain a deployment configuration for the number of replicas, services and routes, persistent storage configuration, health checks, and resource limits for computing resources such as CPU, memory, and I/O.
- Your multitier application consists of a number of separate components, such as a web server, an application server, and a database. You should deploy these components together as a single unit on RHOCP to simplify the process of deploying the application in a staging environment. This allows your QA and acceptance testing teams to rapidly provision applications for testing.



Note

RHOCP templates are deprecated and will be removed in future RHOCP versions. Although still supported, RHOCP templates are covered for completeness, but Red Hat does not recommend its usage.

Template Syntax

The syntax of an RHOCP template follows the general syntax of an RHOCP 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 RHOCP 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 RHOCP 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 such as 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. RHOCP 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. RHOCP 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
```

RHOCP 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 RHOCP, 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 RHOCP 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 RHOCP Container Platform provides several templates in the `openshift` project. All RHOCP cluster users have read access to the `openshift` project, but only cluster administrators have permission to create or delete templates in this project.

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 RHOCP. If the template is meant to be reused by multiple developers, then it is better to create the template resource in a shared project. If the template is meant to be used by a single deployment, then it is better to keep it in a file.

The `oc new-app` command creates resources from the template, and the `oc process` command creates a resource list from the template. You must either save the resource list that is 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 by 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 applies values to a template and stores the results in a local 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 RHOCP 4.6 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 RHOCP 4.6 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 is available in the *Using Templates* chapter in the documentation for Red Hat OpenShift Container Platform 4.6 at
https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6/html-single/images/using-templates#using-templates

► Quiz

Describing OpenShift Templates

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 is the recommended way to create resources from a template?**

- a. oc export.
- b. oc new-app.
- c. oc create.
- d. oc process.
- 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 OpenShift Templates

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 is the recommended way to create resources from a template?**

- a. `oc export`.
- b. `oc new-app`.
- c. `oc create`.
- d. `oc process`.
- 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 Helm Chart

Helm Charts

Helm is an open source package manager for Kubernetes applications. It provides a way to package, share, and manage the life cycle of said Kubernetes applications.

A Helm Chart is a collection of files and templates that define a Helm application. These files are later packaged for distribution using Helm repositories.

Helm Life Cycle

The basic commands for the Helm CLI command to manage the life cycle of Helm Charts are:

Helm Commands

Command	Description
dependency	Manage a chart's dependencies
install	Install a chart
list	List releases installed
pull	Download a chart from a repository
rollback	Roll back a release to a previous revision
search	Search for a keyword in charts
show	Show information of a chart
status	Display the status of the named release
uninstall	Uninstall a release
upgrade	Upgrade a release

Helm Chart Structure

The `helm create` command creates the files needed in the correct structure. The files created by this command are the basic ones needed for a Helm Chart, which include the minimum templates required for an application to work.

A Helm chart consists primarily of two YAML files and a list of templates.

The YAML files are:

- `Chart.yaml`: Holds the Chart definition information.
- `values.yaml`: Holds the values that Helm uses in the default and user-created templates.

In addition to these two files, a Helm Chart holds several template files, these files are the basis for the Kubernetes resources that make up the application.

The basic structure of the `Chart.yaml` file is:

```
apiVersion: v2 ①
name: mychart ②
description: A Helm chart for Kubernetes ③
type: application ④
version: 0.1.0 ⑤
appVersion: "1.0.0" ⑥
```

- ① Version of the Helm API to use
- ② Name of the Helm Chart
- ③ Description of the Helm Chart
- ④ Helm Chart Type: Application or Library
- ⑤ Version of the Helm Chart
- ⑥ Version of the application this Chart packages

The `Chart.yaml` file can hold other optional values, one of the most important values is the `dependencies` section. The `dependencies` section holds a list of other Helm Charts that allow this Helm Chart to work.

The structure of the `dependencies` section is:

```
dependencies: ①
  - name: dependency1 ②
    version: 1.2.3 ③
    repository: https://examplerrepo.com/charts ④
  - name: dependency2
    version: 3.2.1
    repository: https://helmrepo.example.com/charts
```

- ① List of dependencies
- ② Name of the first required Helm Chart
- ③ Chart version to depend on
- ④ Repo holding the dependent Helm Chart

Chart values

Helm processes the template files in the chart and replaces the placeholders with the actual values during the processing of the chart. You can provide these values statically by using the `values.yaml` file or dynamically during packaging or installation by using the `--set` flag of the `helm` command-line tool.

The most common practice is to provide the majority of the values by using the `values.yaml` file and to leave the dynamic ones only for values that you must provide at installation time.

The following is an excerpt from the contents of this file:

```

...
image:
  repository: container.repo/name ①
  pullPolicy: IfNotPresent ②
  tag: "2.1" ③
...
serviceAccount:
  create: true ④
  annotations: {}
  name: "" ⑤
...
service:
  type: ClusterIP ⑥
  port: 80 ⑦

```

- ① Link to the application's container image
- ② Container pull policy in the Kubernetes cluster
- ③ Container tag to use, default value is the chart's `appVersion`
- ④ Create a service account for the application or not
- ⑤ Name of the service account, autogenerated if empty
- ⑥ Type of Kubernetes service
- ⑦ External application port

Templates

Helm uses templates to create at runtime the resources needed to deploy the application in the Kubernetes cluster. The Helm CLI tool creates some of these templates, but you can modify them or create new ones to suit your needs.

Helm uses the Go Template language to define the templates in the `templates` directory plus some other functions.

With a section of the common deployment `.yaml` template file you can see the use of placeholders and conditional sections:

```

metadata:
  name: {{ include "mychart.fullname" . }} ①
  labels:
    {{- include "mychart.labels" . | nindent 4 }}
spec:
  {{- if not .Values.autoscaling.enabled }} ②
  replicas: {{ .Values.replicaCount }} ③

```

```
{{{- end }}} ④
template:
spec:
  containers:
    - name: {{ .Chart.Name }} ⑤
```

- ①** The prefix for values from the `Chart.yaml` file is the name of the chart
- ②** Output the block if the value is false
- ③** The prefix for values from the `values.yaml` file is `Values`
- ④** End the conditional block
- ⑤** The prefix for the name of the chart is `Chart`



References

Helm

<https://helm.sh/>

Go Template Language

<https://golang.org/pkg/text/template/>

► Guided Exercise

Creating a Helm Chart

In this exercise, you will create a Helm chart for a multi-container application and deploy it to an Red Hat OpenShift (RHOCP) cluster.

Outcomes

You should be able to create a helm chart for the Famous Quotes application and its dependencies to an RHOCP cluster.

Before You Begin

To perform this exercise, ensure you have access to:

- A configured and running RHOCP cluster.
- The Helm CLI tool.

As the student user on the `workstation` machine, use the `lab` command to validate the prerequisites for this exercise:

```
[student@workstation ~]$ lab multicontainer-helm start
```

Instructions

► 1. Create a new Helm Chart.

1.1. Create the `famouschart` Helm chart.

Create a brand new Helm chart by using the `helm create` command and name it `famouschart`.

```
[student@workstation ~]$ cd ~/DO288/labs/multicontainer-helm  
[student@workstation multicontainer-helm]$ helm create famouschart  
Creating famouschart  
[student@workstation multicontainer-helm]$ cd famouschart
```

1.2. Verify the file structure.

```
[student@workstation famouschart]$ tree .  
.  
├── charts  
├── Chart.yaml  
└── templates  
    ├── deployment.yaml  
    ├── _helpers.tpl  
    ├── hpa.yaml  
    ├── ingress.yaml  
    ├── NOTES.txt  
    └── serviceaccount.yaml
```

```
|   └── service.yaml  
|   └── tests  
|       └── test-connection.yaml  
└── values.yaml
```

► 2. Configure application deployment.

Use the `values.yaml` file to configure the resulting deployment for the application.

2.1. Configure image and version values.

Update the `values.yaml` file, setting the `repository` property of the `image` section to the `quay.io/redhattraining/famous-quotes` image, and set the `tag` property of the same section to the `2.1` value for selecting the appropriate version of the application.

The corresponding section of the `values.yaml` file should look like:

```
image:  
  repository: quay.io/redhattraining/famous-quotes  
  pullPolicy: IfNotPresent  
  tag: "2.1"
```

2.2. Make sure that the container uses the correct port to connect to the application.

In the `templates/deployment.yaml` file, change the `containerPort` property included within the `containers` section so that it uses the value `8000`.

The corresponding section of the `templates/deployment.yaml` file should look like:

```
ports:  
  - name: http  
    containerPort: 8000  
    protocol: TCP
```

► 3. Add the database dependency.

The Famous Quotes application uses a database to store the quotes, so you must provide a database alongside the application for it to work. To achieve this, you must provide the dependency for the application chart and configure the database chart.

3.1. Add the `mariadb` dependency to the application chart.

To do this, add the following snippet at the end of the `Chart.yaml` file:

```
dependencies:  
  - name: mariadb  
    version: 9.3.11  
    repository: https://charts.bitnami.com/bitnami
```

You can add this by appending the contents of the `~/DO288/labs/multicontainer-helm/dependencies.yaml` to the `Chart.yaml` file:

```
[student@workstation famouschart]$ cat ../dependencies.yaml >> Chart.yaml
```

3.2. Update the dependencies for the chart.

This downloads the charts added as dependencies and locks its versions.

```
[student@workstation famouschart]$ helm dependency update
Getting updates for unmanaged Helm repositories...
...Successfully got an update from the "https://charts.bitnami.com/bitnami" chart
repository
Saving 1 charts
Downloading mariadb from repo https://charts.bitnami.com/bitnami
Deleting outdated charts
```

- 3.3. Set up the database to use custom values for authentication and security.

To pass the same values to the deployed application, you must control its values instead of letting the database helm chart create ones at random.

Add the following lines at the end of the `values.yaml` file:

```
mariadb:
  auth:
    username: quotes
    password: quotespwd
    database: quotesdb
  primary:
    podSecurityContext:
      enabled: false
    containerSecurityContext:
      enabled: false
```

You can add this by appending the contents of the `~/DO288/Labs/multicontainer-helm/mariadb.yaml` to the `values.yaml` file:

```
[student@workstation famouschart]$ cat ../mariadb.yaml >> values.yaml
```

- 4. Configure the application's database access by using environmental variables.

- 4.1. The default deployment template does not pass any environmental variable to the deployed applications. Modify the `templates/deployment.yaml` template to pass the environmental variables defined in the `values.yaml` file to the application's container, add the following snippet after the `imagePullPolicy` value in the `containers` section:

```
imagePullPolicy: {{ .Values.image.pullPolicy }}
env:
{{- range .Values.env }}
- name: {{ .name }}
  value: {{ .value }}
{{- end }}
```

**Warning**

When dealing with YAML files, make sure that the indentation is correct: the `imagePullPolicy` and `env:` lines must be at the same level, and the `name` and `value` entries must be at the same level, deeper than the `env:` entry. The `-` in the `- name` entry must be at the same level or deeper than the `env:` entry.

- 4.2. Add the appropriate environmental variables at the end of the `values.yaml` file:

```
env:
  - name: "QUOTES_HOSTNAME"
    value: "famousapp-mariadb"
  - name: "QUOTES_DATABASE"
    value: "quotesdb"
  - name: "QUOTES_USER"
    value: "quotes"
  - name: "QUOTES_PASSWORD"
    value: "quotespwd"
```

You can add this by appending the contents of the `~/DO288/labs/multicontainer-helm/env.yaml` to the `values.yaml` file:

```
[student@workstation famouschart]$ cat ./env.yaml >> values.yaml
```

5. Deploy the application using the Helm chart.

- 5.1. Create a project in RHOCP

```
[student@workstation famouschart]$ source /usr/local/etc/ocp4.config
[student@workstation famouschart]$ oc login -u ${RHT_OCP4_DEV_USER} \
-p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
[student@workstation famouschart]$ oc new-project \
${RHT_OCP4_DEV_USER}-multicontainer-helm
```

- 5.2. Use the `helm install` command to deploy the application in the RHOCP cluster:

```
[student@workstation famouschart]$ helm install famousapp .
NAME: famousapp
LAST DEPLOYED: Thu May 20 19:12:09 2021
NAMESPACE: youruser-multicontainer-helm
STATUS: deployed
REVISION: 1
NOTES:
...output omitted...
```

This command creates an RHOCP deployment called `famousapp`. To verify the status of this deployment, use the `oc get deployments` command:

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
<code>famousapp-famouschart</code>	0/1	1	1	10s

It might take a while for the `mariadb` pod to be fully available for the application, hence the not Ready state of the deployment.

Use the `oc get pods` command several times to verify that the application and the database are correctly deployed:

```
[student@workstation famouschart]$ oc get pods
NAME                               READY   STATUS    RESTARTS   AGE
famousapp-famouschart-7bff544d88-f289l   1/1     Running   3          5m
famousapp-mariadb-0                  1/1     Running   0          5m
```

▶ 6. Test the application.

6.1. Expose the application's service:

```
[student@workstation famouschart]$ oc expose service famousapp-famouschart
route.route.openshift.io/famousapp-famouschart exposed
```

6.2. Call the /random endpoint of the deployed application:

```
[student@workstation famouschart]$ FAMOUS_URL=$(oc get route \
-n ${RHT_OCP4_DEV_USER}-multicontainer-helm famousapp-famouschart \
-o jsonpath='{.spec.host}'/random)
[student@workstation famouschart]$ curl $FAMOUS_URL
8: Those who can imagine anything, can create the impossible.
- Alan Turing
```

This endpoint returns a random quote from the database, so it is very likely that you will see a different quote. Execute again the call to verify this random behavior.

```
[student@workstation famouschart]$ curl $FAMOUS_URL
1: When words fail, music speaks.
- William Shakespeare
```

Finish

On the workstation VM, run the `lab multicontainer-helm 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 famouschart]$ cd ~
[student@workstation ~]$ lab multicontainer-helm finish
```

This concludes the guided exercise.

Customizing a Deployment with Kustomize

Kustomize

Kustomize is a tool that customizes Kubernetes resources for different environments or needs. Kustomize is a template free solution that helps reuse configurations and provides an easy way to patch any resource.

The most common use case is the need to define different resources for different environments such as development, staging, and production. But it is not limited to that. It is up to you to decide what defines an environment in your use case, which environments you have, and what different configurations each environment needs.

Kustomize separates these configuration sets into two types: base and overlays.

Base holds all configuration and resources common to all the derivatives. Overlays represent the differences from the base on a specific environment. Kustomize uses directories to represent these configuration sets.

An example of a Kustomize layout could be:

```
myapp
└── base
└── overlays
    ├── production
    └── staging
```

Kustomization file

For each configuration set Kustomize needs a `kustomization.yaml` file that can contain:

- Resource files to include
- Base configuration set to start from
- Resources that patch the base configuration
- Common resources definitions that apply to any configuration set depending on this one

This file must reside in the directory of the configuration set.

Resources files

The resources section of the `kustomization.yaml` file is a list of files that combined create all the resources needed for this specific environment. For example:

```
resources:
- deployment.yaml
- secrets.yaml
- service.yaml
```

The layout for a base definition as represented with these three resources would look like:

```
myapp
└── base
    ├── deployment.yaml
    ├── kustomization.yaml
    ├── secrets.yaml
    └── service.yaml
```

Separating Kubernetes objects definition into smaller files helps to maintain the base set if it does not come from a single external source.

Base configuration

The `kustomization.yaml` file of an overlay needs to point to the base configuration sets that are the starting point. For example:

```
bases:
- ../../base
```

If you follow the expected directory structure, then the relative path is the most portable solution.

This overlay starts with the base configuration and then applies any patch defined in it.

Common resources

Kustomize allows adding new configurations in all the resources that derive from a base set.

You can set Labels and Annotations with the `commonLabels` and `commonAnnotations` sections of a `kustomization.yaml` file.

For example, if you would like to add the `origin=kustomize` label to the base set and all the overlays that depend on it, then you must add this to the `base/kustomization.yaml`:

```
commonLabels:
  origin: kustomize
```

Patches

Each overlay can provide any number of modifications to the base configuration through the list of patches to apply. First you add the reference to the patch file in the `kustomization.yaml` file of the overlay:

```
patches:
- replica_count.yaml
```

And then, in the patch file provided, you need a way to identify the resource to change and the new value:

```

apiVersion: apps/v1
kind: Deployment 1
metadata:
  name: myapp 2
spec:
  replicas: 5 3

```

1 **2** Kind and name provide a unique way to identify the resource

3 New value in this overlay

Applying customizations

You can use Kustomize as a standalone tool with the `kustomize` command-line tool, or as of Kubernetes 1.14, as part of the `kubectl apply` or `oc apply` commands. These tools only need the location of the folder with the configuration set to apply.

Here you can see an example of the standalone way:

```
[user@host ~]$ kustomize build myapp/base | oc apply -f -
```

This is an example of the integration in the cluster management tools:

```
[user@host ~]$ oc apply -k myapp/base
```

And an example of applying the overlay modifications:

```
[user@host ~]$ oc apply -k myapp/overlays/staging
```

Kustomize is only available as part of the `kubectl apply` or `oc apply` commands because it is an extension of the declarative management of Kubernetes objects, which is designed to work in combination with source version control systems. The declarative management style works by defining the expected state of the objects and can create or modify an object depending on the existence or nonexistence of that object.

If you have the source file of the Kubernetes object available to modify, be it because you made a mistake or need to change something, then it is much cleaner and faster to modify the file and use `oc apply -f myobject.yaml` than to delete and recreate it again. The changes are also persisted in the file for future uses.

The imperative management of Kubernetes objects uses the `kubectl` and `oc` commands such as `create`, `delete`, `edit`, and `set`; and fail if the operation can not be performed. For example, if you try to create an object that already exists then the tool will raise an error. This management style is intended for manual operations and live interaction with the cluster and not for automated tools.



References

Kustomize - Kubernetes native configuration management

<https://kustomize.io/>

Kubernetes Object Management

<https://kubernetes.io/docs/concepts/overview/working-with-objects/object-management/>

► Guided Exercise

Customizing Deployments with Kustomize

In this exercise, you will customize an Red Hat OpenShift deployment using Kustomize.

Outcomes

You should be able to customize the Famous Quotes application deployment and deploy it to an RHOCP cluster.

Before You Begin

To perform this exercise, ensure you have access to:

- A configured and running RHOCP cluster.

As the student user on the workstation machine, use the `lab` command to validate the prerequisites for this exercise:

```
[student@workstation ~]$ lab multicontainer-kustomize start
```

Instructions

You are going to set up three environments: Development, Staging, and Production. These three environments have different needs in performance and number of running pods.

Find the requirements of each environment in the following table:

Environment	Pods	Memory	CPU Limit
Development	1	128Mi	250m
Staging	2	256Mi	500m
Production	5	512Mi	1000m

► 1. Review the Famous Quotes deployment file.

The `famous-quotes.yaml` file contains all the Resources needed to deploy the Famous Quotes application used in the Creating a Helm Chart guided exercise.

```
[student@workstation ~]$ cd ~/DO288/labs/multicontainer-kustomize
[student@workstation multicontainer-kustomize]$ cat famous-quotes.yaml
...
containers:
  - name: famouschart
    securityContext:
```

```
{
  image: "quay.io/redhattraining/famous-quotes:2.1"
}
```

► 2. Create the Kustomize folder structure.

To customize an application's resources file, you should structure the files so that it is easy to understand and reference.

2.1. Create the folder to hold all the Kustomize files.

```
[student@workstation multicontainer-kustomize]$ mkdir famous-kustomize
[student@workstation multicontainer-kustomize]$ cd famous-kustomize
```

2.2. Create the base reference.

Use the `famous-quotes.yaml` file as the base for the Kustomize structure.

```
[student@workstation famous-kustomize]$ mkdir base
[student@workstation famous-kustomize]$ cp ../famous-quotes.yaml \
base/deployment.yaml
```

2.3. Create the Kustomize description file for the base definition.

Create a new `kustomization.yaml` file in the base folder to hold the base Kustomize definition. Add the `deployment.yaml` file as a resource in the `kustomization.yaml` file of the base folder.

```
resources:
- deployment.yaml
```

► 3. Test the base definition.

To customize a deployment, first, you must verify that the base definition works as expected.

3.1. Create a project in RHOCP.

```
[student@workstation famous-kustomize]$ source /usr/local/etc/ocp4.config
[student@workstation famous-kustomize]$ oc login -u ${RHT_OCP4_DEV_USER} \
-p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
[student@workstation famous-kustomize]$ oc new-project \
${RHT_OCP4_DEV_USER}-multicontainer-kustomize
```

3.2. Apply the Kustomize base definition.

Apply the base definition by using the `oc apply` command:

```
[student@workstation famous-kustomize]$ oc apply -k base
serviceaccount/famous-quotes-service created
serviceaccount/famousapp-mariadb created
configmap/famousapp-mariadb created
secret/famousapp-mariadb created
service/famousapp-famouschart created
service/famousapp-mariadb created
```

```
deployment.apps/famousapp-famouschart created
statefulset.apps/famousapp-mariadb created
pod/famousapp-famouschart-test-connection created
```

This command creates an RHOCP deployment called `famousapp`. To verify the status of this deployment, use the `oc get deployments` command:

```
[student@workstation famous-kustomize]$ oc get deployments
NAME          READY   UP-TO-DATE   AVAILABLE   AGE
famousapp-famouschart   1/1     1           1          10s
```

It might take a while for the `mariadb` pod to be fully available for the application, wait until the deployment reaches the ready state.

3.3. Expose the application's service:

```
[student@workstation famous-kustomize]$ oc expose service famousapp-famouschart
route.route.openshift.io/famousapp-famouschart exposed
```

3.4. Call the /random endpoint of the deployed application:

```
[student@workstation famous-kustomize]$ FAMOUS_URL=$(oc get route \
-n ${RHT_OCP4_DEV_USER}-multicontainer-kustomize famousapp-famouschart \
-o jsonpath='{.spec.host}'/random)
[student@workstation famous-kustomize]$ curl $FAMOUS_URL
5: Imagination is more important than knowledge.
- Albert Einstein
```

You should see a random quote displayed for each request.

► 4. Create the Development environment definition.

4.1. Create the development overlay in the Kustomize folder.

```
[student@workstation famous-kustomize]$ mkdir -p overlays/dev
```

4.2. Adjust the policies of the Development environment.

Create a `replica_limits.yaml` file in the `overlays/dev` directory to hold the changes in the deployment definition:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: famousapp-famouschart
spec:
  replicas: 1
  template:
    spec:
      containers:
        - name: famouschart
          resources:
```

```
limits:
  memory: "128Mi"
  cpu: "250m"
```

- 4.3. Create the `overlays/dev/kustomization.yaml` file, add the base folder as the base definition, and the `replica_limits.yaml` file as a patch:

```
bases:
- ../../base
patches:
- replica_limits.yaml
```

- 4.4. Apply the Development overlay to the running instance of the Famous Quotes application:

```
[student@workstation famous-kustomize]$ oc apply -k overlays/dev
serviceaccount/famous-quotes-service unchanged
serviceaccount/famousapp-mariadb unchanged
configmap/famousapp-mariadb unchanged
secret/famousapp-mariadb unchanged
service/famousapp-famouschart unchanged
service/famousapp-mariadb configured
deployment.apps/famousapp-famouschart configured
statefulset.apps/famousapp-mariadb configured
pod/famousapp-famouschart-test-connection unchanged
```

- 4.5. Verify that the application still works correctly:

```
[student@workstation famous-kustomize]$ curl $FAMOUS_URL
4: Nothing that glitters is gold.
- Mark Twain
```

- 4.6. Verify that Kustomize applied the memory limit change:

```
[student@workstation famous-kustomize]$ oc get deployments famousapp-famouschart \
-o jsonpath='{.spec.template.spec.containers[0].resources.limits.memory}'
128Mi
```

► 5. Create the Staging environment.

- 5.1. Create the Staging environment by copying the Development environment:

```
[student@workstation famous-kustomize]$ cp -R overlays/dev overlays/stage
```

- 5.2. Modify the configuration in the `overlays/stage/replica_limits.yaml` file according to the values provided in the requirements table:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: famousapp-famouschart
```

```
spec:
  replicas: 2
  template:
    spec:
      containers:
        - name: famouschart
          resources:
            limits:
              memory: "256Mi"
              cpu: "500m"
```

- 5.3. Apply the Staging overlay to the running instance of the Famous Quotes application:

```
[student@workstation famous-kustomize]$ oc apply -k overlays/stage
serviceaccount/famous-quotes-service unchanged
serviceaccount/famousapp-mariadb unchanged
configmap/famousapp-mariadb unchanged
secret/famousapp-mariadb unchanged
service/famousapp-famouschart unchanged
service/famousapp-mariadb configured
deployment.apps/famousapp-famouschart configured
statefulset.apps/famousapp-mariadb configured
pod/famousapp-famouschart-test-connection unchanged
```

- 5.4. Verify that the application still works correctly:

```
[student@workstation famous-kustomize]$ curl $FAMOUS_URL
2: Happiness depends upon ourselves.
- Aristotle
```

- 5.5. Verify that Kustomize applied the memory limit change:

```
[student@workstation famous-kustomize]$ oc get deployments famousapp-famouschart \
-o jsonpath='{.spec.template.spec.containers[0].resources.limits.memory}'
256Mi
```

▶ 6. Create the Production environment.

- 6.1. Create the Production environment by copying the Development environment:

```
[student@workstation famous-kustomize]$ cp -R overlays/dev overlays/prod
```

- 6.2. Modify the configuration in the `overlays/prod/replica_limits.yaml` file according to the values provided in the requirements table:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: famousapp-famouschart
spec:
  replicas: 5
  template:
```

```
spec:
  containers:
    - name: famouschart
      resources:
        limits:
          memory: "512Mi"
          cpu: "1000m"
```

- 6.3. Apply the Production overlay to the running instance of the Famous Quotes application:

```
[student@workstation famous-kustomize]$ oc apply -k overlays/prod
serviceaccount/famous-quotes-service unchanged
serviceaccount/famousapp-mariadb unchanged
configmap/famousapp-mariadb unchanged
secret/famousapp-mariadb unchanged
service/famousapp-famouschart unchanged
service/famousapp-mariadb configured
deployment.apps/famousapp-famouschart configured
statefulset.apps/famousapp-mariadb configured
pod/famousapp-famouschart-test-connection unchanged
```

- 6.4. Verify that the application still works correctly:

```
[student@workstation famous-kustomize]$ curl $FAMOUS_URL
8: Those who can imagine anything, can create the impossible.
- Alan Turing
```

- 6.5. Verify that Kustomize applied the memory limit change:

```
[student@workstation famous-kustomize]$ oc get deployments famousapp-famouschart \
-o jsonpath='{.spec.template.spec.containers[0].resources.limits.memory}'
512Mi
```

Finish

On the workstation VM, run the `lab multicontainer-kustomize 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 famous-kustomize]$ cd ~
[student@workstation ~]$ lab multicontainer-kustomize finish
```

This concludes the guided exercise.

▶ Lab

Deploying Multi-container Applications

In this lab, you will create a Helm Chart for an application, and deploy it to an OpenShift cluster. Then you will customize the deployed application using Kustomize.



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 a Helm Chart for an application.
- Deploy the application to an OpenShift cluster.
- Use Kustomize to modify the deployment of the application.

Before You Begin

To perform this exercise, ensure you have access to:

- A running OpenShift cluster.
- An application container image in `quay.io/redhattraining/exoplanets:v1.0`.
- CockroachDB Helm Chart from repository <https://charts.cockroachdb.com/>.

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 multicontainer-review start
```

Requirements

The Exoplanets application is a web application that shows some information about extra solar planets. This application needs a PostgreSQL compatible database, such as CockroachDB to store the information and it has a prepackaged container image in Quay. To access the web application, use a web browser pointing to the exposed service.

Create the Helm Chart for the Exoplanets application and deploy it to the OpenShift cluster according to the following requirements:

- Use `exochart` as the Helm Chart name.
- Use the application container in `quay.io/redhattraining/exoplanets:v1.0`.
- Use the `cockroachdb` Helm Chart with version `6.0.4` from the repository <https://charts.cockroachdb.com/> as the database dependency.

- Use the port 8080 for the application.
- Use the following environmental variables to configure the database connection:

Database configuration environmental variables

Variable	Value
DB_HOST	exoplanets-cockroachdb
DB_PORT	26257
DB_USER	root
DB_NAME	postgres

- Use `youruser-multicontainer-review` as the RHOCP project name.
- Use `exoplanets` as the Helm Chart installation name.
- Use `exokustom` as the Kustomize directory.
- Use `helm template` command to create the base Kustomize definition from the Helm Chart.
- Use `test` as the directory for the Test overlay.
- Use the following resource limits for the Test overlay:

Environment	Pods	Memory	CPU Limit
Test	5	128Mi	250m

Instructions

1. Create a Helm Chart for the Exoplanets application named `exochart`.
2. Add the database dependency and configure the database using environmental variables.
3. Create an OpenShift project and deploy the application into the project using the Helm CLI tool.
4. Test the application deployment.

Expose the application service, get the address from the application's route, and use a browser to navigate to the application's address. A correct deployment of the application is similar to this image:

Exoplanets

The planets listed here are a small subset of the known planets found outside of our solar system. Mass and radius are listed in "Jupiter mass" and "Jupiter radius" units. The orbital period is measured in Earth days. The full dataset is available from the [Open Exoplanet Catalogue](#).

2M 0746+20 b

Mass	Radius	Period
30	0.97	4640

2M 2140+16 b

Mass	Radius	Period
20	0.92	7340

5. Create the base Kustomize directory structure and the deployment files. Create the Kustomize directory named exokustom in the ~/D0288/labs/multicontainer-review directory.



Note

You can use the `helm template` command to extract the object definitions from a Helm Chart into the base definition:

```
[student@workstation ~]$ helm template app-name helm-directory > base/deployment.yaml
```

6. Create and apply the Test overlay in the Kustomize directory, using a directory called `test`.
7. Test the application in the browser and verify that the custom values have been applied.

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 exokustom]$ lab multicontainer-review grade
```

Finish

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

```
[student@workstation exokustom]$ cd  
[student@workstation ~]$ lab multicontainer-review finish
```

This concludes the lab.

► Solution

Deploying Multi-container Applications

In this lab, you will create a Helm Chart for an application, and deploy it to an OpenShift cluster. Then you will customize the deployed application using Kustomize.



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 a Helm Chart for an application.
- Deploy the application to an OpenShift cluster.
- Use Kustomize to modify the deployment of the application.

Before You Begin

To perform this exercise, ensure you have access to:

- A running OpenShift cluster.
- An application container image in `quay.io/redhattraining/exoplanets:v1.0`.
- CockroachDB Helm Chart from repository <https://charts.cockroachdb.com/>.

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 multicontainer-review start
```

Requirements

The Exoplanets application is a web application that shows some information about extra solar planets. This application needs a PostgreSQL compatible database, such as CockroachDB to store the information and it has a prepackaged container image in Quay. To access the web application, use a web browser pointing to the exposed service.

Create the Helm Chart for the Exoplanets application and deploy it to the OpenShift cluster according to the following requirements:

- Use `exochart` as the Helm Chart name.
- Use the application container in `quay.io/redhattraining/exoplanets:v1.0`.
- Use the `cockroachdb` Helm Chart with version `6.0.4` from the repository <https://charts.cockroachdb.com/> as the database dependency.

- Use the port 8080 for the application.
- Use the following environmental variables to configure the database connection:

Database configuration environmental variables

Variable	Value
DB_HOST	exoplanets-cockroachdb
DB_PORT	26257
DB_USER	root
DB_NAME	postgres

- Use `youruser-multicontainer-review` as the RHOCP project name.
- Use `exoplanets` as the Helm Chart installation name.
- Use `exokustom` as the Kustomize directory.
- Use `helm template` command to create the base Kustomize definition from the Helm Chart.
- Use `test` as the directory for the Test overlay.
- Use the following resource limits for the Test overlay:

Environment	Pods	Memory	CPU Limit
Test	5	128Mi	250m

Instructions

1. Create a Helm Chart for the Exoplanets application named `exochart`.

- 1.1. Create the `exochart` Helm chart.

Create the Helm chart by using the `helm create` command and name it `exochart`.

```
[student@workstation ~]$ cd ~/DO288/labs/multicontainer-review
[student@workstation multicontainer-review]$ helm create exochart
Creating exochart
[student@workstation multicontainer-review]$ cd exochart
```

- 1.2. Configure application container image and version values.

Update the `values.yaml` file, setting the `repository` property of the `image` section to the `quay.io/redhattraining/exoplanets` image, and set the `tag` property of the same section to the `v1.0` value for selecting the appropriate version of the application.

The corresponding section of the `values.yaml` file should look like:

```
image:
  repository: quay.io/redhattraining/exoplanets
  pullPolicy: IfNotPresent
  tag: "v1.0"
```

2. Add the database dependency and configure the database using environmental variables.

- 2.1. Add the cockroachdb dependency to the application chart.

To do this, add the following snippet at the end of the `Chart.yaml` file:

```
dependencies:
- name: cockroachdb
  version: 6.0.4
  repository: https://charts.cockroachdb.com/
```

- 2.2. Update the dependencies for the chart.

This command downloads the charts added as dependencies and locks its versions.

```
[student@workstation exochart]$ helm dependency update
Getting updates for unmanaged Helm repositories...
...Successfully got an update from the "https://charts.cockroachdb.com/" chart
repository
Saving 1 charts
Downloading cockroachdb from repo https://charts.cockroachdb.com/
Deleting outdated charts
```

- 2.3. The default deployment template does not pass any environmental variables to the deployed applications. Modify the `templates/deployment.yaml` template to pass the environmental variables defined in the `values.yaml` file to the application's container, add the following snippet after the `imagePullPolicy` value in the `containers` section:

```
apiVersion: apps/v1
kind: Deployment
...output omitted...
spec:
  ...output omitted...
  template:
    ...output omitted...
    spec:
      ...output omitted...
      containers:
        - name: {{ .Chart.Name }}
          ...output omitted...
          imagePullPolicy: {{ .Values.image.pullPolicy }}
          env:
            {{- range .Values.env }}
            - name: "{{ .name }}"
              value: "{{ .value }}"
            {{- end }}
          ...output omitted...
```

- 2.4. Make sure that the container uses the correct port to connect to the application.

In the `templates/deployment.yaml` file, change the `containerPort` property included within the `containers` section so that it uses the value `8080`.

The corresponding section of the `templates/deployment.yaml` file should look like:

```
apiVersion: apps/v1
kind: Deployment
...output omitted...
spec:
  ...output omitted...
  template:
    ...output omitted...
    spec:
      ...output omitted...
      containers:
        - name: {{ .Chart.Name }}
          ...output omitted...
          ports:
            - name: http
              containerPort: 8080
              protocol: TCP
```

- 2.5. Add the appropriate environmental variables at the end of the `values.yaml` file:

```
env:
  - name: "DB_HOST"
    value: "exoplanets-cockroachdb"
  - name: "DB_NAME"
    value: "postgres"
  - name: "DB_USER"
    value: "root"
  - name: "DB_PORT"
    value: "26257"
```

3. Create an OpenShift project and deploy the application into the project using the Helm CLI tool.

- 3.1. Create a project in OpenShift.

```
[student@workstation exochart]$ source /usr/local/etc/ocp4.config
[student@workstation exochart]$ oc login -u ${RHT_OCP4_DEV_USER} \
-p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful.
...output omitted...
[student@workstation exochart]$ oc new-project \
${RHT_OCP4_DEV_USER}-multicontainer-review
```

- 3.2. Use the `helm install` command to deploy the application to the OpenShift cluster:

```
[student@workstation exochart]$ helm install exoplanets .
NAME: exoplanets
LAST DEPLOYED: Thu May 20 19:12:09 2021
NAMESPACE: youruser-multicontainer-review
STATUS: deployed
REVISION: 1
NOTES:
...output omitted...
```

Verify that the deployment of the application has started correctly:

```
[student@workstation exochart]$ oc get deployments
NAME          READY   UP-TO-DATE   AVAILABLE   AGE
exoplanets-exochart   0/1       1           1          10s
```

Wait for the three cockroachdb and the application pods to be fully available for the application. Use the `oc get pods` command several times to verify that the application and the database are correctly deployed:

```
[student@workstation exochart]$ oc get pods
NAME                           READY   STATUS    RESTARTS   AGE
exoplanets-cockroachdb-0      1/1     Running   0          1m
exoplanets-cockroachdb-1      1/1     Running   0          1m
exoplanets-cockroachdb-2      1/1     Running   0          1m
exoplanets-exochart-7bff544d88-f289l   1/1     Running   3          1m
```

4. Test the application deployment.

Expose the application service, get the address from the application's route, and use a browser to navigate to the application's address. A correct deployment of the application is similar to this image:

Exoplanets

The planets listed here are a small subset of the known planets found outside of our solar system. Mass and radius are listed in "Jupiter mass" and "Jupiter radius" units. The orbital period is measured in Earth days. The full dataset is available from the [Open Exoplanet Catalogue](#).

2M 0746+20 b

Mass	Radius	Period
30	0.97	4640

2M 2140+16 b

Mass	Radius	Period
20	0.92	7340

- 4.1. Expose the application's service:

```
[student@workstation exochart]$ oc expose service exoplanets-exochart
route.route.openshift.io/exoplanets-exochart exposed
```

- 4.2. Open the deployed application in a browser:

```
[student@workstation exochart]$ firefox $(oc get route exoplanets-exochart \
-o jsonpath='{.spec.host}' \
-n ${RHT_OCP4_DEV_USER}-multicontainer-review ) &
```

5. Create the base Kustomize directory structure and the deployment files.

Create the Kustomize directory named `exokustom` in the `~/DO288/labs/multicontainer-review` directory.



Note

You can use the `helm template` command to extract the object definitions from a Helm Chart into the base definition:

```
[student@workstation ~]$ helm template app-name helm-directory >
base/deployment.yaml
```

- 5.1. Create the `exokustom` directory to hold all the Kustomize directories.

```
[student@workstation exochart]$ cd ..
[student@workstation multicontainer-review]$ mkdir exokustom
[student@workstation exochart]$ cd exokustom
```

- 5.2. Create the directory for the base definition.

```
[student@workstation exokustom]$ mkdir base
```

- 5.3. Use the `helm template` command to extract the object definitions from the Helm Chart into the base definition:

```
[student@workstation exokustom]$ helm template exoplanets \
..../exochart > base/deployment.yaml
```

- 5.4. Create the Kustomize description file for the base definition.

Create a new `kustomization.yaml` file in the `base` directory to hold the base Kustomize definition. Add the `deployment.yaml` file as a resource in the `kustomization.yaml` file of the `base` directory.

```
resources:
- deployment.yaml
```

6. Create and apply the Test overlay in the Kustomize directory, using a directory called `test`.

- 6.1. Create the test overlay in the Kustomize directory.

```
[student@workstation exokustom]$ mkdir -p overlays/test
```

- 6.2. Adjust the policies of the Test environment.

Create a `replica_limits.yaml` file in the `overlays/test` directory to hold the changes in the deployment definition. These changes must reflect the resource limits defined in the Lab Requirements:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: exoplanets-exochart
spec:
  replicas: 5
  template:
    spec:
      containers:
        - name: exochart
          resources:
            limits:
              memory: "128Mi"
              cpu: "250m"
```

- 6.3. Create the `overlays/test/kustomization.yaml` file, add the `base` directory as the base definition, and the `replica_limits.yaml` file as a patch:

```
bases:  
- ../../base  
patches:  
- replica_limits.yaml
```

- 6.4. Apply the Test overlay to the running instance of the Exoplanets application:

```
[student@workstation exokustom]$ oc apply -k overlays/test  
...output omitted...  
serviceaccount/exoplanets-exochart configured  
service/exoplanets-cockroachdb-public configured  
service/exoplanets-cockroachdb configured  
service/exoplanets-exochart configured  
deployment.apps/exoplanets-exochart configured  
statefulset.apps/exoplanets-cockroachdb configured  
...output omitted...
```

Any warning in this form can be safely ignored:

```
Warning: oc apply should be used on resource created by either oc create --save-config or oc apply
```

7. Test the application in the browser and verify that the custom values have been applied.

- 7.1. Verify that the application still works correctly.

Refresh the browser and you should still see the same application shown previously.

- 7.2. Verify that Kustomize applied the CPU and memory limit change:

```
[student@workstation exokustom]$ oc get deployments exoplanets-exochart \  
-o jsonpath='{.spec.template.spec.containers[0].resources.limits}'  
{"cpu":"250m", "memory":"128Mi"}
```

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 exokustom]$ lab multicontainer-review grade
```

Finish

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

```
[student@workstation exokustom]$ cd  
[student@workstation ~]$ lab multicontainer-review finish
```

This concludes the lab.

Summary

In this chapter, you learned:

- How to convert a set of Red Hat OpenShift resources into templates.
- How to use Helm Charts to package Kubernetes applications.
- To customize Kubernetes resources for different environments using Kustomize.
- How to use Kustomize to customize a packaged Kubernetes application without altering it.

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 startup, readiness and liveness probes to monitor application readiness and health.

Red Hat OpenShift Container Platform Readiness and Liveness Probes

Applications can become unreliable for a variety of reasons, for example:

- Temporary connection loss
- Configuration errors
- Application errors

Developers can use *probes* to monitor their applications. Probes make developers aware of events such as application status, resource usage, and errors.

Monitoring of such events is useful for fixing problems, but can also help with resource planning and managing.

A probe is a periodic check that monitors the health of an application. Developers can configure probes by using either the `oc` command-line client, a YAML deployment template, or by using the Red Hat OpenShift web console.

There are currently three types of probes in Red Hat OpenShift:

Startup Probe

A startup probe verifies whether the application within a container is started. Startup probes run before any other probe, and, unless it finishes successfully, disables other probes. If a container fails its startup probe, the container is killed and will follow the pod's `restartPolicy`.

This type of probe is only executed at startup, unlike readiness probes, which are run periodically.

The startup probe is configured in the `spec.containers.startupProbe` attribute of the pod configuration.

Readiness Probe

Readiness probes determine whether or not a container is ready to serve requests. If the readiness probe returns a failed state, Red Hat OpenShift removes the IP address for the container from the endpoints of all services.

Developers can use readiness probes to signal to Red Hat OpenShift that even though a container is running, it should not receive any traffic from a proxy. This can be useful for waiting for an application to perform network connections, loading files and cache, or generally any initial tasks that might take considerable time and only temporarily affect the application.

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, Red Hat OpenShift kills the container and tries to redeploy it.

The liveness probe is configured in the `spec.containers.livenessprobe` attribute of the pod configuration.

Red Hat OpenShift provides five options that control these 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, Red Hat OpenShift assumes that the probe failed.	1
<code>periodSeconds</code>	No	Specifies the frequency of the checks.	1
<code>successThreshold</code>	No	Specifies the minimum consecutive successes for the probe to be considered successful after it has failed.	1
<code>failureThreshold</code>	No	Specifies the minimum consecutive failures for the probe to be considered failed after it has succeeded.	3

Methods of Checking Application Health

Startup, 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.

HTTP probe uses GET requests to check the health of an application. The check is 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:

```
...contents omitted...
readinessProbe:
  httpGet:
    path: /health❶
    port: 8080
  initialDelaySeconds: 15❷
  timeoutSeconds: 1❸
...contents omitted...
```

- ❶ The readiness probe endpoint.
- ❷ How long to wait after the container starts before checking its health.
- ❸ How long to wait for the probe to finish.

Container Execution Checks

Container execution checks are ideal in scenarios where you must determine the status of the container based on the exit code of a process or shell script running in the container.

When using container execution checks, Red Hat 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:

```
...contents omitted...
livenessProbe:
  exec:
    command:❶
    - cat
    - /tmp/health
  initialDelaySeconds: 15
  timeoutSeconds: 1
...contents omitted...
```

- ❶ The command to run and its arguments, as a 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, Red Hat 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 by using the TCP socket check method:

```
...contents omitted...
livenessProbe:
  tcpSocket:
    port: 8080①
  initialDelaySeconds: 15
  timeoutSeconds: 1
...contents omitted...
```

- ① The TCP port to check.

Manage Probes By Using the Web Console

Developers can create probes by editing the Deployment YAML file using either `oc edit` or the Red Hat OpenShift Web Console.

You can directly edit the Deployment YAML file from the **Workloads** → **Deployment** → <deployment name> page in the web console. Click the **Actions** drop down, and select **Edit Deployment**.

The screenshot shows the 'Deployment Details' page for a deployment named 'example'. The 'Actions' dropdown menu is open, and the 'Edit Deployment' option is highlighted with a red box. Other options in the menu include 'Edit Pod Count', 'Pause Rollouts', 'Add Health Checks', 'Add Horizontal Pod Autoscaler', 'Add Storage', 'Edit Update Strategy', 'Edit Labels', 'Edit Annotations', and 'Delete Deployment'.

Name	Update Strategy
example	RollingUpdate

Namespace	Max Unavailable
NS your-project	25% of 1 pod

Figure 7.1: Adding probes by using the web console

The following example shows the YAML editor in the web console for a deployment.

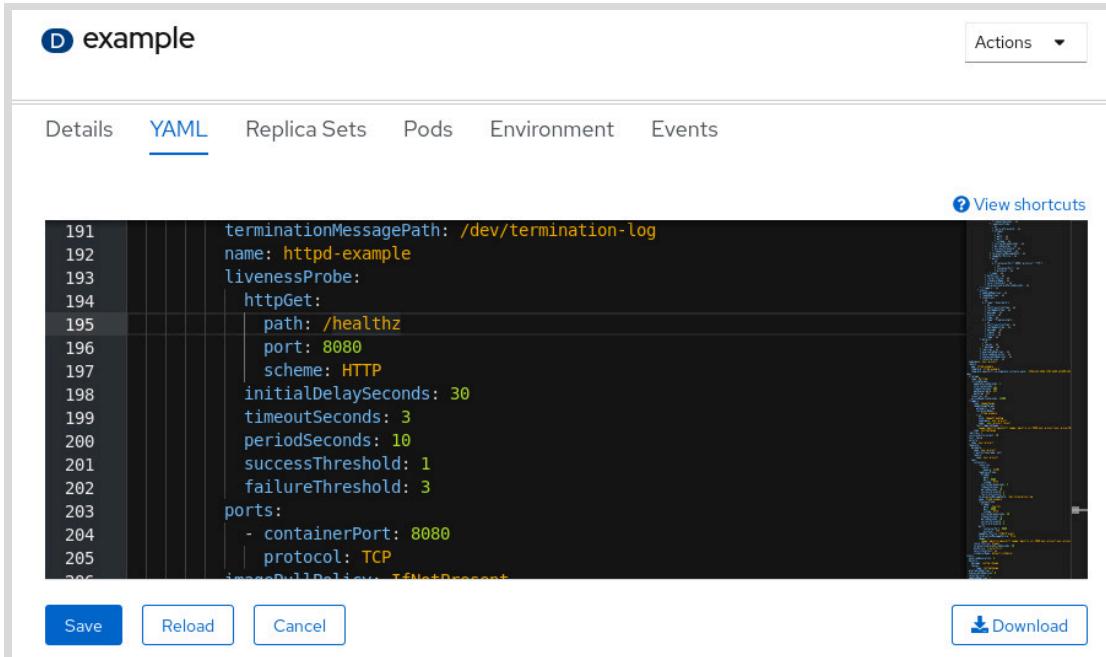


Figure 7.2: Adding probes by using the web console YAML editor

Creating Probes By Using CLI

The `oc set probe` command provides an alternative approach to editing the deployment YAML definition directly. When you are creating probes for running applications, use the command `oc set probe`, which 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, 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 deployment myapp --readiness \
--get-url=http://:8080/healthz --period=20
```

```
[user@host ~]$ oc set probe deployment myapp --liveness \
--open-tcp=3306 --period=20 \
--timeout-seconds=1
```

```
[user@host ~]$ oc set probe deployment 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 by using health checks* section of the *Applications* chapter of the *Official Documentation* for Red Hat OpenShift Container Platform 4.6 at
https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6/html-single/applications/index#application-health

Further information is available in the *Configure Liveness, Readiness and Startup Probes* page of the *Kubernetes* website at
<https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/>

► Guided Exercise

Activating Probes

In this exercise, you will configure liveness and readiness probes to monitor the health of an application deployed to your Red Hat OpenShift cluster.

The application you deploy in this exercise exposes two HTTP GET endpoints:

- The `/healthz` endpoint responds with a 200 HTTP status code when the application pod can receive requests.

The endpoint indicates that the application pod is healthy and reachable. It does not indicate that the application is ready to serve requests.

- The `/ready` endpoint responds with a 200 HTTP status code if the overall application works.

The endpoint indicates that the application is ready to serve requests.

In this exercise, the `/ready` endpoint responds with the 200 HTTP status code when the application pod starts. The `/ready` endpoint responds with the 503 HTTP status code for the first 30 seconds after deployment to simulate slow application startup.

You will configure the `/healthz` endpoint for the liveness probe, and the `/ready` endpoint for the readiness probe.

You will simulate network failures in your Red Hat OpenShift cluster and observe behavior in the following scenarios:

- The application is not available.
- The application is available but cannot reach the database. Consequently, it cannot serve requests.

Outcomes

You should be able to:

- Configure 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 Red Hat OpenShift cluster.
- The Node.js S2I builder image.
- The sample application in the DO288-apps 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
```

Instructions

- 1. Create a new project, and then deploy the sample application in the `probes` subdirectory of the Git repository to the Red Hat OpenShift cluster.

- 1.1. Source your classroom environment configuration:

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

- 1.2. Log in to Red Hat OpenShift with 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 deployment with the following parameters:

- Name: `probes`
- Build image: `nodejs:12`
- Application directory: `probes`
- Build variables:
 - Name: `npm_config_registry`, value: `http://${RHT_OCP4_NEXUS_SERVER}/repository/nodejs`

You can copy or execute the following command by using the `/home/student/D0288/labs/probes/oc-new-app.sh` script:

```
[student@workstation ~]$ oc new-app \
--name probes --context-dir probes --build-env \
npm_config_registry=http://${RHT_OCP4_NEXUS_SERVER}/repository/nodejs \
nodejs:12-https://github.com/${RHT_OCP4_GITHUB_USER}/D0288-apps
--> 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 Red Hat OpenShift registry:

```
[student@workstation ~]$ oc logs -f bc/probes
...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          46s
probes-6cc59f8f98-vxfw9 1/1     Running   0          10s
```

- 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:

```
[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 the /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 configure the liveness and readiness probes with the following parameters:
 - For the liveness probe, use the `/healthz` endpoint on the port `8080`.
 - For the readiness probe, use the `/ready` endpoint on the port `8080`.
 - For both probes:
 - Configure initial delay of `2` seconds.
 - Configure the timeout of `2` seconds.

```
[student@workstation ~]$ oc set probe deployment probes --liveness \
--get-url=http://:8080/healthz \
--initial-delay-seconds=2 --timeout-seconds=2
deployment.apps/probes updated
[student@workstation ~]$ oc set probe deployment probes --readiness \
--get-url=http://:8080/ready \
--initial-delay-seconds=2 --timeout-seconds=2
deployment.apps/probes updated
```

3.2. Verify the value in the `livenessProbe` and `readinessProbe` entries:

```
[student@workstation D0288-apps]$ oc describe deployment probes | \
grep -ia 1 liveness
  Liveness: http-get http://:8080/healthz delay=2s timeout=2s period=10s
#success=1 #failure=3
  Readiness: http-get http://:8080/ready delay=2s timeout=2s period=10s
#success=1 #failure=3
```

3.3. Wait for the application pod to redeploy and change into the `READY` state:

```
[student@workstation D0288-apps]$ oc get pods
NAME           READY   STATUS    RESTARTS   AGE
...output omitted...
probes-6cc59f8f98-fsf8x8  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-6cc59f8f98-fsf8x8  1/1   Running     0          62s
```

- 3.4. Use the `oc logs` command to see the results of the liveness and readiness probes:

```
[student@workstation ~]$ POD=$(oc get pods -o name | grep -v build)
[student@workstation ~]$ oc logs -f $POD
...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 network failure.

In case of a network failure, a service becomes unresponsive. This means both the liveness and readiness probes fail.

Red Hat OpenShift can resolve the issue by recreating the container on a different node.

- 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...
Switched app state to not ready...
```

- 4.2. Return to the terminal where you are monitoring the application deployment:

```
[student@workstation ~]$ oc logs -f $POD
...output omitted...
ping /healthz => pong [healthy]
Received kill request for health probe.
Received kill request for readiness probe.
ping /ready => pong [notready]
ping /healthz => pong [unhealthy]
ping /ready => pong [notready]
ping /healthz => pong [unhealthy]
ping /ready => pong [notready]
ping /healthz => pong [unhealthy]
npm info lifecycle probes@1.0.0~poststart: probes@1.0.0
npm timing npm Completed in 150591ms
npm info ok
```

Red Hat OpenShift restarts the pod when the liveness probe fails repeatedly (three consequent failures by default). This means Red Hat OpenShift restarts the application on an available node not affected by the network failure.

You see this log output only when you immediately check the application logs after you issue the kill request. If you check the logs after Red Hat OpenShift restarts the pod, then the logs are cleared and you only see the output shown in the next step.

- 4.3. Verify that Red Hat 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 D0288-apps]$ oc get pods
NAME           READY   STATUS    RESTARTS   AGE
...output omitted...
probes-6cc59f8f98-fsf8x8  1/1     Running   1          62s
```

Wait until the previous pod terminates before you continue with the next step.

- 4.4. Check the application logs. The liveness probe succeeds and the application reports a healthy state.

```
[student@workstation ~]$ POD=$(oc get pods -o name | grep -v build)
[student@workstation ~]$ oc logs -f $POD
...output omitted...
ping /ready => pong [ready]
ping /healthz => pong [healthy]
...output omitted...
```

▶ 5. Simulate a failure to reach the application database.

In case of a network failure between the application and the database, the application still responds to requests, but cannot serve requests.

This means the liveness probe passes but the readiness probe fails.

- 5.1. Execute the `~/D0288/labs/probes/not-ready.sh` script to simulate a readiness probe failure:

```
[student@workstation ~]$ ~/D0288/labs/probes/not-ready.sh
Switched app state to not ready...
```

- 5.2. Check the application logs. The readiness probe returns failure.

```
[student@workstation ~]$ oc logs -f $POD
...output omitted...
ping /ready => pong [notready]
ping /healthz => pong [healthy]
...output omitted...
```

- 5.3. Check that the application pod is not in the READY state:

```
[student@workstation D0288-apps]$ oc get pods
NAME           READY   STATUS    RESTARTS   AGE
...output omitted...
probes-6cc59f8f98-fsf8x8  0/1     Running   1          78m
```

- 5.4. Check that the service is unavailable:

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
[student@workstation ~]$ curl -is \
probes-${RHT_OCP4_DEV_USER}-probes.${RHT_OCP4_WILDCARD_DOMAIN} \
| grep 'HTTP/1.0'

HTTP/1.0 503 Service Unavailable
```

In a production environment, Red Hat OpenShift would re-route requests to redundant application pods. When the application reaches the database, the readiness probe passes again, the pod switches to the READY state, and Red Hat OpenShift resumes routing traffic to the pod.

► 6. Verify that you can see the failure of the probes in the event log.

Use the `oc describe` command on the pod from the previous step:

```
[student@workstation ~]$ POD=$(oc get pods -o name | grep -v build)
[student@workstation ~]$ oc describe $POD
Events:
  Type      Reason     Age   From      Message
  ----      ----     --   --       --
...output omitted...
  Warning  Unhealthy  ...   ...     Liveness probe failed: ... statuscode: 503
  Normal   Killing    ...   ...     Container probes failed liveness probe, will be
  restarted
...output omitted...
  Warning  Unhealthy  ...   ...     Readiness probe failed: ... statuscode: 503
```

► 7. Clean up. Delete the `probes` project in Red Hat 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.

Choosing DeploymentConfig over Deployments

By default, most Red Hat OpenShift Container Platform commands, such as `oc new-app create Deployment` resources, which are first-class native API resources found in every Kubernetes distribution. However, Red Hat OpenShift provides a resource called `DeploymentConfig`.

The `DeploymentConfig` resource enables you to use additional features, such as

- Custom deployment strategies
- Lifecycle hooks

The `Deployment` resource does not support custom strategies.

The `Deployment` resource enables you to define container lifecycle hooks, such as `PostStart` and `PreStop`, which are similar to the lifecycle hooks provided by the `DeploymentConfig` resource. However, the `Deployment` container lifecycle hooks are not guaranteed to be executed before the `ENTRYPOINT` command of the container pod.

Consequently, this limits the use case of the `Deployment` container lifecycle hooks.

Use the `Deployment` resource if you do not require any additional features provided by the `DeploymentConfig` resource, or if your deployments must be compatible with other distributions of Kubernetes.

Deployment Strategies in Red Hat 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.

Red Hat OpenShift provides several deployment strategies. These strategies can be organized into two primary categories:

- By using the deployment strategy defined in the application deployment configuration.
- By using the Red Hat OpenShift router to route traffic to specific application pods.

Strategies defined within 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.

This strategy progressively replaces instances of the previous version of an application with instances of the new version of the application. This strategy executes the readiness probe

to determine when new pod is ready. After a readiness probe for the new pod succeeds, the deployment controller scales down the old pod.

If a significant issue occurs, the deployment controller aborts the rolling deployment. Developers can also manually abort the rolling deployment by using the `oc rollout cancel` command.

Rolling deployments in Red Hat OpenShift are canary deployments; Red Hat OpenShift tests a new version (the canary) before replacing all of the old instances. If the readiness probe never succeeds, Red Hat OpenShift removes the canary instance and automatically rolls back the deployment configuration.

Use a rolling deployment strategy when:

- You require no downtime during an application update.
- Your application supports running an older version and a newer version at the same time.

Figure 7.3: Rolling Update deployment strategy

Recreate

In this strategy, Red Hat 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 specify the strategy in a `DeploymentConfig` resource in the `spec.strategy.type` attribute.

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.image` attribute of the application deployment configuration. You can also customize environment variables and the command to execute for the deployment.

Integrating Red Hat 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. Red Hat OpenShift deployments contain three life-cycle hooks:

Pre-Lifecycle Hook

Red Hat 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.

These lifecycle hooks are defined on the `Strategy` resource under one of three attributes:

- `rollingParams` for Rolling strategies.
- `recreateParams` for Recreate strategies.
- `customParams` for Custom strategies.

You can add a `pre`, `mid`, and `post` attribute, which contains the lifecycle hooks.

Life-cycle hooks run in separate containers, which are short-lived. Red Hat 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 Red Hat OpenShift Router

Advanced Deployment strategies that use the Red Hat 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 Red Hat 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 Red Hat 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.

Additionally, consider 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

Red Hat OpenShift allows application instances to shut down gracefully before removing the instances from the router's load-balancing roster. Applications must ensure they gracefully terminate user connections before they exit.

Red Hat OpenShift sends a SIGTERM 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, Red Hat OpenShift sends a SIGKILL 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 4.6 at https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6/html-single/applications/index#deployments

A detailed explanation of differences between Deployment and Deployment Config resources can be found at

<https://docs.openshift.com/container-platform/4.6/applications/deployments/what-deployments-are.html>

An introduction to Blue-Green, canary, and rolling deployments

<https://opensource.com/article/17/5/colorful-deployments>

Application Release Strategies with OpenShift

<https://access.redhat.com/articles/2897391>

A description of lifecycle hooks available for Deployment resources can be found at **Container Lifecycle Hooks**

<https://kubernetes.io/docs/concepts/containers/container-lifecycle-hooks/>

► Guided Exercise

Implementing a Deployment Strategy

In this exercise, you will initialize a database with a deployment life-cycle hook.

Outcomes

You should be able to:

- Change the deployment strategy of a MySQL database `DeploymentConfig` resource to `Recreate`.
- Add a post-deployment life-cycle hook to the `DeploymentConfig` resource 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 Red Hat OpenShift cluster.
- The MySQL 8.0 container image (`rhel8/mysql-80`).

Run the following command on `workstation` to validate the exercise prerequisites, and to download the lab and solution files:

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

Instructions

- 1. Create a new project, and deploy an application based on the `rhel8/mysql-80` container image to the Red Hat OpenShift cluster.

- 1.1. Source the configuration of your classroom environment:

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

- 1.2. Log in to Red Hat OpenShift by 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 with the following parameters:
- Name: mysql
 - Variables:
 - Name: MYSQL_USER, value: test
 - Name: MYSQL_PASSWORD, value: redhat
 - Name: MYSQL_DATABASE, value: testdb
 - Name: MYSQL_AI0, value: 0
 - Container image: `registry.redhat.io/rhel8/mysql-80`
 - Use the `deploymentconfig` resource

Copy or execute the following command from the `~/D0288/labs/strategy/oc-new-app.sh` script:

```
[student@workstation ~]$ oc new-app --as-deployment-config \
--name mysql -e MYSQL_USER=test -e MYSQL_PASSWORD=redhat \
-e MYSQL_DATABASE=testdb -e MYSQL_AI0=0 \
--docker-image registry.redhat.io/rhel8/mysql-80
--> Found container image ... "registry.redhat.io/rhel8/mysql-80"
...output omitted...
--> Creating resources ...
...output omitted...
--> Success
...output omitted...
```

- 1.5. Wait until the MySQL pod is deployed. The pod should be in the READY state:

```
[student@workstation ~]$ oc get pods
NAME          READY   STATUS    RESTARTS   AGE
mysql-1-54bhp 1/1     Running   0          11s
mysql-1-deploy 0/1     Completed  0          16s
```

► 2. Change the deployment strategy to Recreate.

- 2.1. Verify that the default deployment strategy for the MySQL application is Rolling:

```
[student@workstation ~]$ oc describe dc/mysql | grep -i strategy:
Strategy: 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 ~/D0288/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. Review the ~/D0288/labs/strategy/users.sql file.

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. Review the ~/D0288/labs/strategy/import.sh script.

The script downloads and runs the previous SQL script to initialize the database. The post life-cycle hook downloads and executes the `import.sh` script:

```
#!/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 implements retries because the hook pod starts at the same time as the database pod. Consequently, the database pod must become ready before the pod hook can execute the SQL script.

- 3.3. Review the ~/D0288/labs/strategy/post-hook.sh script. The script adds a new post life-cycle hook to the DeploymentConfig resource:

```
[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. The hook downloads and executes the files during startup.

- 3.4. Execute the ~/D0288/labs/strategy/post-hook.sh script:

```
[student@workstation ~]$ ~/DO288/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 -iA 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 reaches the Running state. Then, the `mysql-2-deploy` and `mysql-2-hook-post` pods reach the Running state:

```
[student@workstation ~]$ watch -n 2 oc get pods
NAME          READY   STATUS    RESTARTS   ...   NODE
mysql-2-deploy 1/1     Running   0          ... 
mysql-2-hook-post 1/1     Running   0          ... 
mysql-2-kbnpr   1/1     Running   0          ...
```

After a few seconds, the post life-cycle hook pod and the second deployment pod fail:

NAME	READY	STATUS	RESTARTS	AGE
mysql-1-deploy	0/1	Completed	0	13m
mysql-1-vnq68	1/1	Running	0	114s
mysql-2-deploy	0/1	Error	0	2m11s
mysql-2-hook-post	0/1	Error	0	119s

When the hook and the deployment pods reach the error state, press `Ctrl+C` to exit the `watch` command.

The `mysql-1-vnq68` 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 by 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, with 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**:

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 Red Hat 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 with the latest changes to the deployment configuration:

NAME	READY	STATUS	RESTARTS	AGE
mysql-1-deploy	0/1	Completed	0	4m29s
mysql-2-deploy	0/1	Error	0	2m2s
mysql-2-hook-post	0/1	Error	0	113s
mysql-3-deploy	0/1	Completed	0	62s
mysql-3-29jwt	1/1	Running	0	49s
mysql-3-hook-post	0/1	Completed	0	45s

After the mysql-3-hook-post pod runs and exits, press **Ctrl+C** to exit the `watch` command.

Notice that Red Hat 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 -e "select * from users;"
...output omitted...
+-----+-----+-----+
| user_id | name   | email        |
+-----+-----+-----+
|       1 | user1 | user1@example.com |
|       2 | user2 | user2@example.com |
|       3 | user3 | user3@example.com |
+-----+-----+-----+
```

- 6.3. Exit the MySQL session and the container shell:

```
sh-4.2$ exit
exit
```

► 7. Clean up.

Delete the `strategy` project:

```
[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.

Red Hat OpenShift automatically creates a replication controller that represents the deployment configuration pod template when you create a deployment configuration.

When the deployment configuration changes, Red Hat 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. Red Hat 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 Red Hat 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 By 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 latest dc/name` command.

```
[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 dc/name` 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 configuration that failed, run the `oc rollout` command with the `retry` option:

```
[user@host ~]$ oc rollout retry dc/name
```

You may retry a deployment configuration 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 configuration, it restarts the deployment process but does not create a new deployment revision. Red Hat 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 configuration, 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 by 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.

This feature is not available at the moment for Deployment resources, where you're expected to redeploy builds by setting tags. When working with Deployment resources, you can find hashes from builds by using `oc describe imagestream name`, and then proceed to set hash as a tag with the command `oc tag imagename:latest hash-from-older-build`.

- 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 by 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 Red Hat 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 by 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 4.6 at https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6/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 Red Hat OpenShift cluster.

Outcomes

You should be able to:

- Deploy a Thorntail-based application to an Red Hat 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 Red Hat 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
```

Instructions

► 1. Review the application source code.

- 1.1. Enter your local clone of the `D0288-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 D0288-apps
[student@workstation D0288-apps]$ git checkout main
...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 ~/D0288-apps/quip/src/main/java/com/redhat/training/example/Quip.java file.

The quip application is a 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 / endpoint return a quote.
- ② Requests to the /ready endpoint return an OK message.

► 2. Create an application based on the source code.

- 2.1. Source your classroom environment configuration:

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

- 2.2. Log in to Red Hat OpenShift by 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 with the following parameters:

- Name: quip
- Build environment variables:
 - Name: MAVEN_MIRROR_URL, value: http://\${RHT_OCP4_NEXUS_SERVER}/repository/java
- Image stream: redhat-openjdk18-openshift:1.5
- Directory of the application: /quip

You can execute the /home/student/D0288/labs/app-deploy/oc-new-app.sh script, or execute the following command:

```
[student@workstation ~]$ oc new-app --as-deployment-config --name quip \
--build-env MAVEN_MIRROR_URL=http://${RHT_OCP4_NEXUS_SERVER}/repository/java \
-i redhat-openjdk18-openshift:1.5 --context-dir quip \
https://github.com/${RHT_OCP4_GITHUB_USER}/D0288-apps#app-deploy
--> 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 Red Hat 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 the READY state:

```
[student@workstation ~]$ oc get pods
NAME        READY   STATUS    RESTARTS   AGE
quip-1-59j8q   1/1     Running   0          10s
quip-1-build   0/1     Completed  0          89s
quip-1-deploy  0/1     Completed  0          12s
```

► 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 get pods
NAME        READY   STATUS    RESTARTS   AGE
quip-1-59j8q   1/1     Running   0          50s
...output omitted...
[student@workstation ~]$ oc logs quip-1-59j8q
...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 Service resource for the application has a registered endpoint to route incoming requests:

```
[student@workstation ~]$ oc describe svc/quip
Name:           quip
...output omitted...
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. Test the application by 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 `DeploymentConfig` resource with the following parameters for both probes:

- Endpoint: /ready
- Port: 8080

- Initial delay: 30 seconds
- Timeout: 2 seconds

```
[student@workstation ~]$ oc set probe dc/quip \
--liveness --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 | grep http-get
Liveness: http-get http://:8080/ready delay=30s timeout=2s period=10s
#success=1 #failure=3
Readiness: http-get http://:8080/ready delay=30s timeout=2s period=10s
#success=1 #failure=3
```

4.3. Wait for the application pod to redeploy and appear in the READY state:

```
[student@workstation ~]$ oc get pods
...output omitted...
quip-2-n6nzw 1/1 Running 0 26s
```

4.4. Use the oc describe command to inspect the running pod and ensure the probes are active:

```
[student@workstation ~]$ oc describe pod quip-2-n6nzw | 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 by 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 changes to the application source and redeploy the application.

Verify that you can see the changes when you test the application.

5.1. Review the ~/D0288/labs/app-deploy/app-change.sh script.

The script changes the source code to print the message in English, commits, and then pushes the change to the Git repository.

```
[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. Execute 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 the READY state:

```
[student@workstation ~]$ oc get pods
NAME        READY   STATUS    RESTARTS   AGE
quip-1-build  0/1     Completed  0          12m
quip-1-deploy  0/1     Completed  0          11m
quip-2-build  0/1     Completed  0          2m11s
quip-2-deploy  0/1     Completed  0          4m59s
quip-3-deploy  0/1     Completed  0          60s
quip-3-gvzs5   1/1     Running   0          56s
```

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 previous *Veni, vidi, vici...* message.

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 #4 rolled back to quip-2
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 the READY state:

```
[student@workstation ~]$ oc get pods
NAME          READY   STATUS    RESTARTS   AGE
quip-1-build  0/1     Completed  0          17m
quip-1-deploy  0/1     Completed  0          16m
quip-2-build  0/1     Completed  0          7m1s
quip-2-deploy  0/1     Completed  0          9m49s
quip-3-deploy  0/1     Completed  0          5m50s
quip-4-9h6ql  1/1     Running   0          89s
quip-4-deploy  0/1     Completed  0          95s
```

- 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 project:

```
[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

In this lab, you will manage the deployment of an application, and scale it on an Red Hat OpenShift Container Platform (RHOC) cluster.

Outcomes

You should be able to:

- Deploy a PHP-based application to an Red Hat 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 Red Hat OpenShift cluster.
- The `php-scale` application source code in the DO288-apps Git repository fork.

Run the following command on `workstation` to validate the prerequisites:

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

Requirements

This lab uses a PHP-based application that prints the following information:

- The version of the application
- The name of the application pod
- The IP address of the application pod

Deploy and test the application on an Red Hat OpenShift cluster according to the following instructions:

- Use the `php:7.3` image stream to deploy the application.
- Ensure that the application name for Red Hat 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.
- Ensure that the application uses the `DeploymentConfig` resource.

Instructions

1. 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 main  
...output omitted...
```

2. 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.
```

3. Log in to the Red Hat OpenShift cluster by using your personal development user name. Create a new project named `youruser-manage-deploy`. Deploy the application in the `php-scale` directory by using the `php:7.3` image stream. Name the application `scale` and use the `DeploymentConfig` resource to manage the application. Ensure that you reference the `manage-deploy` branch you created in the previous step when you deploy the application. Expose and test the application by using the generated route URL. Verify that you can see `version 1` and the pod name in the output.
4. Verify that the `Rolling` strategy is the default deployment strategy.
5. 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.
6. Change the version number in the `index.php` file to 2. Commit and push the changes to the remote Git repository. Do not make any other changes to the source code.
7. Start a new build of the application. Verify that Red Hat 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 by using the `curl` command. Verify that `version 2` is in the output.
8. Roll back to the previous deployment.

Use the `curl` command to retest the application. Verify that `version 1` is seen in the output.

9. Grade your work:

```
[student@workstation DO288-apps]$ lab manage-deploy grade
```

10. Clean up and delete the project.

```
[student@workstation DO288-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.

► Solution

Managing Application Deployments

In this lab, you will manage the deployment of an application, and scale it on an Red Hat OpenShift Container Platform (RHOC) cluster.

Outcomes

You should be able to:

- Deploy a PHP-based application to an Red Hat 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 Red Hat OpenShift cluster.
- The `php-scale` application source code in the `D0288-apps` Git repository fork.

Run the following command on `workstation` to validate the prerequisites:

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

Requirements

This lab uses a PHP-based application that prints the following information:

- The version of the application
- The name of the application pod
- The IP address of the application pod

Deploy and test the application on an Red Hat OpenShift cluster according to the following instructions:

- Use the `php:7.3` image stream to deploy the application.
- Ensure that the application name for Red Hat 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.
- Ensure that the application uses the `DeploymentConfig` resource.

Instructions

1. 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 main  
...output omitted...
```

2. 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.
```

3. Log in to the Red Hat OpenShift cluster by using your personal development user name.

Create a new project named `youruser-manage-deploy`.

Deploy the application in the `php-scale` directory by using the `php:7.3` image stream. Name the application `scale` and use the `DeploymentConfig` resource to manage the application.

Ensure that you reference the `manage-deploy` branch you created in the previous step when you deploy the application.

Expose and test the application by using the generated route URL. Verify that you can see `version 1` and the pod name in the output.

3.1. Source your classroom environment configuration:

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

3.2. Log in to Red Hat 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 "yourname-manage-deploy" on server "https://  
api.cluster.domain.example.com:6443".  
...output omitted...
```

3.3. 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 ...
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 Red Hat OpenShift registry:

```
[student@workstation D0288-apps]$ oc logs -f bc/scale
...output omitted...
Push successful
```

3.5. Wait until the application is deployed. The application pod should be in the READY 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. 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-w48nd (10.128.1.21)
```

4. Verify that the Rolling strategy is the default deployment strategy.

```
[student@workstation D0288-apps]$ oc describe dc/scale | grep -i strategy
Strategy: Rolling
```

5. 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 the Red Hat OpenShift web console URL in a web browser:

```
[student@workstation D0288-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.

Verify your credentials by using the /usr/local/etc/ocp4.config file:

- The user name is the value of the RHT_OCP4_DEV_USER variable.
- The password is the value of the RHT_OCP4_DEV_PASSWORD variable.

- 5.3. If you are not in the administrator view, click **Developer** → **Administrator** to switch to the administrator view.

Click **youruser-manage-deploy** on the **Projects** page to open the **Overview** page for the project.

Click the **Workloads** tab to show the deployments available for the project.

The screenshot shows the OpenShift Workloads tab interface. At the top, there are tabs: Overview, Details, YAML, Workloads (which is highlighted with a red box), and Role Bindings. Below the tabs are three input fields: Display Options, Filter by Resource, and Find by name... with a magnifying glass icon. The main content area is titled 'Deployment Config' and contains a single entry: 'DC scale'. The 'scale' entry is preceded by a blue circle with the letters 'DC'.

- 5.4. Select the **scale** deployment configuration entry.

Click the **Details** section. Then, click the upper arrow on the right of the blue circle to increase the number of pods to two.

Watch as Red Hat 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 Red Hat OpenShift router enables session stickiness by default. Therefore you cannot verify the load-balancing behavior. Test the application by using the `curl` command.

6. Change the version number in the `index.php` file to 2. Commit and push the changes to the remote Git repository.
Do not make any other changes to the source code.
 - 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 D0288-apps]$ git commit -a -m "Updated app to version 2"  
[manage-deploy 3633f74] updating version  
 1 file changed, 1 insertion(+), 1 deletion(-)  
[student@workstation D0288-apps]$ git push  
...output omitted...
```

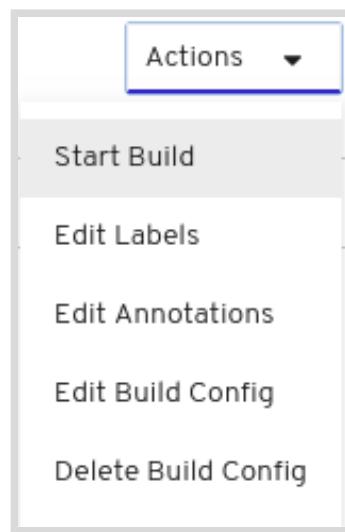
7. Start a new build of the application.

Verify that Red Hat 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 by using the `curl` command. Verify that `version 2` is in the output.

- 7.1. In the left menu of the Red Hat OpenShift web console, click **Builds** → **Build Configs** entry.

Select the **scale** build configuration, and click **Actions** → **Start Build**.



- 7.2. The console redirects you to the summary page about the new build. Click the **Logs** tab to watch the build log.

- 7.3. After the build finishes, Red Hat OpenShift scales up two new pods of the new version of the application.

When the new pods are ready to receive traffic, Red Hat OpenShift scales down the two older pods.

Click **Workloads** → **Pods** to see the new application pods.

- 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.



Note

If you do not see the new version, verify that:

- You pushed the changes to the remote Git repository.
- The new build finished successfully.

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 the READY state:

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-bcxpath	1/1	Running	0	58s
scale-3-deploy	0/1	Completed	0	77s
scale-3-lnp46	1/1	Running	0	68s

8.3. Use the `curl` command to verify the application response:

```
[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:

```
[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

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)
- Containerizing Services (and Guided Exercise)
- Deploy a cloud-native application using Eclipse JKube (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 by 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` project, 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, then 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 4.6 at
https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6/html-single/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.

Instructions

► 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. Verify that the application is ready and running:

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 the prefix `apps.` with `mysql.ocp-`.

```
[student@workstation ~]$ dbhost=$(echo \
mysql.ocp-${RHT_OCP4_WILDCARD_DOMAIN#"apps."})
[student@workstation ~]$ echo ${dbhost}
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 -h${dbhost} \
-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 ${dbhost}
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.ocp-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 a 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 might 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 might fail for one of several possible reasons. Resolving these issues might require customization of the application's deployment 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 might expect to run as the `root` or another fixed user. It might 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 happens with legacy, monolithic applications, that were not designed for a microservice-based architecture. OpenShift cluster administrators set up default limit ranges that provide default values for resource requests and limits to your project, and these might be too small for your legacy application.

You can specify increased resource requests and resource limits on your deployment 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 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.

- 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 forbidden by your OpenShift cluster administrators. These applications might 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 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 might require customization to work under OpenShift. Deployment resources that work with upstream Kubernetes (and also with other vendor distributions of Kubernetes) might 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 might 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." \
      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 process 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 in 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/..."2

ADD solo.json.erb /var/chef/solo.json.erb2

# 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 \3
```

```
&& 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.

- ④ 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 can result in scheduling and stability issues.
- ⑤ 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 --docker-image`, it might not work reliably. To ensure the container image works well, you must make a few customizations to its deployment.

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 the minimum amount of resources the application requires to run. The OpenShift scheduler finds a worker node in the cluster with sufficient available CPU and memory to run the application, which prevents resource starvation failures at start up.

Resource Limits state how much the CPU and memory usage of an application can 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:

```
...output omitted...
- apiVersion: apps/v1
  kind: Deployment
...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: apps/v1
  kind: Deployment
...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) \ ❶
  http://localhost:8081/service/metrics/ping | grep pong ❷
```

- ❶ The `-u` option passes a username and password in the HTTP GET request.
- ❷ 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: apps/v1
  kind: Deployment
...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

Containerizing Nexus as a Service

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 helm chart uses an image stream that uses a container image from a private registry.
- Verify that a helm chart defines resource requests and resource limits for a Java application, and overrides environment variables that would set heap sizes.
- Configure a helm chart to use a persistent volume claim.
- Configure a helm chart to use liveness and readiness probes.
- Deploy an instance of Nexus by using the Helm chart.

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 building the Nexus container 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
```

Instructions

► 1. Review and build the Nexus Dockerfile file.

- 1.1. Enter the `nexus3` subdirectory of your local clone of the `D0288-apps` Git repository, and then checkout the `main` branch of the course repository to ensure that you start this exercise from a known good state:

```
[student@workstation ~]$ cd ~/D0288-apps/nexus3
[student@workstation nexus3]$ git checkout main
...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.

- 1.3. 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="" -Xms2703m -Xmx2703m -XX:MaxDirectMemorySize=2703m` -
Djava.util.prefs.userRoot=${NEXUS_DATA}/javaprefs"
```

- 1.4. 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
```

- 1.5. 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 the 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]$ podman build -t nexus3 .
...output omitted...
STEP 33: COMMIT nexus3
[student@workstation nexus3]$ podman login -u ${RHT_OCP4_QUAY_USER} quay.io
Password:
Login Succeeded!
[student@workstation nexus3]$ 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, then press **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]$ skopeo copy \
docker://quay.io/redhattraining/nexus3:latest \
docker://quay.io/${RHT_OCP4_QUAY_USER}/nexus3
```

- ▶ 2. Complete the helm chart to deploy Nexus as a service.

A partial chart is provided to deploy Nexus. Update it to deploy Nexus as a service.

- 2.1. Create a copy of the starter chart to edit, and exit the `~/D0288-apps/nexus3` folder.

```
[student@workstation nexus3]$ cp -r ~/D0288/labs/nexus-service/nexus-chart ~/
```

```
[student@workstation nexus3]$ cd ~
```

- 2.2. Inspect the `~/nexus-chart/templates/imagestream.yaml` file to verify that it uses the container image set from a configuration value.

```
[student@workstation ~]$ grep -A1 "kind: DockerImage" ~/nexus-chart/templates/
imagestream.yaml
  kind: DockerImage
    name: {{ .Values.imageName }}
```

- 2.3. Open the `~/nexus-chart/values.yaml` file with a text editor and update it to set the `imageName` value to point to the container image you built. Keep this file open in the editor.

```
...output omitted...
imageName: quay.io/youruser/nexus3:latest
...output omitted...
```

- 2.4. Inspect the `~/nexus-chart/templates/deployment.yaml` file to verify that it defines resource requests and resource limits for the application pod:

```
[student@workstation ~]$ grep -B1 -A5 limits: ~/nexus-chart/templates/
deployment.yaml
  resources:
    limits:
      cpu: "1"
      memory: {{ .Values.memoryLimit }}
    requests:
      cpu: 500m
      memory: 256Mi
```

- 2.5. Open the `~/nexus-chart/templates/deployment.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. Keep the file open in the editor.

```
apiVersion: v1
kind: Deployment
...output omitted...
- env:
  - name: INSTALL4J_ADD_VM_PARAMS
    value: -Djava.util.prefs.userRoot=/nexus-data/javaprefs
...output omitted...
```

- 2.6. Update the `~/nexus-chart/values.yaml` file using your text editor, to set the `memoryLimit` value to `2703Mi`
- 2.7. 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 resource. Go to the `~/nexus-chart/templates/deployment.yaml` file in your 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.

```
apiVersion: v1
kind: Deployment
...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...
```

- 2.8. 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 resource. In the `~/nexus-chart/templates/deployment.yaml` file, configure the probe to start 120 seconds after the container starts. Also, configure the probe for a maximum waiting execution time of 30 seconds.

```

apiVersion: v1
kind: Deployment
...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...

```

- 2.9. In the deployment resource, configure the volume mount to use the **nexus-data** volume at a mount path of **/nexus-data**:

```

apiVersion: v1
kind: Deployment
...output omitted...
  volumeMounts:
    - mountPath: /nexus-data
      name: nexus-data
...output omitted...

```

- 2.10. In the deployment resource, name the volume **nexus-data**, and then configure the volume to use the **nexus-data-pvc** persistent volume claim:

```

apiVersion: v1
kind: Deployment
...output omitted...
  volumes:
    - name: nexus-data
      persistentVolumeClaim:
        claimName: nexus-data-pvc
...output omitted...

```

- 2.11. A persistent volume claim is required to persist Nexus data. Open the `~/nexus-chart/templates/pvc.yaml` file in your text editor. In the persistent volume claim resource, name the persistent volume claim **nexus-data-pvc** by updating the `metadata.name` attribute:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  labels:
    app: {{ .Values.nexusServiceName }}
  name: nexus-data-pvc
...output omitted...
```

- 2.12. The helm chart is now complete. Save your edits.

To verify the changes made during this step, compare your helm chart files to the solution files at `~/DO288/solutions/nexus-service/nexus-chart/`. 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
```

- 3.6. Set the correct hostname in `~/nexus-chart/values.yaml`. Copy the output of the following command, and paste it into the values file to set the `hostname` value.

```
[student@workstation ~]$ echo "nexus3-${RHT_OCP4_DEV_USER}.\n${RHT_OCP4_WILDCARD_DOMAIN}"
```

▶ 4. Create a new application.

- 4.1. Create a new application named nexus3 from the helm chart.

You can either copy the complete command, or execute it directly from /home/student/D0288/labs/nexus-service/oc-new-app.sh.

```
[student@workstation ~]$ helm install nexus3 ~/nexus-chart
NAME: nexus3
LAST DEPLOYED: Mon May 31 12:05:20 2021
NAMESPACE: youruser-nexus-service
STATUS: deployed
REVISION: 1
NOTES:
...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-kfwwh  0/1     Running   0          1m25s
[student@workstation ~]$ oc logs -f nexus3-1-kfwwh
...output omitted...
-----
Started Sonatype Nexus OSS 3.30.1-01
-----
...output omitted...
```

Press **Ctrl+C** to exit the `oc logs` command.

- 4.3. Wait for the application to be ready and running. OpenShift might take a few seconds to get a healthy state from the application health probes.

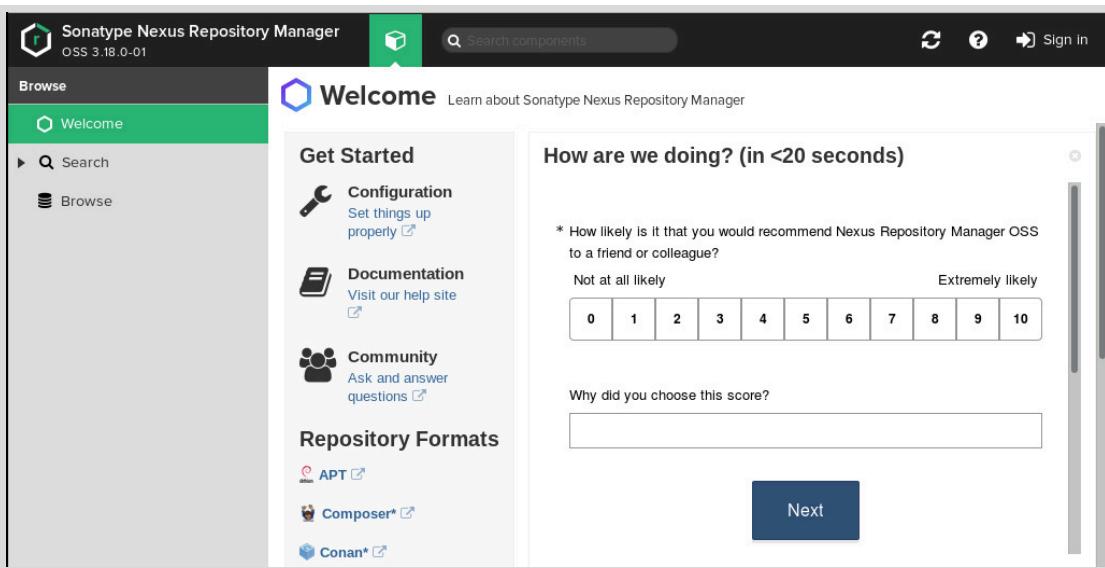
```
[student@workstation ~]$ oc get pods
NAME          READY   STATUS    RESTARTS   AGE
nexus3-a1b2c3d4e5f6-kfwwh  1/1     Running   0          4m25s
```

▶ 5. Test the Nexus application.

- 5.1. Retrieve the route for the Nexus application:

```
[student@workstation ~]$ oc get route
NAME      HOST/PORT
nexus3   nexus3-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, then 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:

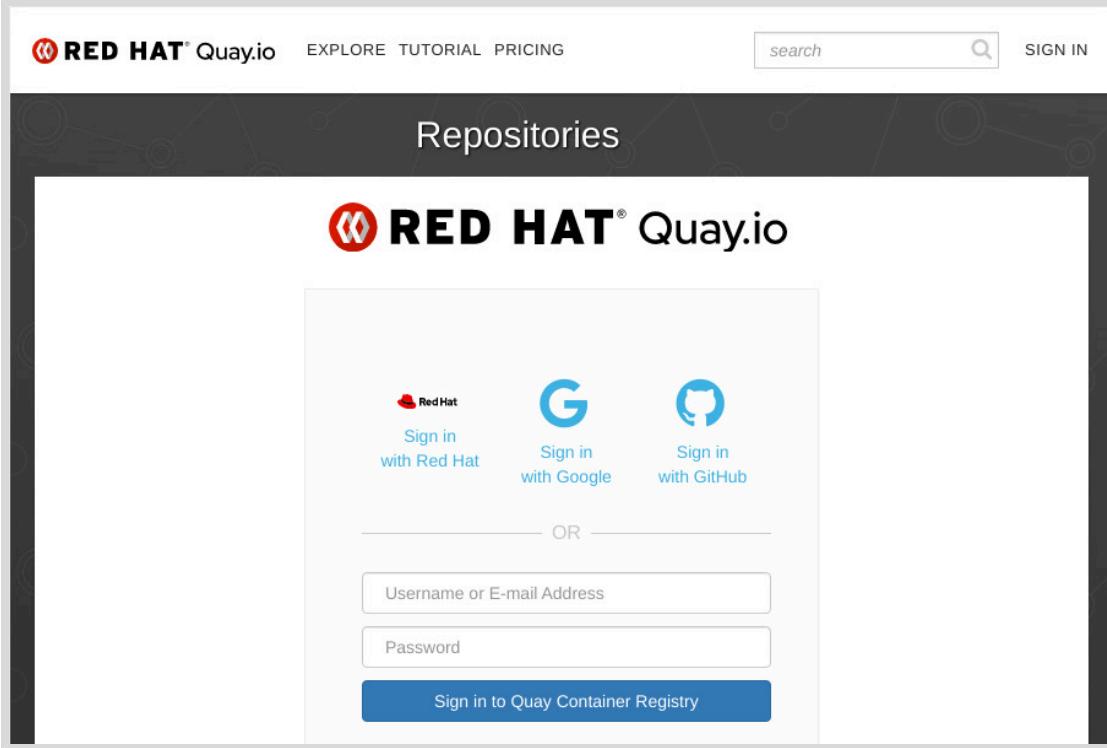
```
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-nexus-service
project.project.openshift.io "youruser.nexus-service" deleted
```

6.2. Delete the container image from the external registry:

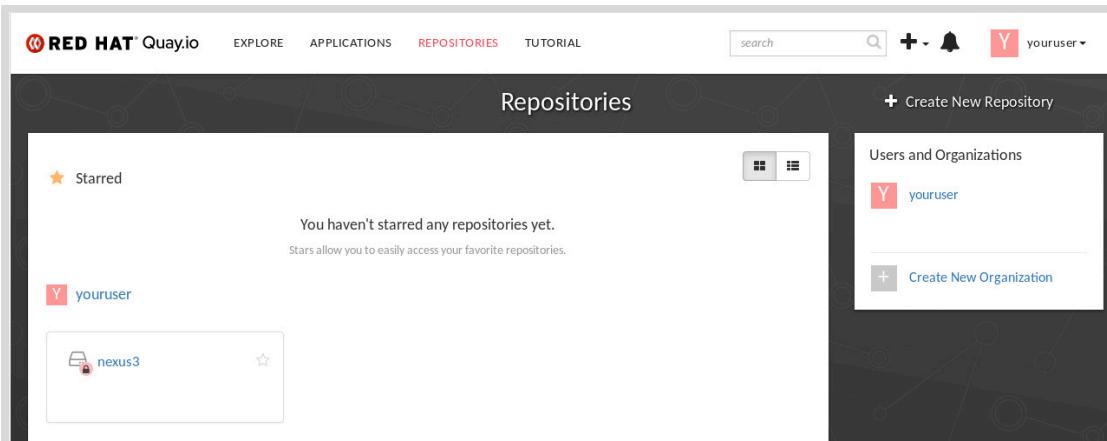
```
[student@workstation ~]$ skopeo delete \
docker://quay.io/${RHT_OCP4_QUAY_USER}/nexus3
```

6.3. Log in to Quay.io using your personal free account.

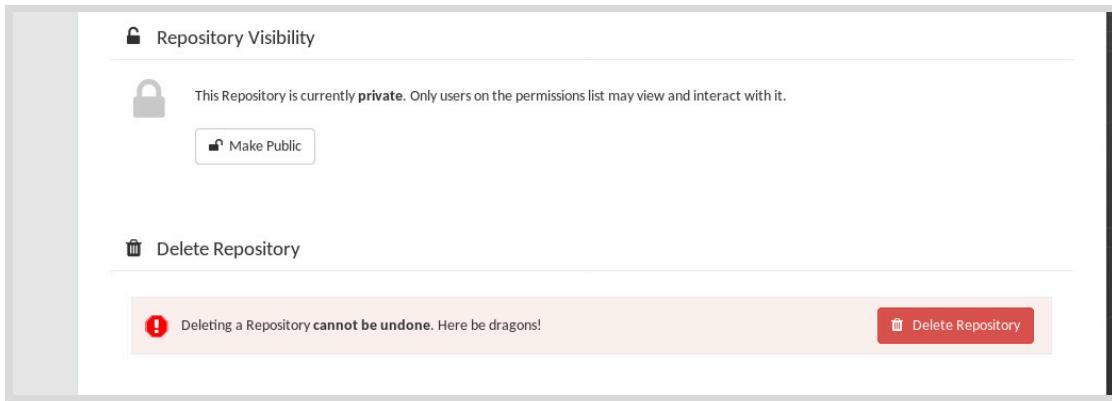
Navigate to <https://quay.io>, and then click **Sign In** to provide your user credentials. Click **Sign in to Quay Container Registry** to log in to Quay.io.



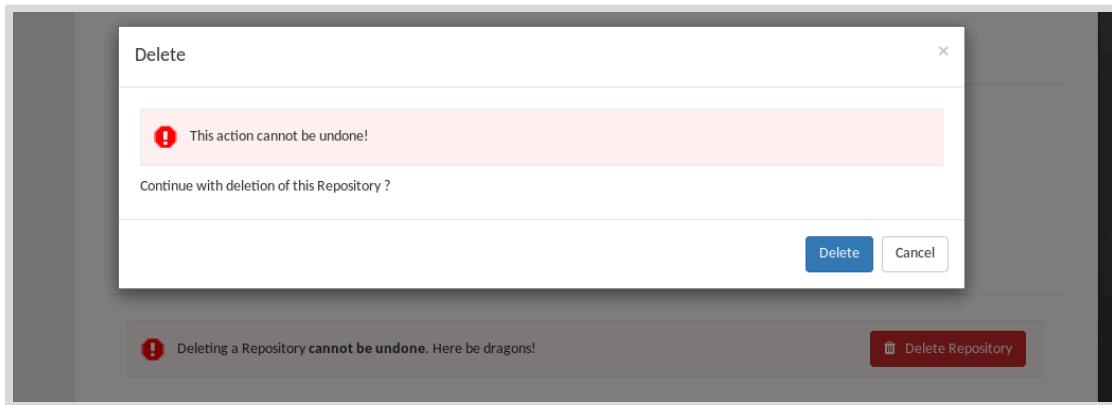
- 6.4. On the Quay.io main menu, click **Repositories** and look for **nexus**. The lock icon indicates that it is a private repository that requires authentication for both pulls and pushes. Click **nexus3** to display the **Repository Activity** page.



- 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 Cloud-Native Applications with JKube

Objectives

After completing this section, students should be able to deploy a cloud-native application using Eclipse JKube.

Cloud-native Applications

Cloud-native is a very general term that aims to describe technologies. These technologies are designed to build and run scalable applications in public, private, and hybrid clouds, where containers, microservices, Kubernetes, and other modern technologies serve as examples.

Technologies that are cloud-native enable resiliency, maintainability, observability, and their processes are automated to allow for frequent updates and deployments. For example, a developer that uses cloud-native tools, such as Quarkus or JKube, does not need to manually create Dockerfiles for the creation of their container images.

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.

Eclipse JKube

Eclipse JKube is a set of open source plugins and libraries that can build container images via different strategies, and generates and deploys Java applications to Kubernetes and OpenShift, with little to no configuration, making your applications cloud-native.

JKube is the result of the refactoring and rebranding of Fabric8, which resulted in the decoupling of the Fabric8 ecosystem, and is now a more general-purpose plug-in for applications targeting Kubernetes and OpenShift.

JKube is composed of the **JKube Kit**, which is the core set of tools for building images and generating deployment manifests, the Kubernetes Maven plugin, and the OpenShift Maven plugin. These plugins are used for deploying to their corresponding cluster type.

JKube, through its Maven plugins, supports several Java frameworks, such as:

Quarkus

A modern full-stack, cloud-native framework, with a small memory footprint and reduced boot time.

Spring Boot

A cloud-native development framework based on the popular Spring Framework and auto configuration.

Vert.x

A reactive, low-latency development framework based on asynchronous I/O and event streams.

Micronaut

A modern full-stack toolkit for building modular, easily testable microservices and serverless applications.

Open Liberty

A flexible server runtime for Java applications.

Developer Workflow with JKube

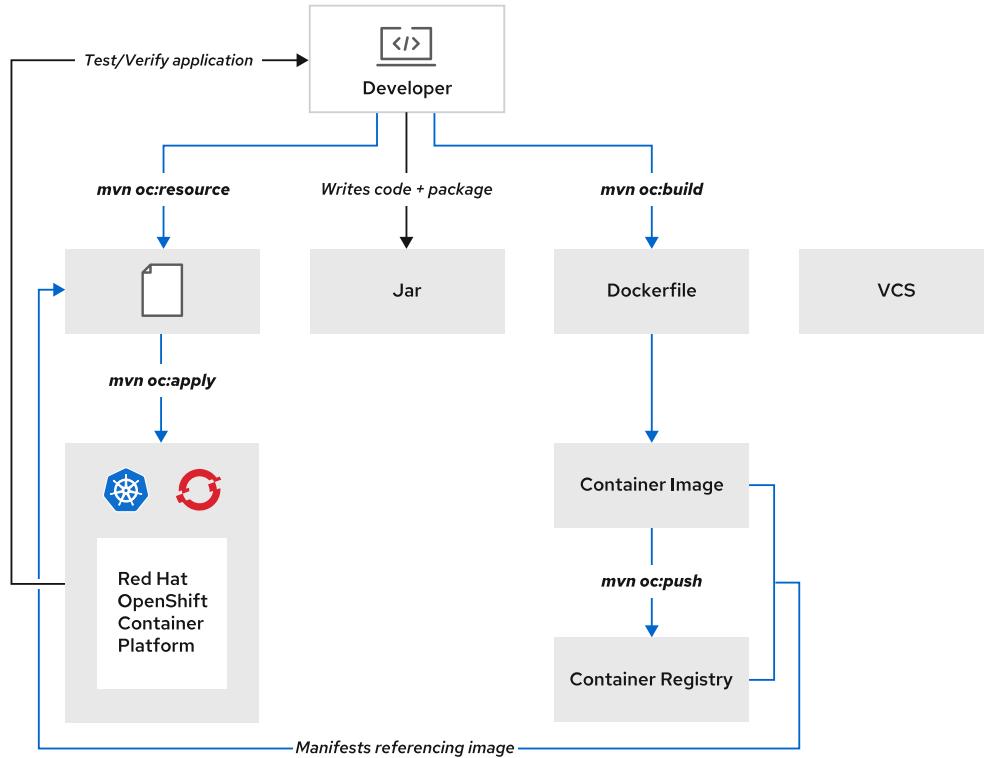


Figure 8.6: Eclipse JKube developer's workflow.

The use of JKube becomes part of the developer's workflow. The different goals help the developer create resource descriptors, deploy them to the OpenShift cluster, build container images, and push them to a desired registry.

There are more functionalities within JKube's Maven plug-ins that are not within the scope of this section, which can aid the developer to improve their workflow.

This workflow enables the developer to perform a zero configuration deployment within a development environment, and both inline configuration and external configuration deployments, for real deployment descriptors.

Kubernetes Maven Plug-in

JKube's Kubernetes Maven Plug-in allows the developer to generate container images, deployment descriptors and configuring deployments.

OpenShift Maven Plug-in

Built on top of the Kubernetes Maven Plug-in, the JKube OpenShift Maven Plug-in is one of the core components of the open source Eclipse JKube project.

You can use the plug-in to deploy Java applications to an OpenShift cluster by 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.

OpenShift Maven Plug-in Configuration

To use the JKube OpenShift Maven plug-in with a project, update the project's `pom.xml` 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 JKube OpenShift Maven plug-in for a Java application:

```
...output omitted...
<build>
  <plugins>
    <plugin>①
      <groupId>org.eclipse.jkube</groupId>
      <artifactId>openshift-maven-plugin</artifactId>
      <version>1.2.0</version>
      <executions>②
        <execution>
          <goals>
            <goal>resource</goal>
            <goal>build</goal>
            <goal>apply</goal>
          </goals>
        </execution>
      </executions>
      <configuration>③
        <!-- additional configuration here -->
      </configuration>
    </plugin>
  </plugins>
</build>
```

- ①** Provides Maven with the required information for it to locate, download, and use the plug-in.
- ②** Configures the standard `install` goal for Maven. This is entirely optional.
- ③** Additional configuration, which can be more specific, allowing you to configure images, resources, volumes for the replica sets, services and config maps to fine-grain details.

Add this XML segment to your project's `pom.xml` file to add the JKube OpenShift Maven plug-in. Refer to the References at the end of this lecture for more details about the JKube OpenShift Maven plug-in configuration.

There is also support for properties that modify several of the different goals. They allow you to modify default behaviors, such as forcing the recreation of a resource whenever you run `oc:apply`, forcing the use of `Deployment` objects instead of using `DeploymentConfig`, and altering other default behavior. You can read more about these properties on the reference page at the end of the section.

OpenShift Maven 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 by using the `mvn` command:

```
[user@host sample-app]$ mvn <plug-in goal name>
```

The OpenShift Maven plug-in provides a set of goals to deal with the development of cloud-native Java applications:

`oc:resource`

Creates Kubernetes and OpenShift resource descriptors. The plug-in stores all generated descriptors in the project's `target/classes/META-INF/openshift` subdirectory.

`oc: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 autodetect 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 `quay.io/jkube/jkube-java-binary-s2i`.

If the plug-in detects access to an OpenShift cluster, then 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.

`oc:apply`

Applies the generated resources to a connected OpenShift cluster.

`oc:deploy`

Similar to `oc:apply`, except it runs in the background.

`oc:undeploy`

Removes resources from the OpenShift cluster.

`oc:watch`

Watches files for changes, which then trigger a rebuild and redeployment. This is useful during development.

The JKube OpenShift 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.



Important

The `package` goal is still required to run in most cases, before the `oc` goals mentioned previously. This means that you need a corresponding build plugin that supports your framework or runtime to build the package, and then you can construct the images, resources and deployment by using JKube.

Customizing OpenShift Resources

With minimal project configuration, the JKube OpenShift 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 might not be sufficient for OpenShift deployment.

You can customize the generated resources in one of two ways: add OpenShift resource YAML fragments to the project's `src/main/jkube` 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 JKube 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/jkube/openshift` directory. All of these resource definitions are combined into an `openshift.yml` file, in the `target/classes/META-INF/jkube` subdirectory for the project.

The plug-in also processes YAML fragment files in the `src/main/jkube` directory and 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, then 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:

<i>Kind</i>	<i>Filename</i>
Service	<code>svc</code> , <code>service</code>
Route	<code>route</code>
Deployment	<code>deployment</code>
DeploymentConfig	<code>dc</code> , <code>deploymentconfig</code>
ConfigMap	<code>cm</code> , <code>configmap</code>
Secret	<code>secret</code>

For example, consider the following content of `src/main/jkube/route.yml`:

```
spec:
  host: app.alternate.com
```

This fragment changes the default host name for the application route to `app.alternate.com`.

You can also create a ConfigMap object with a `src/main/jkube/cm.yml` file. For example:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: example-data
data:
  MSG_TEXT: This is a text value
```



References

The open source documentation for the JKube OpenShift Maven plug-in is available at

<https://www.eclipse.org/jkube/docs/openshift-maven-plugin>

► Guided Exercise

Deploying an Application with JKube

In this exercise, you will use the Eclipse JKube Maven plug-in to deploy a microservice to the OpenShift cluster.

Outcomes

You should be able to:

- Configure a custom OpenShift deployment resource using the Eclipse JKube Maven plug-in.
- Configure an OpenShift configuration map resource by using the Eclipse JKube Maven plug-in.
- Deploy a microservice by using the Eclipse JKube 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
```

Instructions

► 1. Prepare your classroom environment.

- 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
```

- 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...
```

- 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...
```

- 2. Inspect the Java source code of the `micro-java` sample application. The application is a Java Quarkus microservice that is ready for deployment to a non-Kubernetes environment. You add the required configuration to the application to perform a deployment to an OpenShift Container Platform cluster.
- 2.1. Enter the `micro-java` subdirectory of your local clone of the D0288-apps Git repository. Checkout the `main` 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 main
...output omitted...
```

- 2.2. 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.
```

- 2.3. 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 java.util.Optional;

import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

import org.eclipse.microprofile.config.inject.ConfigProperty;

@Path("/api")
public class HelloResource {❶

    @ConfigProperty(name = "HOSTNAME", defaultValue = "unknown")
    String hostname;
    @ConfigProperty(name = "APP_MSG")❷
    Optional<String> message;

    @GET
    @Path("/hello")❸
```

```

@Produces("text/plain")
public String hello() {
    String response = "";

    if (!message.isPresent()) {
        response = "Hello world from host " + hostname + "\n";④
    } else {
        response = "Hello world from host [" + hostname + "].\n";
        response += "Message received = " + message + "\n";⑤
    }
    return response;
}

```

- ➊ This class defines a REST resource for the application.
 - ➋ The application uses the HOSTNAME and APP_MSG environment variables, which are injected via ConfigProperty annotations.
 - ➌ Because you access the resource at /api, you access this endpoint at /api/hello.
 - ➍ If the APP_MSG environment variable is not defined, then the application resource responds with a message containing the host name of the server on which the application is running.
 - ➎ If the APP_MSG environment variable is defined, then the response message contains the value of both the HOSTNAME and APP_MSG environment variables.
- ▶ 3. Configure the application by using the JKube plugin for deployment on OpenShift. Review the configuration of the JKube Maven plug-in in the project pom.xml file:

- 3.1. Update the project-level attributes of the pom.xml file, found near the top, as well as JKube configuration files. Set the jkube.build.switchToDeployment project property to true. This is required because JKube by default uses DeploymentConfig objects with OpenShift, but we want to use Deployment objects instead.

```

<?xml version="1.0" encoding="UTF-8"?>
...output omitted...
<groupId>com.redhat.training.openshift</groupId>
<artifactId>micro-java</artifactId>①
<version>1.0</version>②
...output omitted...
<properties>
...output omitted...
<jkube.build.switchToDeployment>true</jkube.build.switchToDeployment>
</properties>
...output omitted...

```

- ➊ The name and version of the application. The JKube Maven plug-in creates an image stream tag resource from these values, micro-java:1.0.
- ➋ These version properties set the version of the Quarkus application.

- 3.2. Review the list of plug-ins in the `build` section of the `pom.xml`, near the end of the file, and add the following plugin to the end of the build:

```
...output omitted...
<build>
  <plugins>
    ...output omitted...
      <!-- JKube Maven plugin -->
      <plugin>
        <groupId>org.eclipse.jkube</groupId>①
        <artifactId>openshift-maven-plugin</artifactId>
        <version>1.2.0</version>
      </plugin>
    ...output omitted...
```

- ①** We use the `openshift-maven-plugin` plugin for all the building and deployment to OpenShift.

- 3.3. Open the `src/main/jkube/cm.yml` file. This file describes the ConfigMap named `configmap-hello`, which is created in the OpenShift project during deployment.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: configmap-hello
data:
```

- 3.4. Open the `src/main/jkube/deployment.yml` file. This file overrides some default values used by JKube. Add the `envFrom` property for the container, which you can use to set the environment variables from the `configmap-hello` ConfigMap. By adding this reference to the ConfigMap, the defined entries are available to the application as environment variables.

```
...output omitted...
spec:
  containers:
    - env:
        - name: KUBERNETES_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
      envFrom:
        - configMapRef:
            name: configmap-hello
      image: micro-java:1.0
      imagePullPolicy: IfNotPresent
      name: quarkus
    ...output omitted...
```

- 4. Generate OpenShift resources for the application.

- 4.1. From the project directory, execute the `mvn package oc:build oc:resource`. The package creates a JAR file that can be used by JKube for deployment.

`oc:build` performs an S2I build on OpenShift, creating an image named `micro-java` available to the OpenShift project. The `oc:resource` creates several resource files in YAML format, which prepare the application for deployment:

```
[student@workstation micro-java]$ mvn -DskipTests package oc:build oc:resource
...output omitted...
[INFO] -----
[INFO] Building jar: /home/student/D0288-apps/micro-java/target/micro-
java-1.0.jar①
[INFO] -----
...output omitted...
[INFO] --- openshift-maven-plugin:1.2.0:build (default-cli) @ micro-java -②
[INFO] oc: Using OpenShift build with strategy S2I
[INFO] oc: Running in OpenShift mode
[INFO] oc: Running generator quarkus
[INFO] oc: quarkus: Using Docker image quay.io/jkube/jkube-java-binary-s2i:0.0.9
as base / builder
[INFO] oc: [micro-java:latest] "quarkus": Created docker source tar /home/student/
D0288-apps/micro-java/target/docker/micro-java/latest/tmp/docker-build.tar
[INFO] oc: Updating BuildServiceConfig micro-java-s2i for Source strategy
[INFO] oc: Adding to ImageStream micro-java
[INFO] oc: Starting Build micro-java-s2i
[INFO] oc: Waiting for build micro-java-s2i-2 to complete...
...output omitted...
[INFO] --- openshift-maven-plugin:1.2.0:resource (default-cli) @ micro-java -③
[INFO] oc: Using docker image name of namespace: your-project
[INFO] oc: Running generator quarkus
[INFO] oc: quarkus: Using Docker image quay.io/jkube/jkube-java-binary-s2i:0.0.9
as base / builder
[INFO] oc: Using resource templates from /home/student/D0288-apps/micro-java/src/
main/jkube
[INFO] oc: jkube-service: Adding a default service 'micro-java' with ports [8080]
...output omitted...
```

- ① The package goal produces a jar file, required by the rest of the steps. You could also have a native build, which results in a native executable.
- ② The `oc:build` goal containerizes the application by building an image.
- ③ The `oc:resource` Maven goal creates three YAML files. Collectively, these files define a service, deployment, and route resource for the sample application. These files are located in the project `target/classes/META-INF/jkube/openshift` subdirectory.

4.2. Review the generated `micro-java-deployment.yml` file in the `target/classes/META-INF/jkube/openshift` directory.

```
apiVersion: apps.openshift.io/v1
kind: Deployment
metadata:
...output omitted...
  name: micro-java
spec:
  replicas: 1
```

```
...output omitted...
template:
spec:
  containers:
    - env:
        - name: KUBERNETES_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
      envFrom:
        - configMapRef:
            name: configmap-hello
      image: micro-java:1.0
...output omitted...
```

The file defines a deployment named `micro-java` that deploys a single container using an image from the `micro-java:1.0` image stream tag. It is also using the config map as we specified in the `src/main/jkube/deployment.yaml` file.

- 4.3. Review the generated `micro-java-service.yml` file in the `target/classes/META-INF/jkube/openshift` directory.

```
---
apiVersion: v1
kind: Service
metadata:
  ...output omitted...
  name: micro-java
spec:
  ports:
    - name: http
      port: 8080
      protocol: TCP
      targetPort: 8080
  selector:
    app: micro-java
    provider: jkube
  group: com.redhat.training.openshift
```

This file defines a service named `micro-java` for the application pods defined in the `micro-java` deployment.

- 4.4. Review the generated `micro-java-route.yml` file in the `target/classes/META-INF/jkube/openshift` directory.

```
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  ...output omitted...
  name: micro-java
spec:
  port:
    targetPort: 8080
```

```
to:
kind: Service
name: micro-java
```

This file defines a route resource named `micro-java` that exposes the `micro-java` service.

- 4.5. The `oc:resource` Maven goal also generates a combined list of all resources. Review the generated `openshift.yml` file in the project `target/classes/META-INF/jkube` subdirectory:

```
...output omitted...
kind: "List"
...output omitted...
  kind: "Service"
...output omitted...
  kind: "Deployment"
...output omitted...
  kind: "Route"
...output omitted...
```

▶ 5. Build and deploy the application.

- 5.1. Build and deploy the application with the JKube 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 -DskipTests oc:deploy
...output omitted...
[INFO] >>> openshift-maven-plugin:1.2.0:deploy (default-cli) > install @ micro-
java >>>
[INFO]
...output omitted...
[INFO] --- openshift-maven-plugin:1.2.0:deploy (default-cli) @ micro-java -①
[INFO] oc: Using OpenShift at https://api.cluster.domain.example.com:6443 in
namespace your-project with manifest /home/student/D0288-apps/micro-java/target/
classes/META-INF/jkube/openshift.yml ②
[INFO] oc: OpenShift platform detected
[INFO] oc: Creating a Service from openshift.yml namespace your-project name
micro-java③
[INFO] oc: Created Service: target/jkube/applyJson/your-project/service-micro-
java.json
[INFO] oc: Creating a ConfigMap from openshift.yml namespace your-project name
configmap-hello
[INFO] oc: Created ConfigMap: target/jkube/applyJson/your-project/configmap-
configmap-hello.json
[INFO] oc: Creating a Deployment from openshift.yml namespace your-project name
micro-java
[INFO] oc: Created Deployment: target/jkube/applyJson/your-project/deployment-
micro-java.json
[INFO] oc: Creating Route your-project:micro-java host: null
[INFO] oc: HINT: Use the command oc get pods -w to watch your pods start up
[INFO] -----
[INFO] BUILD SUCCESS
```

```
[INFO] -----
[INFO] Total time: 20.921 s ④
...output omitted...
```

- ①** The deployment phase begins.
- ②** The plug-in detects the active OpenShift project. All resources are created in this project.
- ③** The plug-in uses the `openshift.yml` to create an OpenShift service, deployment, and route resource.
- ④** Build time should be within a minute.

You should be able to continue with the activity immediately if the command finishes successfully.

► 6. Review the resources created by the JKube Maven plug-in.

```
[student@workstation micro-java]$ oc status
In project youruser-micro-java on server ...

http://micro-java-youruser... to pod port 8080 (svc/micro-java)
deployment/micro-java deploys ...
deployment #1 deployed 2 minutes ago - 1 pod
bc/micro-java-s2i source builds uploaded code on quay.io/jkube/jkube-java-binary-
s2i:0.0.9
-> istag/micro-java:1.0
...output omitted...
```

A route, service, and deployment resource exist in your project, each with a name of `micro-java`. A build configuration and image stream tag resource also exist in the project.

► 7. Test the application.

- 7.1. Wait for the new application pods to deploy. When the output of the `oc get pods -w` indicates that a new deployment is ready and running, continue to the next step.

```
[student@workstation micro-java]$ oc get pods -w
NAME                      READY   STATUS    RESTARTS   AGE
micro-java-a1b2c3d4e5f6-5pw6q   1/1     Running   1          14m
micro-java-s2i-1-build         0/1     Completed  0          16m
```

- 7.2. 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=$(oc get route \
micro-java --template='{{.spec.host}}')
[student@workstation D0288-apps]$ curl ${ROUTE_URL}/api/hello
Hello world from host micro-java-1-5pw6q
```

Because the application deployment does not define a value for the `APP_MSG` environment variable, the output only contains the container host name.

- 8. Update the project to deploy application pods with a new value for the APP_MSG environment variable.

- 8.1. Update the `src/main/jkube/cm.yml` fragment file to provide it with a new APP_MSG value.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: configmap-hello
data:
  APP_MSG: this is a new value
```

The preceding YAML fragment defines a configuration map resource named `configmap-hello`. This configuration map defines an APP_MSG variable with a value of sample external configuration.

- 8.2. Commit and push the YAML fragments to the `micro-config` branch:

```
[student@workstation micro-java]$ git add src/main/jkube/*.yml
[student@workstation micro-java]$ git commit -am "Add YAML fragments."
[student@workstation micro-java]$ git push
...output omitted...
```

- 9. Redeploy the application. Verify that a JKube 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.

- 9.1. Redeploy the application with the JKube Maven plug-in:

```
[student@workstation micro-java]$ mvn -DskipTests oc:build oc:resource oc:apply
...output omitted...
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
...output omitted...
```

- 9.2. Verify the presence of the `configmap-hello` configuration map:

```
[student@workstation micro-java]$ oc get cm/configmap-hello
NAME          DATA   AGE
configmap-hello    1    84m
```

- 9.3. Wait for the new application pods to deploy. Use the command `oc get pods -w` to wait, as done in a previous step.

```
[student@workstation micro-java]$ oc get pods -w
NAME           READY   STATUS    RESTARTS   AGE
micro-java-1a2b3c4d5e6f-n2rn2   1/1     Running   0          108s
micro-java-s2i-1-build   0/1     Completed   0          15m
```

- 9.4. Verify the application response includes the new value of the APP_MSG variable:

```
[student@workstation micro-java]$ curl ${ROUTE_URL}/api/hello
Hello world from host [micro-java-3-m9k4j].
Message received = this is a new value
```



Note

If the previous `curl` returns an error HTML page stating that the application is not available, then it means the application container is still not ready to serve requests. Wait a few seconds and try the same command again.

- ▶ **10.** 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

In this lab, you will use the Eclipse JKube 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 JKube YAML fragments.
- Deploy the `To Do List` back-end application using the JKube Maven plug-in.

Before You Begin

To perform this exercise, you need access to:

- A running OpenShift cluster.
- The `todo-api` 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` 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.ocp-` For example, if the wildcard domain of your cluster is `apps.cluster.domain.example.com`, then the database FQDN is `mysql.ocp-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:

Use a JKube YAML fragment to create the configuration map resource.

- DATABASE_USER: todoapp
- DATABASE_PASSWORD: redhat123
- DATABASE_SVC_HOSTNAME: tododb
- DATABASE_NAME: todo
 - A custom OpenShift deployment resource is created as part of the deployment.

Use a JKube 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.

Instructions

1. Verify connectivity to the external database server.
2. Create an OpenShift service named `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` 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 JKube 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 JKube 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

In this lab, you will use the Eclipse JKube 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 JKube YAML fragments.
- Deploy the To Do List back-end application using the JKube Maven plug-in.

Before You Begin

To perform this exercise, you need access to:

- A running OpenShift cluster.
- The `todo-api` 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` 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.ocp-` For example, if the wildcard domain of your cluster is `apps.cluster.domain.example.com`, then the database FQDN is `mysql.ocp-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:

Use a JKube YAML fragment to create the configuration map resource.

- DATABASE_USER: todoapp
- DATABASE_PASSWORD: redhat123
- DATABASE_SVC_HOSTNAME: tododb
- DATABASE_NAME: todo
 - A custom OpenShift deployment resource is created as part of the deployment.

Use a JKube 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.

Instructions

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...
mysql>
```

- 1.4. Exit the MySQL client to return to the shell prompt.

```
mysql> exit
Bye
[student@workstation ~]$
```

2. Create an OpenShift service named `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 project subdirectory.

```
[student@workstation ~]$ cd ~/D0288-apps
[student@workstation D0288-apps]$ git checkout main
...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
[student@workstation todo-api]$
```

4. Build and deploy the application. Verify that the application pod fails to deploy because the required environment variables are not defined.

- 4.1. Compile, build and deploy the application.

```
[student@workstation todo-api]$ mvn clean compile package oc:build oc:resource
oc:apply
```

- 4.2. After the application deploys, monitor the logs for the application pod.

```
[student@workstation todo-api]$ oc get pods
NAME          READY   STATUS        RESTARTS   AGE
todo-api-558555647-mpg9j   0/1     CrashLoopBackOff   3          101s
todo-api-s2i-1-build       0/1     Completed      0          2m34s
```

The exact STATUS of the pod can vary, but the number of RESTARTS increases.

**Note**

The STATUS of the pod might be Running before reaching the CrashLoopBackOff state.

```
[student@workstation todo-api]$ oc logs -f todo-api-558555647-mpg9j
```

...output omitted...

One or more configuration errors have prevented the application from starting. The errors are:

- SRCFG00011: Could not expand value DATABASE_USER in property `quarkus.datasource.username`
- SRCFG00011: Could not expand value DATABASE_PASSWORD in property `quarkus.datasource.password`
- SRCFG00011: Could not expand value DATABASE_SVC_HOSTNAME in property `quarkus.datasource.jdbc.url`

A failure occurs because the DATABASE_SVC_HOSTNAME and DATABASE_NAME, among other environment variables, are not defined.

5. Create the YAML fragment that JKube 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/jkube/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/DO288/solutions/todo-migrate` directory. You can copy it to the `src/main/jkube` subdirectory:

```
[student@workstation todo-api]$ cp \
~/DO288/solutions/todo-migrate/cm.yml src/main/jkube
```

6. Create the YAML fragment that JKube uses to generate the custom deployment configuration resource.

- 6.1. Create the `src/main/jkube/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/jkube subdirectory:

```
[student@workstation todo-api]$ cp \
~/D0288/solutions/todo-migrate/deployment.yml src/main/jkube
```

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]$ mvn oc:resource oc:apply
```

- 7.2. Review the resources created by the JKube Maven plug-in.

```
[student@workstation todo-api]$ oc describe deployment/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]$ oc get configmap
NAME           DATA   AGE
db-config       4      5m16s
...output omitted...
```

Verify that the application pod is in a Running state.

```
[student@workstation todo-api]$ oc get pods
NAME           READY   STATUS    RESTARTS   AGE
...output omitted...
todo-api-58fc84655-gs9nv     1/1     Running   0          35s
...output omitted...
```

- 7.3. Wait for the application to be ready and running:

```
[student@workstation todo-api]$ oc logs -f todo-api-58fc84655-gs9nv  
...output omitted...  
INFO: todo-api 1.0.0-SNAPSHOT on JVM (powered by Quarkus 1.11.7.Final-redhat-00009) started in 3.183s. Listening on: http://0.0.0.0:8080
```

- 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]$ ROUTE_URL=$(oc get route todo-api \  
--template={{.spec.host}})  
[student@workstation todo-api]$ curl -s ${ROUTE_URL}/todo/api/items/6 \  
| jq  
{  
  "id": 6,  
  "description": "Verify that the To Do List application works",  
  "done": false  
}
```

8. After application tests succeed, commit and push your code changes to the remote repository.

```
[student@workstation todo-api]$ git add src/main/jkube/*  
[student@workstation todo-api]$ git commit -m "add YAML fragments"  
...output omitted...  
[student@workstation todo-api]$ 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 `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 runnable JAR files.
- The JKube Maven plug-in provides features to generate OpenShift resources and trigger OpenShift processes, such as builds and deployments.

Chapter 9

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: Building and Deploying a Multi-container Application on OpenShift

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, Managing Builds on OpenShift

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, Deploying Multi-container Applications

Deploy multi-container applications using Helm charts and Kustomize.

- Describe the elements of an OpenShift template.
- Build a multicontainer application using Helm Charts.
- Customize OpenShift deployments.

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.

The objectives for this chapter are not included in the Comprehensive Review lab.

Chapter 8, 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

Comprehensive Review

In this review, you will deploy a multicontainer To Do List application on OpenShift. This application is composed of four components:

- A MySQL (MariaDB) database server.
- An HTTP API back-end, based on Node.js.
- A single-page web front-end, based on React and Nginx.
- A task export service, which displays the status of tasks in the list

Outcomes

You should be able to:

- Deploy a multicontainer application on OpenShift by using a variety of build and deploy strategies.
- Optimize a Containerfile to reduce the number of layers in the generated container image.
- Build and publish a container image to an external registry.
- Create Helm Charts to encapsulate reusable configuration.
- Deploy a Node.js application from source code in a Git repository, passing build environment variables.
- Create and consume Config Maps to store application configuration parameters.
- Use Source-to-Image (S2I) to deploy an application.

Before You Begin

To perform this exercise, ensure that you have access to:

- A running OpenShift cluster.
- The MariaDB 10.3 Helm Chart provided by Bitnami.
- The Node.js 12 builder image.
- A personal GitHub fork and a local clone of the DO288-apps repository, which contains the application source code in the directories named `todo-frontend`, `todo-backend`, and `todo-ssr`.

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 review-todo start
```

Specifications

- Containerize and deploy the four components of the application to a new OpenShift project named \${RHT_OCP4_DEV_USER}-review-todo.

Ensure that the deployment follows Red Hat's recommendations for externalizing the configuration of applications deployed to OpenShift.

- Back end requirements:

Deploy the back end and database components using Helm Charts.

The chart should deploy the quay.io/redhattraining/todo-backend:release-46 tag from Quay.io.

The chart should always pull the image on a new deployment.

It should deploy MariaDB 10.3 database using version 9.3.11 of the Bitnami chart as a dependency of the chart.

The environment variables required by the API pod are DATABASE_USER, DATABASE_PASSWORD, DATABASE_NAME, and DATABASE_SVC.

The service should bind to port 3000.

Expose a public route to access the API.

- Front end (SPA) requirements:

Retrieve the todo-frontend sources and Containerfile from a clone of your personal fork of the DO288-apps Git repository.

The provided Containerfile generates an image that is compliant with Open Container Initiative (OCI) container engines, but might require changes to comply with Red Hat recommendations for OpenShift.

Make no change to the HTML and TypeScript sources.

Fix the container image by specifying the user named nginx and port 8080.

Minimize the number of layers in the container image.

Build and publish the todo-frontend container image into your personal Quay.io account with quay.io/yourquayuser/front-end:latest as the name and tag.

Deploy the To Do List UI in the \${RHT_OCP4_DEV_USER}-review-todo project.

Set the environment variable BACKEND_HOST on the front end deployment to the service name of the todo-backend API.

Expose a public route to access the UI.

Test the To Do List front end by using the exposed route.

- Front end (static) requirements:

Use the Source-to-Image (S2I) strategy to build and deploy the application in the todo-ssr directory within the DO288-apps repository.

It should use the Red Hat Node.js 12 S2I builder, which is provided by OpenShift.

Provide the application name `todo-ssr` to the S2I builder.

Create and use a Config Map named `todo-ssr-host` to set the environment variable `API_HOST` to point to the `todo-backend` service on port 3000.

Expose a public route to access the UI.

- Hints:

For the back end service, the `/api/items` endpoint returns the current list of to do items or `[]` if no items exist.

While the database is initializing, it is normal for the application API pod to fail and restart.

If you need to uninstall your Helm Chart, you must also delete the Persistence Volume Claim (PVC) that is associated with the MariaDB database server (deleting the project will also delete the PVC).

The SPA front end is working if you are able to create new to do list items that persist when you refresh the page.

Use the `app=todo-frontend` selector to delete *only* the UI resources.

The static front end is working if you are able to view a page that lists the to do list items you created using the SPA version of the front end.

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 review-todo grade
```

Finish

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

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

This concludes the comprehensive review.

► Solution

Comprehensive Review

In this review, you will deploy a multicontainer To Do List application on OpenShift. This application is composed of four components:

- A MySQL (MariaDB) database server.
- An HTTP API back-end, based on Node.js.
- A single-page web front-end, based on React and Nginx.
- A task export service, which displays the status of tasks in the list

Outcomes

You should be able to:

- Deploy a multicontainer application on OpenShift by using a variety of build and deploy strategies.
- Optimize a Containerfile to reduce the number of layers in the generated container image.
- Build and publish a container image to an external registry.
- Create Helm Charts to encapsulate reusable configuration.
- Deploy a Node.js application from source code in a Git repository, passing build environment variables.
- Create and consume Config Maps to store application configuration parameters.
- Use Source-to-Image (S2I) to deploy an application.

Before You Begin

To perform this exercise, ensure that you have access to:

- A running OpenShift cluster.
- The MariaDB 10.3 Helm Chart provided by Bitnami.
- The Node.js 12 builder image.
- A personal GitHub fork and a local clone of the D0288-apps repository, which contains the application source code in the directories named `todo-frontend`, `todo-backend`, and `todo-ssr`.

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 review-todo start
```

Specifications

- Containerize and deploy the four components of the application to a new OpenShift project named \${RHT_OCP4_DEV_USER}-review-todo.

Ensure that the deployment follows Red Hat's recommendations for externalizing the configuration of applications deployed to OpenShift.

- Back end requirements:

Deploy the back end and database components using Helm Charts.

The chart should deploy the quay.io/redhattraining/todo-backend:release-46 tag from Quay.io.

The chart should always pull the image on a new deployment.

It should deploy MariaDB 10.3 database using version 9.3.11 of the Bitnami chart as a dependency of the chart.

The environment variables required by the API pod are DATABASE_USER, DATABASE_PASSWORD, DATABASE_NAME, and DATABASE_SVC.

The service should bind to port 3000.

Expose a public route to access the API.

- Front end (SPA) requirements:

Retrieve the todo-frontend sources and Containerfile from a clone of your personal fork of the DO288-apps Git repository.

The provided Containerfile generates an image that is compliant with Open Container Initiative (OCI) container engines, but might require changes to comply with Red Hat recommendations for OpenShift.

Make no change to the HTML and TypeScript sources.

Fix the container image by specifying the user named nginx and port 8080.

Minimize the number of layers in the container image.

Build and publish the todo-frontend container image into your personal Quay.io account with quay.io/yourquayuser/front-end:latest as the name and tag.

Deploy the To Do List UI in the \${RHT_OCP4_DEV_USER}-review-todo project.

Set the environment variable BACKEND_HOST on the front end deployment to the service name of the todo-backend API.

Expose a public route to access the UI.

Test the To Do List front end by using the exposed route.

- Front end (static) requirements:

Use the Source-to-Image (S2I) strategy to build and deploy the application in the todo-ssr directory within the DO288-apps repository.

It should use the Red Hat Node.js 12 S2I builder, which is provided by OpenShift.

Provide the application name `todo-ssr` to the S2I builder.

Create and use a Config Map named `todo-ssr-host` to set the environment variable `API_HOST` to point to the `todo-backend` service on port 3000.

Expose a public route to access the UI.

- **Hints:**

For the back end service, the `/api/items` endpoint returns the current list of to do items or `[]` if no items exist.

While the database is initializing, it is normal for the application API pod to fail and restart.

If you need to uninstall your Helm Chart, you must also delete the Persistence Volume Claim (PVC) that is associated with the MariaDB database server (deleting the project will also delete the PVC).

The SPA front end is working if you are able to create new to do list items that persist when you refresh the page.

Use the `app=todo-frontend` selector to delete *only* the UI resources.

The static front end is working if you are able to view a page that lists the to do list items you created using the SPA version of the front end.

1. Within a new OpenShift project named `${RHT_OCP4_DEV_USER} -review-todo`, create a Helm Chart that deploys the Node.js back end. The new chart should have a dependency on the Bitnami MariaDB Helm Chart. Create a new route to the API so that it is publicly available.
 - 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 by using your developer user account:

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
-p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful
...output omitted...
```
 - 1.3. Create a new OpenShift project named `${RHT_OCP4_DEV_USER} -review-todo`.

```
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-review-todo
Now using project "youruser-review-todo" on server ...output omitted...
...output omitted...
```
 - 1.4. Within your lab directory, initialize a new Helm Chart named `todo-list` and change to the newly created directory.

```
[student@workstation ~]$ cd ~/D0288/labs/review-todo
[student@workstation review-todo]$ helm create todo-list
Creating todo-list
[student@workstation review-todo]$ cd todo-list
[student@workstation todo-list]$
```

**Note**

A finished version of the Helm Chart is provided in `~/D0288/solutions/review-todo/todo-list`. If you are unsure of your solution, you can reference or copy the one provided.

- 1.5. Append the following to the contents of the `Chart.yaml` file, which declares the MariaDB chart as a dependency.

```
dependencies:
- name: mariadb
  version: 9.3.11
  repository: https://charts.bitnami.com/bitnami
```

- 1.6. Pull the dependencies.

```
[student@workstation todo-list]$ helm dependency update
Getting updates for unmanaged Helm repositories...
...Successfully got an update from the "https://charts.bitnami.com/bitnami" chart
repository
Saving 1 charts
Downloading mariadb from repo https://charts.bitnami.com/bitnami
Deleting outdated charts
```

- 1.7. Within the `values.yaml` file, update the values in the `image` section.

```
image:
  repository: quay.io/redhattraining/todo-backend
  pullPolicy: Always
  # Overrides the image tag whose default is the chart appVersion.
  tag: "release-46"
```

- 1.8. Append the following sections to the end of `values.yaml`:

```
mariadb:
  auth:
    username: todouser
    password: todopwd
    database: tododb
  primary:
    podSecurityContext:
      enabled: false
    containerSecurityContext:
      enabled: false
```

```
env:
  - name: DATABASE_NAME
    value: tododb
  - name: DATABASE_USER
    value: todouser
  - name: DATABASE_PASSWORD
    value: todopwd
  - name: DATABASE_SVC
    value: todo-list-mariadb
```

- 1.9. Also within `values.yaml`, update the `port` value in the `service` to 3000.

```
service:
  type: ClusterIP
  port: 3000
```

- 1.10. Update `templates/deployment.yaml` to add an `env` section within the `containers` section, and set the `containerPort` to 3000. Make sure the indentation matches up with the following.

```
apiVersion: apps/v1
...
spec:
  ...
  template:
    spec:
      ...
      containers:
        - name: {{ .Chart.Name }}
          securityContext:
            {{- toYaml .Values.securityContext | nindent 12 }}
            image: "{{ .Values.image.repository }}:{{ .Values.image.tag | default .Chart.AppVersion }}"
            imagePullPolicy: {{ .Values.image.pullPolicy }}
          env:
            {{- range .Values.env }}
            - name: {{ .name }}
              value: {{ .value }}
            {{- end }}
          ports:
            - name: http
              containerPort: 3000
              protocol: TCP
```

- 1.11. Install the chart, providing the name `todo-list`:

```
[student@workstation todo-list]$ helm install todo-list .
NAME: todo-list
LAST DEPLOYED: ...output omitted...
NAMESPACE: youruser-review-todo
STATUS: deployed
REVISION: 1
...output omitted...
```

This will create all of the resources defined by the Helm Chart in the currently selected project.

- 1.12. Verify that the application has successfully started:

```
[student@workstation todo-list]$ oc get pods
[student@workstation review-todo]$ oc get pods
NAME                      READY   STATUS    RESTARTS   AGE
todo-list-7b6dbb8ccb-bqvvz   1/1     Running   2          5m57s
todo-list-mariadb-0           1/1     Running   0          5m57s
```



Note

If your API pod continues to crash and restart, you must uninstall your Helm Chart, fix it, and try again.

- 1.13. Create a new route to the API.

```
[student@workstation todo-list]$ oc expose svc/todo-list
route.route.openshift.io/todo-list exposed
```

- 1.14. Retrieve the public URL to the service.

```
[student@workstation todo-list]$ export URL_TO_APPLICATION=$(oc get \
route/todo-list -o jsonpath='{.spec.host}')
...output omitted...
```

- 1.15. Use the URL to verify the API is started by connecting to it.

```
[student@workstation todo-list]$ curl ${URL_TO_APPLICATION}
OK
```

- 1.16. Use the URL to verify the service is connected to the database.

```
[student@workstation todo-list]$ curl ${URL_TO_APPLICATION}/api/items
[]
```

2. Build and deploy the React UI on OpenShift using new-app by pushing an image to Quay.io. Fix and build the todo-frontend image so that it uses the nginx user, exposes port 8080, and minimizes the number of layers. Create a new route to the UI so that it is publicly available.

- 2.1. From your fork of D0288-apps, open the `todo-frontend/Containerfile` file and combine the separate RUN commands into a single instruction.

```
RUN cd /tmp/todo-frontend && \
  npm install && \
  npm run build
```

**Note**

A fixed version of the Containerfile is provided in `~/D0288/solutions/review-todo/Containerfile-frontend-solution`. If you are unsure of your solution, you can reference or copy the one provided.

- 2.2. Also within the `todo-frontend/Containerfile` file, add the EXPOSE and USER commands:

```
COPY --from=appbuild /tmp/todo-frontend/build /usr/share/nginx/html

EXPOSE 8080

USER nginx

CMD nginx -g "daemon off;"
```

- 2.3. Build the image locally with Podman.

```
[student@workstation todo-frontend]$ cd ~/D0288-apps/todo-frontend/
[student@workstation todo-frontend]$ podman build . \
-t quay.io/${RHT_OCP4_QUAY_USER}/todo-frontend:latest
STEP 1: FROM registry.access.redhat.com/ubi8/nodejs-14 AS appbuild
Getting image source signatures
...output omitted...
STEP 18: COMMIT quay.io/youruser/todo-frontend:latest
...output omitted...
```

Note that this step might take several minutes to complete.

- 2.4. Within a browser, navigate to Quay.io and create a new empty repository named `todo-frontend`. Below the name field, set the repository as public. Otherwise, OpenShift will not be able to access it.
- 2.5. Log in into your personal Quay.io account using Podman.

```
[student@workstation ~]$ podman login -u ${RHT_OCP4_QUAY_USER} quay.io
Password:
Login Succeeded!
```

- 2.6. Push the image to Quay.io with Podman.

```
[student@workstation ~]$ podman push quay.io/${RHT_OCP4_QUAY_USER}/todo-frontend
Getting image source signatures
...output omitted...
Writing manifest to image destination
Storing signatures
```

Note that you must be authenticated to Quay.io.

2.7. Deploy the image by using the image stream via `oc new-app`.

```
[student@workstation ~]$ oc new-app quay.io/${RHT_OCP4_QUAY_USER}/todo-frontend
--> Found container image ...output omitted...
...output omitted...
--> Creating resources ...
  imagestream.image.openshift.io "todo-frontend" created
  deployment.apps "todo-frontend" created
  service "todo-frontend" created
--> Success
...output omitted...
```

2.8. Verify that the application has successfully started:

```
[student@workstation ~]$ oc get pods
NAME                      READY   STATUS    RESTARTS   AGE
...output omitted...
todo-frontend-7b4d77b4f8-lsrpm   1/1     Running   0          1m20s
```

2.9. Create a route to the service with the `expose` command:

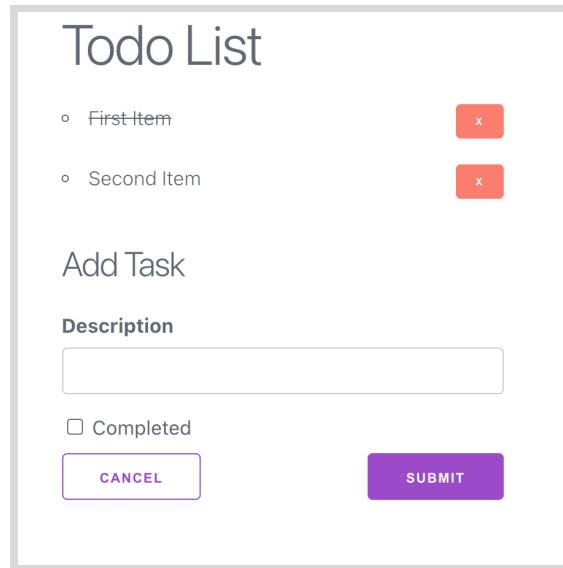
```
[student@workstation ~]$ oc expose svc/todo-frontend
route.route.openshift.io/todo-frontend exposed
```

2.10. Retrieve the public URL by examining the route.

```
[student@workstation ~]$ oc get route todo-frontend -o jsonpath='{.spec.host}'
```

2.11. Verify the UI works by opening the retrieved URL in a browser. The UI should contain an SPA to do list application.

2.12. Verify you can create To Do list items that persist when you refresh the page.



3. Deploy a server-side rendered Node.js application named `todo-ssr` using S2I. The source code is available in the DO288-apps repository within the `todo-ssr` directory. Make sure the service runs using port 3000. Create a new route to the UI so that it is publicly available.

- 3.1. Use `new-app` to initialize the `todo-ssr` application using S2I.

```
[student@workstation ~]$ oc new-app \
https://github.com/RedHatTraining/DO288-apps \
--name todo-ssr --context-dir=todo-ssr --build-env \
npm_config_registry="${RHT_OCP4_NEXUS_SERVER}/repository/nodejs"
--> Found image 9350f28 (5 months old) in image stream "openshift/nodejs" under
tag "12-ubi8" for "nodejs"
...output omitted...
--> Creating resources ...
imagestream.image.openshift.io "todo-ssr" created
buildconfig.build.openshift.io "todo-ssr" created
deployment.apps "todo-ssr" created
service "todo-ssr" created
--> Success
...output omitted...
```

- 3.2. Create a Config Map named `todo-ssr-host` that contains the `API_HOST` environment variable.

```
[student@workstation ~]$ oc create configmap todo-ssr-host \
--from-literal API_HOST="http://todo-list:3000"
configmap/todo-ssr-host created
```

- 3.3. Connect the Config Map to the `todo-ssr` Deployment.

```
[student@workstation ~]$ oc set env deployment/todo-ssr \
--from cm/todo-ssr-host
deployment.apps/todo-ssr updated
```

Note that it might take a few minutes for the application to restart after attaching the Config Map.

- 3.4. Verify that the application has successfully started:

```
[student@workstation ~]$ oc get pods
NAME                  READY   STATUS    RESTARTS   AGE
...output omitted...
todo(ssr-1-build      0/1     Completed  0          6m30s
todo(ssr-64bc5f8987-7654f 1/1     Running   0          1m12s
```

- 3.5. Create a route to the service with the expose command.

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

- 3.6. Retrieve the public URL by examining the route.

```
[student@workstation ~]$ oc get route todo-ssr -o jsonpath='{.spec.host}'
...output omitted...
```

- 3.7. Verify the static UI works by opening the retrieved URL in a browser. The UI should contain a static page with the To Do list items you created in the SPA version of the UI.

Todo List

Note that this version is non-interactive

- foo
- baz

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 review-todo grade
```

Finish

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

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

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.

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

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 repositories.

To create a new GitHub account navigate to <https://github.com>, click **Sign up**, and follow the prompts. 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.

Creating GitHub Repositories

Once authenticated, create new Git repositories by clicking **New** in the **Repositories** pane on the left of the GitHub home page.

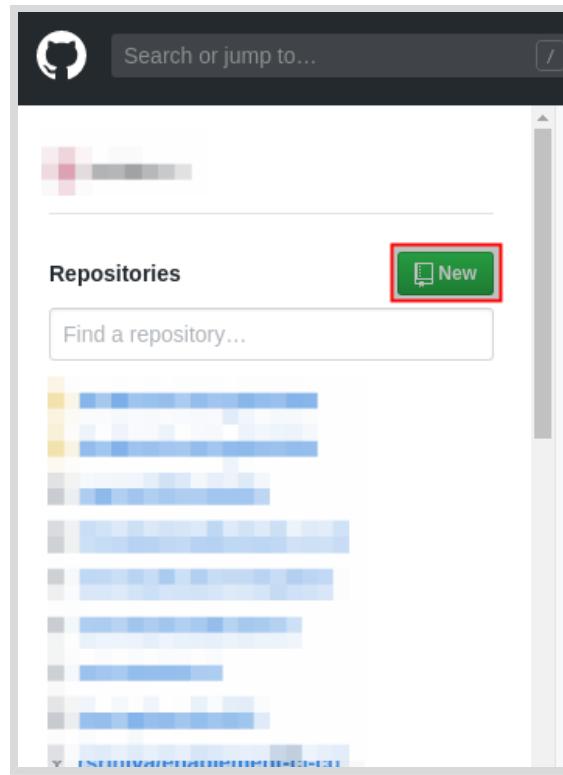


Figure A.1: Creating a new Git repository

Alternatively, click the plus icon (+) in the upper-right corner (to the right of the bell icon) and then click **New repository**.

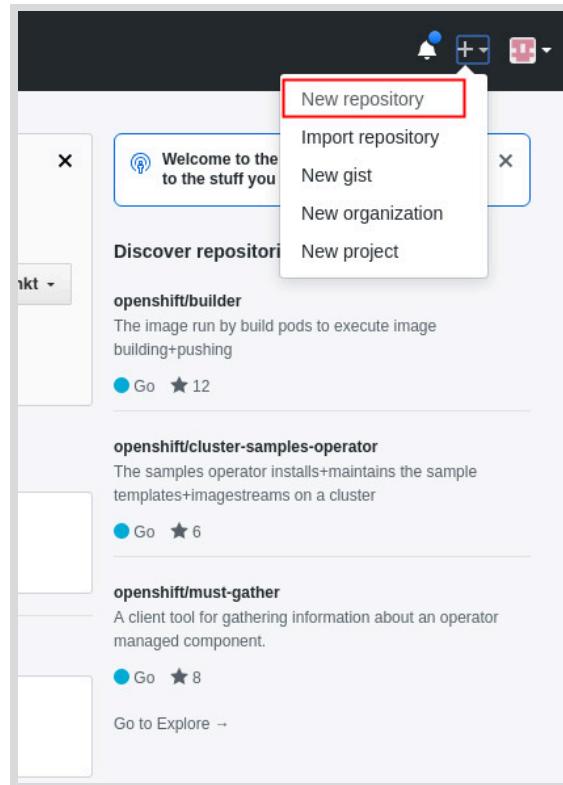


Figure A.2: Creating new Git repository

Forking GitHub Repositories

To fork a repository on GitHub, navigate to the repository and click **Fork** in the top right. Make sure to click **Fork** and not the *number* next to the button.

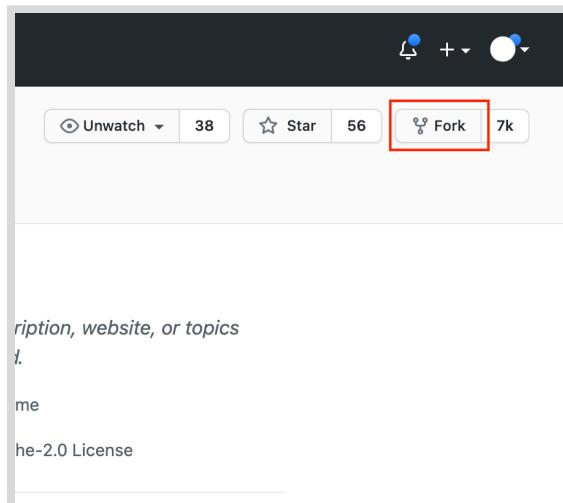


Figure A.3: Forking a Git repository

Select your username as the destination and wait for the process to finish.

When cloning the repository to your workstation, use the URL for your fork found by clicking **Code**.

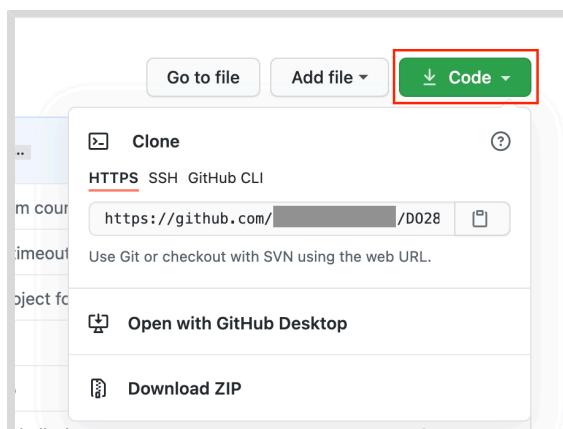


Figure A.4: Cloning a forked repository

Updating Your Fork

If the RedHatTraining copy of the repository is updated, then you will need to update your fork. To update your fork, add a remote to your local copy of the repository, pull the changes, and then push the changes to your fork.

```
[student@workstation D0288-apps]$ git remote add upstream \
  https://github.com/RedHatTraining/D0288-apps.git
[student@workstation D0288-apps]$ git pull upstream main
...output omitted...
[student@workstation D0288-apps]$ git push origin main
...output omitted...
```

Creating a GitHub Personal Access Token

GitHub has deprecated password-based authentication. This means that you must use a personal access token to run Git commands that require authentication, such as cloning private repositories or pushing changes to GitHub.

If you do not already have a personal access token, then use the Creating a personal access token [https://docs.github.com/en/github/authenticating-to-github/keeping-your-account-and-data-secure/creating-a-personal-access-token] guide to create a new one.

Some exercises in this course will require you to authenticate to GitHub. When asked for your password, enter your personal access token.



References

Signing up for a new GitHub account

<https://help.github.com/en/articles/signing-up-for-a-new-github-account>

Creating a personal access token

<https://docs.github.com/en/github/authenticating-to-github/keeping-your-account-and-data-secure/creating-a-personal-access-token>

Appendix B

Creating a Quay Account

Goal

Describe how to create a Quay account for labs in the course.

Creating a Quay Account

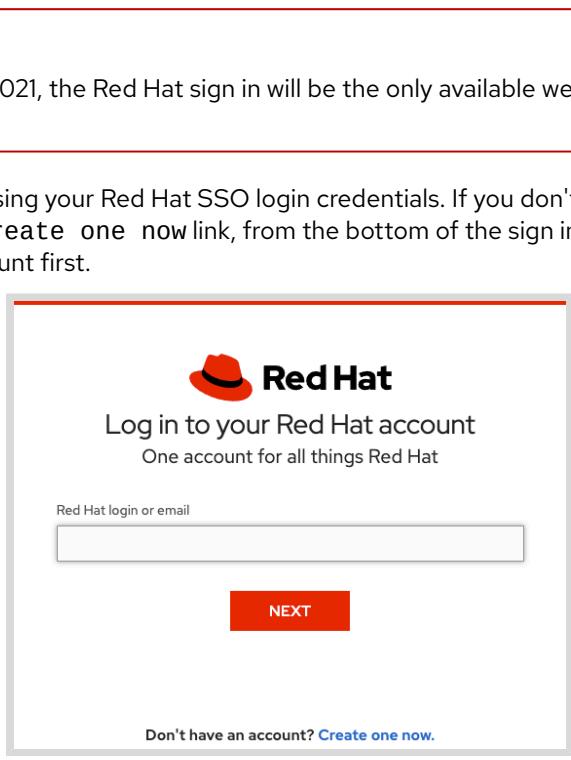
Objectives

After completing this section, you should be able to create a Quay account and use an encrypted password for the labs in the course which require Quay.io access.

Creating a Quay Account

You need a Quay account to create one or more 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.

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, log in using your Red Hat credentials. The image shows a screenshot of a web browser window with a red border. Inside, there's a Red Hat logo at the top. Below it, the text "Log in to your Red Hat account" and "One account for all things Red Hat". There's a text input field labeled "Red Hat login or email" with a placeholder "Email or Red Hat ID". Below the input field is a red "NEXT" button. At the bottom of the window, there's a link "Don't have an account? [Create one now.](#)".



Warning

After July 31, 2021, the Red Hat sign in will be the only available web login option.

4. Log into Quay.io using your Red Hat SSO login credentials. If you don't have a Red Hat SSO account use the [create one now](#) link, from the bottom of the sign in dialog, to create a free Red Hat SSO account first.

Creating an image repository

In the labs, image repositories are created from the command-line tools. However, you can also create them in the web interface.

Appendix B | Creating a Quay Account

1. After you have logged in to Quay.io you can create new image repositories by clicking **Create New Repository** on the Repositories page.
2. Alternatively, click the plus icon (+) in the upper-right corner (to the left of the bell icon), and then click **Create New Repository**.

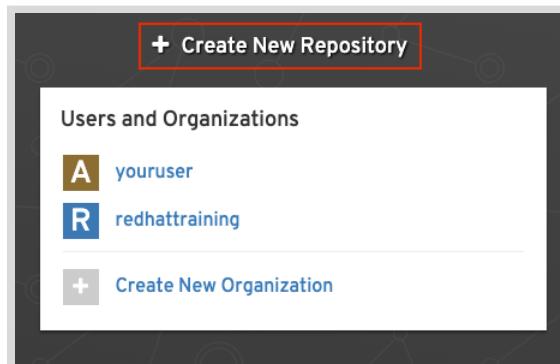


Figure B.2: Creating a new image repository

Making a repository public

By default, new repositories are created *private*. While this is sufficient for the labs that use login secrets, some labs require *public* repositories. Private repositories have a small lock icon next to the repository name.

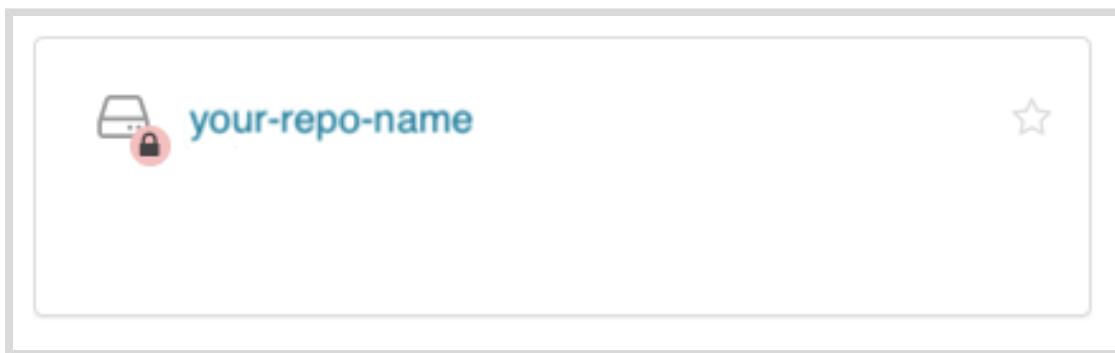


Figure B.3: Private repository

Use the following steps to make a repository public.

1. Select the link for your repository.
2. Click the *Settings* icon from the left side menu.

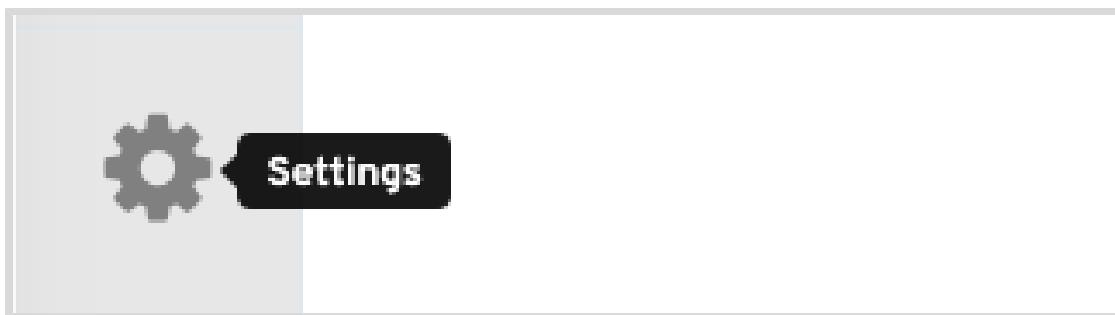
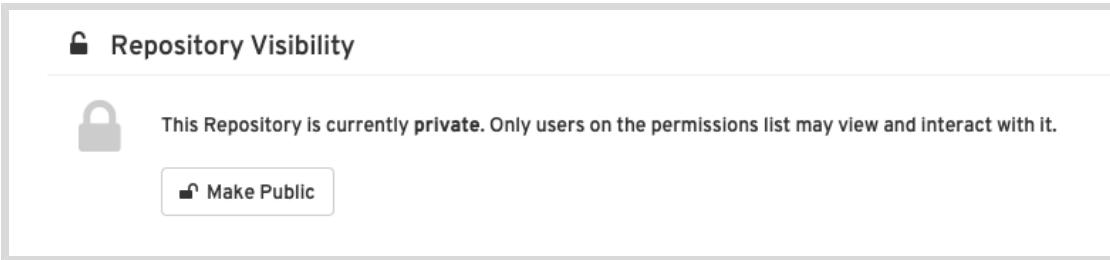


Figure B.4: Quay repository settings

3. From the settings page click **Make Public** from the **Repository Visibility** section.

**Figure B.5: Make the repository public**

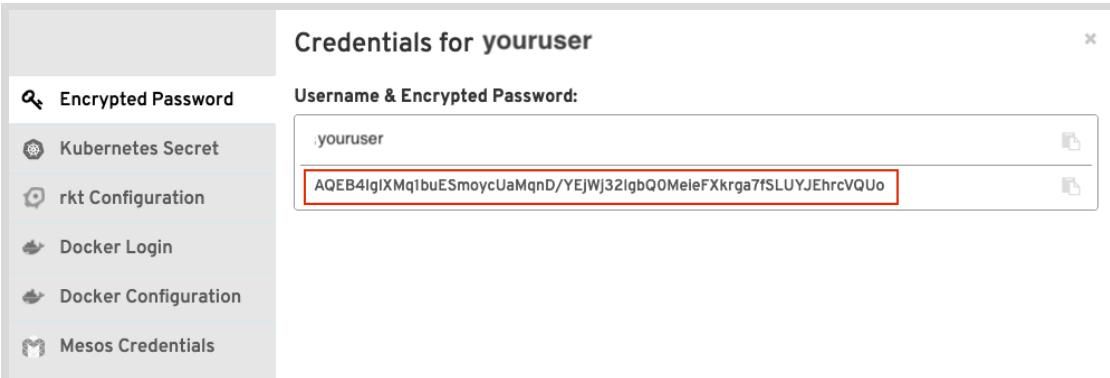
Working with CLI tools

After you create your account with Red Hat SSO, you need to set an account password to use CLI tools like Podman.

1. Click your name in the upper-right corner.
2. Click **Account Settings**.
3. Click **Change password**.

The podman CLI stores passwords entered on the command line in plaintext. It is therefore highly recommended to generate an encrypted version of your password to use with podman `login`.

4. From the top of the **Account Settings** page click **Generate Encrypted Password**.
5. Enter your account password at the prompt and generate the encrypted equivalent.

**Figure B.6: Encrypted Quay password**

6. In the lab exercises, use the encrypted password when prompted for a Quay.io password.

```
[student@workstation]$ podman login -u ${RHT_OCP4_QUAY_USER} quay.io
Password: <use your encrypted password here>
Login Succeeded!
```



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 commit any changes on the `main` branch.

A list of scenarios and the corresponding commands that you can use are listed below.

Abandoning an Exercise Completely and Restarting It

To redo an exercise from scratch after you have completed it, perform the following steps:

1. Commit and push all the changes in your local branch as part of performing the exercise. Finish 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 D0288-apps repository and switch to the `main` branch:

```
[student@workstation ~]$ cd ~/D0288-apps  
[student@workstation D0288-apps]$ git checkout main
```

3. Delete your local branch:

```
[student@workstation D0288-apps]$ git branch -d your-branch
```

4. Delete the remote branch on your personal GitHub account:

```
[student@workstation D0288-apps]$ git push origin --delete your-branch
```

5. Discard any pending changes using `git reset --hard`:

```
[student@workstation D0288-apps]$ git reset --hard
```

Appendix C | Useful Git Commands

6. If you have changes that you will want to use later, but you need to temporarily discard, use `git stash -u`:

```
[student@workstation D0288-apps]$ git stash -u
```

7. Use the `start` subcommand to restart the exercise:

```
[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 reserve shared resources and you might exhaust your quota on the cloud provider and on the RHOCUP cluster. If you think it might 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:

1. Verify which files you modified, to see what to add to your commit.
2. Add files with changes you want to save.
3. Commit the changes to 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 status  
...output omitted...  
[student@workstation D0288-apps]$ git add some/file/you/changed  
[student@workstation D0288-apps]$ git commit -m 'Paused at step X.Y'  
[student@workstation D0288-apps]$ git push origin branch-name
```

4. 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.
5. Start the next exercise by running its `start` subcommand:

```
[student@workstation ~]$ lab your-exercise start
```

6. The next exercise switches to the `main` 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.

```
[student@workstation D0288-apps]$ git checkout main
```

7. 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 branch-name
```

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-ls>

Git Tools - Stashing

<https://git-scm.com/book/en/v1/Git-Tools-Stashing>