





Join a community dedicated to learning open source

The Red Hat® Learning Community is a collaborative platform for users to accelerate open source skill adoption while working with Red Hat products and experts.



Network with tens of thousands of community members



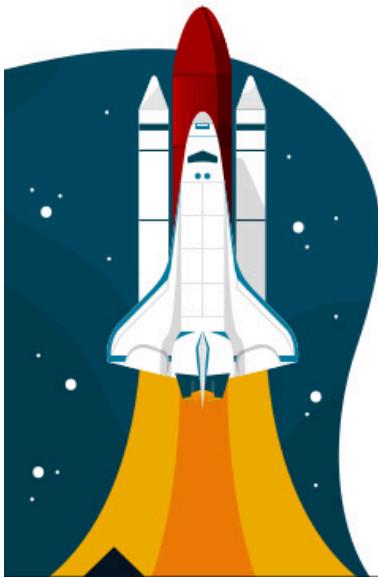
Engage in thousands of active conversations and posts



Join and interact with hundreds of certified training instructors



Unlock badges as you participate and accomplish new goals



This knowledge-sharing platform creates a space where learners can connect, ask questions, and collaborate with other open source practitioners.

Access free Red Hat training videos

Discover the latest Red Hat Training and Certification news

Connect with your instructor - and your classmates - before, after, and during your training course.

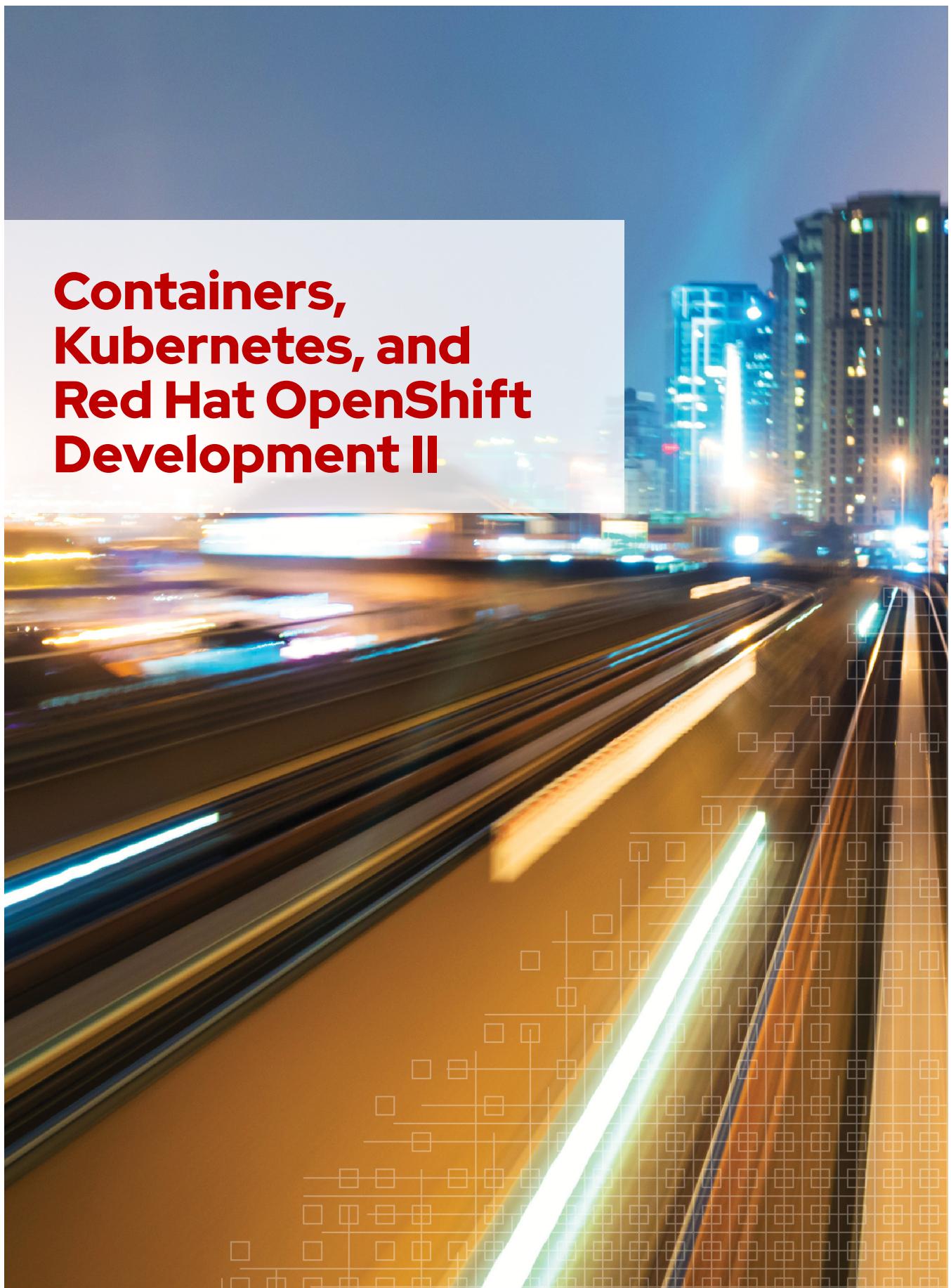
Join peers as you explore Red Hat products

Join the conversation learn.redhat.com



Copyright © 2020 Red Hat, Inc. Red Hat, Red Hat Enterprise Linux, the Red Hat logo, and Ansible are trademarks or registered trademarks of Red Hat, Inc. or its subsidiaries in the United States and other countries.

Containers, Kubernetes, and Red Hat OpenShift Development II



OCP 4.6 DO295
Containers, Kubernetes, and Red Hat OpenShift Development
II
Edition 2 20211213
Publication date 20211213

Authors: Richard Allred, Federico Fapitalle, Zach Guterman,
Michael Jarrett, Dan Kolepp, Maria Fernanda Ordonez Casado,
Eduardo Ramirez Ronco, Harpal Singh, Jordi Sola Alaball
Editor: Sam Ffrench, Seth Kenlon, Nicole Muller, Connie Petlitzer, Dave
Sacco

Copyright © 2021 Red Hat, Inc.

The contents of this course and all its modules and related materials, including handouts to audience members, are
Copyright © 2021 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: Forrest Taylor, Manuel Aude Morales, James Mighion, Michael Phillips, Fiona Allen

| | |
|--|-------------|
| Document Conventions | xi |
| | xi |
| Introduction | xiii |
| Containers, Kubernetes, and Red Hat OpenShift Development II | xiii |
| Orientation to the Classroom Environment | xiv |
| Performing Lab Exercises | xx |
| 1. Introducing Container Technology | 1 |
| Overview of Container Technology | 2 |
| Quiz: Overview of Container Technology | 5 |
| Overview of Container Architecture | 9 |
| Quiz: Overview of Container Architecture | 12 |
| Overview of Kubernetes and OpenShift | 14 |
| Quiz: Describing Kubernetes and OpenShift | 17 |
| Guided Exercise: Configuring the Classroom Environment | 19 |
| Summary | 29 |
| 2. Creating Containerized Services | 31 |
| Provisioning Containerized Services | 32 |
| Guided Exercise: Creating a MySQL Database Instance | 38 |
| Summary | 41 |
| 3. Managing Containers | 43 |
| Managing the Life Cycle of Containers | 44 |
| Guided Exercise: Managing a MySQL Container | 52 |
| Attaching Persistent Storage to Containers | 55 |
| Guided Exercise: Create MySQL Container with Persistent Database | 59 |
| Accessing Containers | 62 |
| Guided Exercise: Loading the Database | 64 |
| Lab: Managing Containers | 67 |
| Summary | 75 |
| 4. Managing Container Images | 77 |
| Accessing Registries | 78 |
| Quiz: Working With Registries | 84 |
| Manipulating Container Images | 88 |
| Guided Exercise: Creating a Custom Apache Container Image | 94 |
| Lab: Managing Images | 99 |
| Summary | 107 |
| 5. Creating Custom Container Images | 109 |
| Designing Custom Container Images | 110 |
| Quiz: Approaches to Container Image Design | 114 |
| Building Custom Container Images with Containerfiles | 116 |
| Guided Exercise: Creating a Basic Apache Container Image | 122 |
| Lab: Creating Custom Container Images | 126 |
| Summary | 133 |
| 6. Deploying Containerized Applications on OpenShift | 135 |
| Creating Kubernetes Resources | 136 |
| Guided Exercise: Deploying a Database Server on OpenShift | 148 |
| Creating Routes | 153 |
| Guided Exercise: Exposing a Service as a Route | 157 |
| Creating Applications with Source-to-Image | 162 |
| Guided Exercise: Creating a Containerized Application with Source-to-Image | 172 |
| Lab: Deploying Containerized Applications on OpenShift | 178 |
| Summary | 182 |

| | |
|---|------------|
| 7. Deploying and Managing Applications on an OpenShift Cluster | 183 |
| Managing Applications with the Web Console | 184 |
| Guided Exercise: Managing an Application with the Web Console | 189 |
| Managing Applications with the CLI | 197 |
| Guided Exercise: Managing an Application with the CLI | 202 |
| Summary | 211 |
| 8. Designing Containerized Applications for OpenShift | 213 |
| Selecting a Containerization Approach | 214 |
| Quiz: Selecting a Containerization Approach | 218 |
| Building Container Images with Advanced Containerfile Instructions | 224 |
| Guided Exercise: Building Container Images with Advanced Containerfile Instructions | 232 |
| Injecting Configuration Data into an Application | 241 |
| Guided Exercise: Injecting Configuration Data into an Application | 249 |
| Lab: Designing Containerized Applications for OpenShift | 257 |
| Summary | 268 |
| 9. Publishing Enterprise Container Images | 269 |
| Managing Images in an Enterprise Registry | 270 |
| Guided Exercise: Using an Enterprise Registry | 278 |
| Allowing Access to the OpenShift Registry | 284 |
| Guided Exercise: Allowing Access to the OpenShift Registry | 288 |
| Creating Image Streams | 292 |
| Guided Exercise: Creating an Image Stream | 299 |
| Lab: Publishing Enterprise Container Images | 303 |
| Summary | 312 |
| 10. Managing Builds on OpenShift | 313 |
| Describing the Red Hat OpenShift Build Process | 314 |
| Quiz: The OpenShift Build Process | 318 |
| Managing Application Builds | 324 |
| Guided Exercise: Managing Application Builds | 328 |
| Triggering Builds | 334 |
| Guided Exercise: Triggering Builds | 337 |
| Implementing Post-commit Build Hooks | 343 |
| Guided Exercise: Implementing Post-Commit Build Hooks | 345 |
| Lab: Building Applications for OpenShift | 350 |
| Summary | 358 |
| 11. Customizing Source-to-Image Builds | 359 |
| Customizing an Existing S2I Base Image | 360 |
| Guided Exercise: Customizing S2I Builds | 363 |
| Creating an S2I Base Image | 368 |
| Guided Exercise: Creating an S2I Base Image | 374 |
| Lab: Customizing Source-to-Image Builds | 385 |
| Summary | 399 |
| 12. Deploying Multi-container Applications | 401 |
| Creating a Helm Chart | 402 |
| Guided Exercise: Creating a Helm Chart | 406 |
| Customizing a Deployment with Kustomize | 411 |
| Guided Exercise: Customizing Deployments with Kustomize | 415 |
| Lab: Deploying Multi-container Applications | 421 |
| Summary | 433 |
| 13. Managing Application Deployments | 435 |
| Monitoring Application Health | 436 |
| Guided Exercise: Activating Probes | 442 |

| | |
|---|------------|
| Selecting the Appropriate Deployment Strategy | 449 |
| Guided Exercise: Implementing a Deployment Strategy | 454 |
| Managing Application Deployments with CLI Commands | 462 |
| Guided Exercise: Managing Application Deployments | 467 |
| Lab: Managing Application Deployments | 474 |
| Summary | 485 |
| 14. Building Applications for OpenShift | 487 |
| Integrating External Services | 488 |
| Guided Exercise: Integrating an External Service | 490 |
| Deploying Cloud-Native Applications with JKube | 494 |
| Guided Exercise: Deploying an Application with JKube | 500 |
| Lab: Building Cloud-Native Applications for OpenShift | 510 |
| Summary | 519 |
| 15. Comprehensive Review: Containers, Kubernetes, and Red Hat OpenShift Development II | 521 |
| Comprehensive Review | 522 |
| Lab: Comprehensive Review | 524 |
| A. Creating a GitHub Account | 537 |
| Creating a GitHub Account | 538 |
| B. Creating a Quay Account | 541 |
| Creating a Quay Account | 542 |
| Repository Visibility | 545 |
| C. Creating a Red Hat Account | 549 |
| Creating a Red Hat Account | 550 |
| D. Useful Git Commands | 553 |
| Git Commands | 554 |

Document Conventions

This section describes various conventions and practices that are used throughout all Red Hat Training courses.

Admonitions

Red Hat Training courses use the following admonitions:



References

References describe where to find external documentation that is 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 something that makes your life easier.



Important

They provide details of information that is easily missed: configuration changes that apply only to the current session, or services that need restarting before an update applies. Ignoring these admonitions will not cause data loss, but might cause irritation and frustration.



Warning

Warnings should not be ignored. Ignoring these admonitions will most likely cause data loss.

Inclusive Language

Red Hat Training is currently reviewing its use of language in various areas to help remove any potentially offensive terms. This is an ongoing process and requires alignment with the products and services that are covered in Red Hat Training courses. Red Hat appreciates your patience during this process.

Introduction

Containers, Kubernetes, and Red Hat OpenShift Development II

Containers, Kubernetes, and Red Hat OpenShift Development II (DO295) teaches students how to design, build, and deploy containerized software applications to an OpenShift cluster. Whether writing container native applications or migrating existing brownfield applications, this course provides hands-on training to boost developer productivity powered by Red Hat OpenShift for developers, administrators, and site reliability engineers.

Course Objectives

- Design container images to containerize applications.
- Customize application builds and implement post-commit build hooks.
- Create a multi container application template.
- Implement health checks to improve system reliability.

Audience

- Developers who wish to containerize software applications.
- Administrators who are new to container technology and container orchestration.
- Architects who are considering using container technologies in software architectures.
- Site Reliability Engineers who are considering using Kubernetes and OpenShift.

Prerequisites

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

Orientation to the Classroom Environment

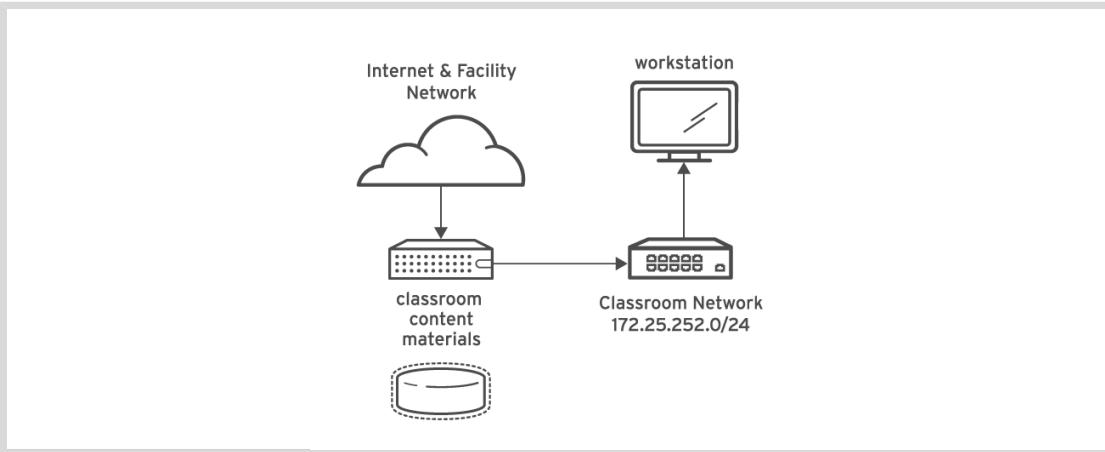


Figure O.1: Classroom environment

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

rht-vmctl Commands

| Action | Command |
|--|-------------------------------------|
| Start the virtual machine named server . | <code>rht-vmctl start server</code> |
| Display the system console for the virtual machine named server , to log in and work on the system. | <code>rht-vmview view server</code> |
| Reset the virtual machine named server to its initial course state and restart it. | <code>rht-vmctl reset server</code> |

You are assigned remote computers in a Red Hat Online Learning classroom. Self-paced courses are accessed through a web application that is hosted at rol.redhat.com [<http://rol.redhat.com>]. If your course is an instructor-led virtual training event, you will be provided with your course location URL. Log in to this site with your Red Hat Customer Portal user credentials.

Controlling the Virtual Machines

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

The screenshot shows a web-based interface for managing a lab environment. At the top, there are tabs for 'Table of Contents', 'Course', 'Lab Environment', and icons for a star and a question mark. Below this, a section titled 'Lab Controls' contains instructions: 'Click CREATE to build all of the virtual machines needed for the classroom lab environment. This may take several minutes to complete. Once created the environment can then be stopped and restarted to pause your experience.' It also states that deleting the lab will remove all virtual machines and lose progress. Below this section is a table listing five virtual machines:

| | | DELETE | STOP | |
|-------------|----------|--------|------|-----------------------|
| bastion | active | | | ACTION - OPEN CONSOLE |
| classroom | active | | | ACTION - OPEN CONSOLE |
| servera | building | | | ACTION - OPEN CONSOLE |
| serverb | building | | | ACTION - OPEN CONSOLE |
| workstation | active | | | ACTION - OPEN CONSOLE |

Figure 0.2: An example course Lab Environment management page

Machine States

| Virtual Machine State | Description |
|------------------------------|---|
| building | The virtual machine is being created. |
| active | The virtual machine is running and available. If just started, it might still be starting services. |
| stopped | The virtual machine is completely shut down. On starting, the virtual machine boots into the same state as it was before it was shut down. The disk state is preserved. |

Classroom Actions

| Button or Action | Description |
|-------------------------|--|
| CREATE | Create the ROLE classroom. Creates and starts all of the virtual machines that are needed for this classroom. |
| CREATING | The ROLE classroom virtual machines are being created. Creates and starts all of the virtual machines that are needed for this classroom. Creation can take several minutes to complete. |
| DELETE | Delete the ROLE classroom. Destroys all virtual machines in the classroom. <i>All saved work on that system's disks is lost.</i> |
| START | Start all virtual machines in the classroom. |
| STARTING | All virtual machines in the classroom are starting. |
| STOP | Stop all virtual machines in the classroom. |

Machine Actions

| Button or Action | Description |
|-------------------------|--|
| OPEN CONSOLE | Connect to the system console of the virtual machine in a new browser tab. You can log in directly to the virtual machine and run commands, when required. Normally, log in to the workstation virtual machine only, and from there, 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 disk contents. |
| ACTION → Power Off | Forcefully shut down the virtual machine, while still preserving disk contents. 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. <i>All saved work on that system's disks is lost.</i> |

Introduction

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** on every virtual machine in the list.

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



Warning

The **DELETE** operation cannot be undone. All completed work in the classroom environment is lost.

The Auto-stop and Auto-destroy Timers

The Red Hat Online Learning enrollment entitles you to a set allotment of computer time. To help to conserve your allotted time, the ROLE classroom uses timers, which shut down or delete the classroom when the appropriate timer expires.

To adjust the timers, locate the two **] buttons at the bottom of the course management page.** Click the auto-stop{nbsp}btn:[button to add another hour to the auto-stop timer. Click the auto-destroy + button to add another day to the auto-destroy timer. Auto-stop has a maximum of 11 hours, and auto-destroy has a maximum of 14 days. Be careful to keep the timers set while you are working, so that your environment is not unexpectedly shut down. Be careful not to set the timers unnecessarily high, which could waste your subscription time allotment.

Controlling Your Systems

You are assigned remote computers in a Red Hat Online Learning classroom. Self-paced courses are accessed through a web application that is hosted at rol.redhat.com [<http://rol.redhat.com>]. If your course is an instructor-led virtual training event, you will be provided with your course location URL. Log in to this site with your Red Hat Customer Portal user credentials.

Controlling the Virtual Machines

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

The screenshot shows a web-based interface for managing a lab environment. At the top, there are tabs for 'Table of Contents', 'Course', 'Lab Environment', and icons for a star and help. Below this is a section titled 'Lab Controls' with instructions for creating and deleting the lab environment. A large button at the bottom left says 'DELETE' and a larger one next to it says 'STOP'. To the right of these buttons is an information icon. The main area displays a table of virtual machines:

| Machine Name | Status | Action | Open Console |
|--------------|----------|----------|--------------|
| bastion | active | ACTION ▾ | OPEN CONSOLE |
| classroom | active | ACTION ▾ | OPEN CONSOLE |
| servera | building | ACTION ▾ | OPEN CONSOLE |
| serverb | building | ACTION ▾ | OPEN CONSOLE |
| workstation | active | ACTION ▾ | OPEN CONSOLE |

Figure 0.3: An example course Lab Environment management page

Machine States

| Virtual Machine State | Description |
|-----------------------|---|
| building | The virtual machine is being created. |
| active | The virtual machine is running and available. If just started, it might still be starting services. |
| stopped | The virtual machine is completely shut down. On starting, the virtual machine boots into the same state as it was before it was shut down. The disk state is preserved. |

Classroom Actions

| Button or Action | Description |
|------------------|--|
| CREATE | Create the ROLE classroom. Creates and starts all of the virtual machines that are needed for this classroom. |
| CREATING | The ROLE classroom virtual machines are being created. Creates and starts all of the virtual machines that are needed for this classroom. Creation can take several minutes to complete. |
| DELETE | Delete the ROLE classroom. Destroys all virtual machines in the classroom. All saved work on that system's disks is lost. |
| START | Start all virtual machines in the classroom. |
| STARTING | All virtual machines in the classroom are starting. |
| STOP | Stop all virtual machines in the classroom. |

Machine Actions

| Button or Action | Description |
|--------------------|--|
| OPEN CONSOLE | Connect to the system console of the virtual machine in a new browser tab. You can log in directly to the virtual machine and run commands, when required. Normally, log in to the workstation virtual machine only, and from there, use <code>ssh</code> to connect to the other virtual machines. |
| ACTION → Start | Start (power on) the virtual machine. |
| ACTION → Shutdown | Gracefully shut down the virtual machine, preserving disk contents. |
| ACTION → Power Off | Forcefully shut down the virtual machine, while still preserving disk contents. 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. All saved work on that system's disks 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** on every virtual machine in the list.

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



Warning

The **DELETE** operation cannot be undone. All completed work in the classroom environment is lost.

The Auto-stop and Auto-destroy Timers

The Red Hat Online Learning enrollment entitles you to a set allotment of computer time. To help to conserve your allotted time, the ROLE classroom uses timers, which shut down or delete the classroom when the appropriate timer expires.

To adjust the timers, locate the two + buttons at the bottom of the course management page. Click the auto-stop + button to add another hour to the auto-stop timer. Click the auto-destroy + button to add another day to the auto-destroy timer. Auto-stop has a maximum of 11 hours, and auto-destroy has a maximum of 14 days. Be careful to keep the timers set while you are working, so that your environment is not unexpectedly shut down. Be careful not to set the timers unnecessarily high, which could waste your subscription time allotment.

Performing Lab Exercises

Run the `lab` command from `workstation` to prepare your environment before each hands-on exercise, and again to clean up after an exercise. Each hands-on exercise has a unique name within a course. The exercise is prepended with `lab-` as its file name in `/usr/local/lib`. For example, the `instances-cli` exercise has the file name `/usr/local/lib/lab-instances-cli`. To list the available exercises, use tab completion in the `lab` command:

```
[student@workstation ~]$ lab Tab Tab
administer-users  deploy-overcloud-lab  prep-deploy-ips      stacks-autoscale
analyze-metrics   instances-cli        prep-deploy-router  stacks-deploy
assign-roles       manage-interfaces  public-instance-deploy verify-overcloud
```

Exercises are in two types. The first type, a *guided exercise*, is a practice exercise that follows a course narrative. If a narrative is followed by a quiz, then usually the topic did not have an achievable practice exercise. The second type, an *end-of-chapter lab*, is a gradable exercise to help verify your learning. When a course includes a comprehensive review, the review exercises are structured as gradable labs. The syntax for running an exercise script is:

```
[student@workstation ~]$ lab exercise action
```

The `action` is a choice of `start`, `grade`, or `finish`. All exercises support `start` and `finish`. Only end-of-chapter labs and comprehensive review labs support `grade`. Older courses might still use `setup` and `cleanup` instead of the current `start` and `finish` actions.

start

Formerly `setup`. A script's start logic verifies the required resources to begin an exercise. It might include configuring settings, creating resources, checking prerequisite services, and verifying necessary outcomes from previous exercises. You can take an exercise at any time, even without taking prerequisite exercises.

grade

End-of-chapter labs help verify what you have learned, after practicing with earlier guided exercises. The `grade` action directs the `lab` command to display a list of grading criteria, with a `PASS` or `FAIL` status for each. To achieve a `PASS` status for all criteria, fix the failures and rerun the `grade` action.

finish

Formerly `cleanup`. A script's finish logic deletes exercise resources that are no longer necessary. You can take an exercise as many times as you want.

Troubleshooting Lab Scripts

Exercise scripts do not exist on `workstation` until each is first run. When you run the `lab` command with a valid exercise and action, the script named `lab-exercise` is downloaded from the `classroom` server content share to `/usr/local/lib` on `workstation`. The `lab` command creates two log files in `/var/tmp/labs`, plus the directory if it does not exist. One file, named `exercise`, captures standard output messages that are normally displayed on your terminal. The other file, named `exercise.err`, captures error messages.

```
[student@workstation ~]$ ls -l /usr/local/lib
-rwxr-xr-x. 1 root root 4131 May  9 23:38 lab-instances-cli
-rwxr-xr-x. 1 root root 93461 May  9 23:38 labtool.cl110.shlib
-rwxr-xr-x. 1 root root 10372 May  9 23:38 labtool.shlib

[student@workstation ~]$ ls -l /var/tmp/labs
-rw-r--r--. 1 root root 113 May  9 23:38 instances-cli
-rw-r--r--. 1 root root 113 May  9 23:38 instances-cli.err
```



Note

Scripts download from the `http://content.example.com/courses/course/release/grading-scripts` share, but only if the script does not yet exist on `workstation`. When you need to download a script again, such as when a script on the share is modified, manually delete the current exercise script from `/usr/local/lib` on `workstation`, and then run the `lab` command for the exercise again. The newer exercise script then downloads from the `grading-scripts` share.

To delete all current exercise scripts on `workstation`, use the `lab` command's `--refresh` option. A refresh deletes all scripts in `/usr/local/lib` but does not delete the log files.

```
[student@workstation ~]$ lab --refresh
[student@workstation ~]$ ls -l /usr/local/lib

[student@workstation ~]$ ls -l /var/tmp/labs
-rw-r--r--. 1 root root 113 May  9 23:38 instances-cli
-rw-r--r--. 1 root root 113 May  9 23:38 instances-cli.err
```

Interpreting the Exercise Log Files

Exercise scripts send output to log files, even when the scripts are successful. Step header text is added between steps, and additional date and time headers are added at the start of each script run. The exercise log normally contains messages that indicate successful completion of command steps. Therefore, the exercise output log is useful for observing messages that are expected if no problems occur, but offers no additional help when failures occur.

Instead, the exercise error log is more useful for troubleshooting. Even when the scripts succeed, messages are still sent to the exercise error log. For example, a script that verifies that an object already exists before attempting to create it should cause an *object not found* message when the object does not exist yet. In this scenario, that message is expected and does not indicate a failure. Actual failure messages are typically more verbose, and experienced system administrators should recognize common log message entries.

Although exercise scripts are always run from `workstation`, they perform tasks on other systems in the course environment. Many course environments, including OpenStack and OpenShift, use a command-line interface (CLI) that is invoked from `workstation` to communicate with server systems by using API calls. Because script actions typically distribute tasks to multiple systems, additional troubleshooting is necessary to determine where a failed task occurred. Log in to those other systems and use Linux diagnostic skills to read local system log files and determine the root cause of the lab script failure.

Chapter 1

Introducing Container Technology

Goal

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

Objectives

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

Sections

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

Overview of Container Technology

Objectives

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

Containerized Applications

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

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

The major drawback to a traditionally deployed software application is that the application's dependencies are entangled with the runtime environment.

An application may break when any updates or patches are applied to the base operating system (OS).

For example, an OS update to the TLS shared library removes TLS 1.0 as a supported protocol. This breaks the deployed Python application because it is written to use the TLS 1.0 protocol for network requests. This forces the system administrator to roll back the OS update to keep the application running, preventing other applications from using the benefits of the updated package.

Therefore, a company developing traditional software applications may require a full set of tests to guarantee that an OS update does not affect applications running on the host.

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

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

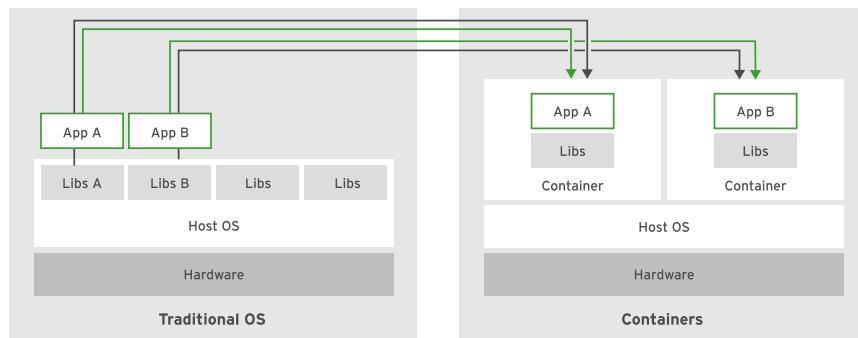


Figure 1.1: Container versus operating system differences

Alternatively, a software application can be deployed using a container.

A container is a set of one or more processes that are isolated from the rest of the system.

Containers provide many of the same benefits as virtual machines, such as security, storage, and network isolation. Containers require far fewer hardware resources and are quick to start and terminate. They also isolate the libraries and the runtime resources (such as CPU and storage) for an application to minimize the impact of any OS update to the host OS, as described in *Figure 1.1*.

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

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

The following are other major advantages to using containers:

Low hardware footprint

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

Environment isolation

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

Quick deployment

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

Multiple environment deployment

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

Reusability

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

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

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

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



References

Home - Open Containers Initiative

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

► Quiz

Overview of Container Technology

Choose the correct answers to the following questions:

- ▶ 1. **Which two options are examples of software applications that might run in a container? (Choose two.)**
 - a. A database-driven Python application accessing services such as a MySQL database, a file transfer protocol (FTP) server, and a web server on a single physical host.
 - b. A Java Enterprise Edition application with an Oracle database, and a message broker running on a single VM.
 - c. An I/O monitoring tool responsible for analyzing the traffic and block data transfer.
 - d. A memory dump application tool capable of taking snapshots from all the memory CPU caches for debugging purposes.
- ▶ 2. **Which two of the following use cases are best suited for containers? (Choose two.)**
 - a. A software provider needs to distribute software that can be reused by other companies in a fast and error-free way.
 - b. A company is deploying applications on a physical host and would like to improve its performance by using containers.
 - c. Developers at a company need a disposable environment that mimics the production environment so that they can quickly test the code they develop.
 - d. A financial company is implementing a CPU-intensive risk analysis tool on their own containers to minimize the number of processors needed.
- ▶ 3. **A company is migrating their PHP and Python applications running on the same host to a new architecture. Due to internal policies, both are using a set of custom made shared libraries from the OS, but the latest update applied to them as a result of a Python development team request broke the PHP application. Which two architectures would provide the best support for both applications? (Choose two.)**
 - a. Deploy each application to different VMs and apply the custom-made shared libraries individually to each VM host.
 - b. Deploy each application to different containers and apply the custom-made shared libraries individually to each container.
 - c. Deploy each application to different VMs and apply the custom-made shared libraries to all VM hosts.
 - d. Deploy each application to different containers and apply the custom-made shared libraries to all containers.

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

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

► Solution

Overview of Container Technology

Choose the correct answers to the following questions:

- ▶ 1. **Which two options are examples of software applications that might run in a container? (Choose two.)**
 - a. A database-driven Python application accessing services such as a MySQL database, a file transfer protocol (FTP) server, and a web server on a single physical host.
 - b. A Java Enterprise Edition application with an Oracle database, and a message broker running on a single VM.
 - c. An I/O monitoring tool responsible for analyzing the traffic and block data transfer.
 - d. A memory dump application tool capable of taking snapshots from all the memory CPU caches for debugging purposes.
- ▶ 2. **Which two of the following use cases are best suited for containers? (Choose two.)**
 - a. A software provider needs to distribute software that can be reused by other companies in a fast and error-free way.
 - b. A company is deploying applications on a physical host and would like to improve its performance by using containers.
 - c. Developers at a company need a disposable environment that mimics the production environment so that they can quickly test the code they develop.
 - d. A financial company is implementing a CPU-intensive risk analysis tool on their own containers to minimize the number of processors needed.
- ▶ 3. **A company is migrating their PHP and Python applications running on the same host to a new architecture. Due to internal policies, both are using a set of custom made shared libraries from the OS, but the latest update applied to them as a result of a Python development team request broke the PHP application. Which two architectures would provide the best support for both applications? (Choose two.)**
 - a. Deploy each application to different VMs and apply the custom-made shared libraries individually to each VM host.
 - b. Deploy each application to different containers and apply the custom-made shared libraries individually to each container.
 - c. Deploy each application to different VMs and apply the custom-made shared libraries to all VM hosts.
 - d. Deploy each application to different containers and apply the custom-made shared libraries to all containers.

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

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

Overview of Container Architecture

Objectives

After completing this section, you should be able to:

- Describe the architecture of Linux containers.
- Describe the podman tool for the managing of containers.

Introducing Container History

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

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

Namespaces

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

Control groups (cgroups)

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

Seccomp

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

SELinux

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

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

A common use case of containers is having several replicas of the same service (for example, a database server) in the same host. Each replica has isolated resources (file system, ports, memory), so there is no need for the service to handle resource sharing. Isolation guarantees that

a malfunctioning or harmful service does not impact other services or containers in the same host, nor in the underlying system.

Describing Linux Container Architecture

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

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

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

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

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



Note

This course uses the public image registry Quay, so students can operate with images without worrying about interfering with each other.

Managing Containers with Podman

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

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

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



References

Red Hat Quay Container Registry

<https://quay.io>

Podman site

<https://podman.io/>

Open Container Initiative

<https://www.opencontainers.org>

► Quiz

Overview of Container Architecture

Choose the correct answers to the following questions:

- ▶ 1. **Which three of the following Linux features are used for running containers? (Choose three.)**
 - a. Namespaces
 - b. Integrity Management
 - c. Security-Enhanced Linux
 - d. Control Groups

- ▶ 2. **Which of the following best describes a container image?**
 - a. A virtual machine image from which a container is created.
 - b. A container blueprint from which a container is created.
 - c. A runtime environment where an application will run.
 - d. The container's index file used by a registry.

- ▶ 3. **Which three of the following components are common across container architecture implementations? (Choose three.)**
 - a. Container runtime
 - b. Container permissions
 - c. Container images
 - d. Container registries

- ▶ 4. **What is a container in relation to the Linux kernel?**
 - a. A virtual machine.
 - b. An isolated process with regulated resource access.
 - c. A set of file-system layers exposed by UnionFS.
 - d. An external service providing container images.

► Solution

Overview of Container Architecture

Choose the correct answers to the following questions:

- ▶ 1. **Which three of the following Linux features are used for running containers? (Choose three.)**
 - a. Namespaces
 - b. Integrity Management
 - c. Security-Enhanced Linux
 - d. Control Groups

- ▶ 2. **Which of the following best describes a container image?**
 - a. A virtual machine image from which a container is created.
 - b. A container blueprint from which a container is created.
 - c. A runtime environment where an application will run.
 - d. The container's index file used by a registry.

- ▶ 3. **Which three of the following components are common across container architecture implementations? (Choose three.)**
 - a. Container runtime
 - b. Container permissions
 - c. Container images
 - d. Container registries

- ▶ 4. **What is a container in relation to the Linux kernel?**
 - a. A virtual machine.
 - b. An isolated process with regulated resource access.
 - c. A set of file-system layers exposed by UnionFS.
 - d. An external service providing container images.

Overview of Kubernetes and OpenShift

Objectives

After completing this section, students should be able to:

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

Limitations of Containers

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

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

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

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

Kubernetes Overview

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

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

Kubernetes Features

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

Service discovery and load balancing

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

Horizontal scaling

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

Self-healing

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

Automated rollout

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

Secrets and configuration management

You can manage the configuration settings and secrets of your applications without rebuilding containers. Application secrets can be user names, passwords, and service endpoints, or any configuration setting that must be kept private.

Operators

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

OpenShift Overview

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

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



Note

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

OpenShift Features

OpenShift adds the following features to a Kubernetes cluster:

Integrated developer workflow

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

Routes

Easily expose services to the outside world.

Metrics and logging

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

Unified UI

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



References

Production-Grade Container Orchestration - Kubernetes

<https://kubernetes.io/>

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

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

► Quiz

Describing Kubernetes and OpenShift

Choose the correct answers to the following questions:

- ▶ 1. **Which three of the following statements are correct regarding container limitations? (Choose three.)**
 - a. Containers are easily orchestrated in large numbers.
 - b. Lack of automation increases response time to problems.
 - c. Containers do not manage application failures inside them.
 - d. Containers are not load-balanced.
 - e. Containers are heavily-isolated, packaged applications.
- ▶ 2. **Which two of the following statements are correct regarding Kubernetes? (Choose two.)**
 - a. Kubernetes is a container.
 - b. Kubernetes can only use Docker containers.
 - c. Kubernetes is a container orchestration system.
 - d. Kubernetes simplifies management, deployment, and scaling of containerized applications.
 - e. Applications managed in a Kubernetes cluster are harder to maintain.
- ▶ 3. **Which three of the following statements are true regarding Red Hat OpenShift v4? (Choose three.)**
 - a. OpenShift provides additional features to a Kubernetes infrastructure.
 - b. Kubernetes and OpenShift are mutually exclusive.
 - c. OpenShift hosts use Red Hat Enterprise Linux as the base operating system.
 - d. OpenShift simplifies development incorporating a Source-to-Image technology and CI/CD pipelines.
 - e. OpenShift simplifies routing and load balancing.
- ▶ 4. **What features does OpenShift offer that extend Kubernetes capabilities? (Choose two.)**
 - a. Operators and the Operator Framework.
 - b. Routes to expose services to the outside world.
 - c. An integrated development workflow.
 - d. Self-healing and health checks.

► Solution

Describing Kubernetes and OpenShift

Choose the correct answers to the following questions:

- ▶ 1. **Which three of the following statements are correct regarding container limitations? (Choose three.)**
 - a. Containers are easily orchestrated in large numbers.
 - b. Lack of automation increases response time to problems.
 - c. Containers do not manage application failures inside them.
 - d. Containers are not load-balanced.
 - e. Containers are heavily-isolated, packaged applications.

- ▶ 2. **Which two of the following statements are correct regarding Kubernetes? (Choose two.)**
 - a. Kubernetes is a container.
 - b. Kubernetes can only use Docker containers.
 - c. Kubernetes is a container orchestration system.
 - d. Kubernetes simplifies management, deployment, and scaling of containerized applications.
 - e. Applications managed in a Kubernetes cluster are harder to maintain.

- ▶ 3. **Which three of the following statements are true regarding Red Hat OpenShift v4? (Choose three.)**
 - a. OpenShift provides additional features to a Kubernetes infrastructure.
 - b. Kubernetes and OpenShift are mutually exclusive.
 - c. OpenShift hosts use Red Hat Enterprise Linux as the base operating system.
 - d. OpenShift simplifies development incorporating a Source-to-Image technology and CI/CD pipelines.
 - e. OpenShift simplifies routing and load balancing.

- ▶ 4. **What features does OpenShift offer that extend Kubernetes capabilities? (Choose two.)**
 - a. Operators and the Operator Framework.
 - b. Routes to expose services to the outside world.
 - c. An integrated development workflow.
 - d. Self-healing and health checks.

► 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 the workstation machine 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 the workstation machine.

Before You Begin

To perform this exercise, ensure you have:

- Access to the DO180 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, GitHub access token.

Instructions

Before starting any exercise, ensure you have:

- 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.

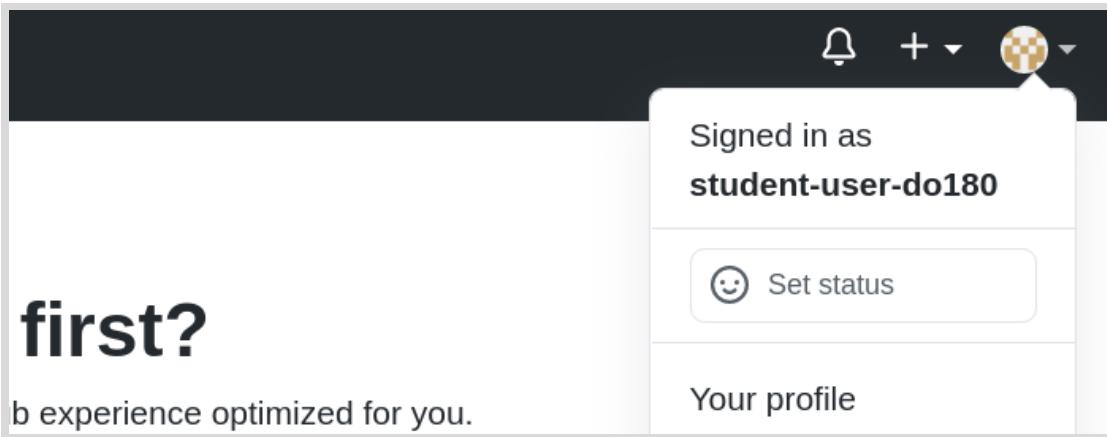


Figure 1.2: User menu

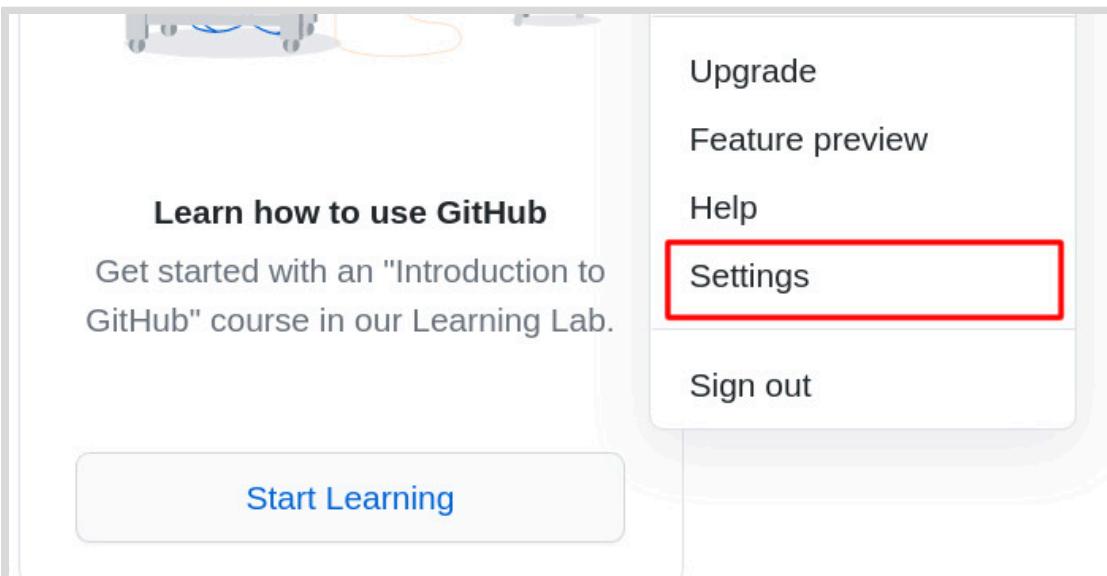


Figure 1.3: Settings menu

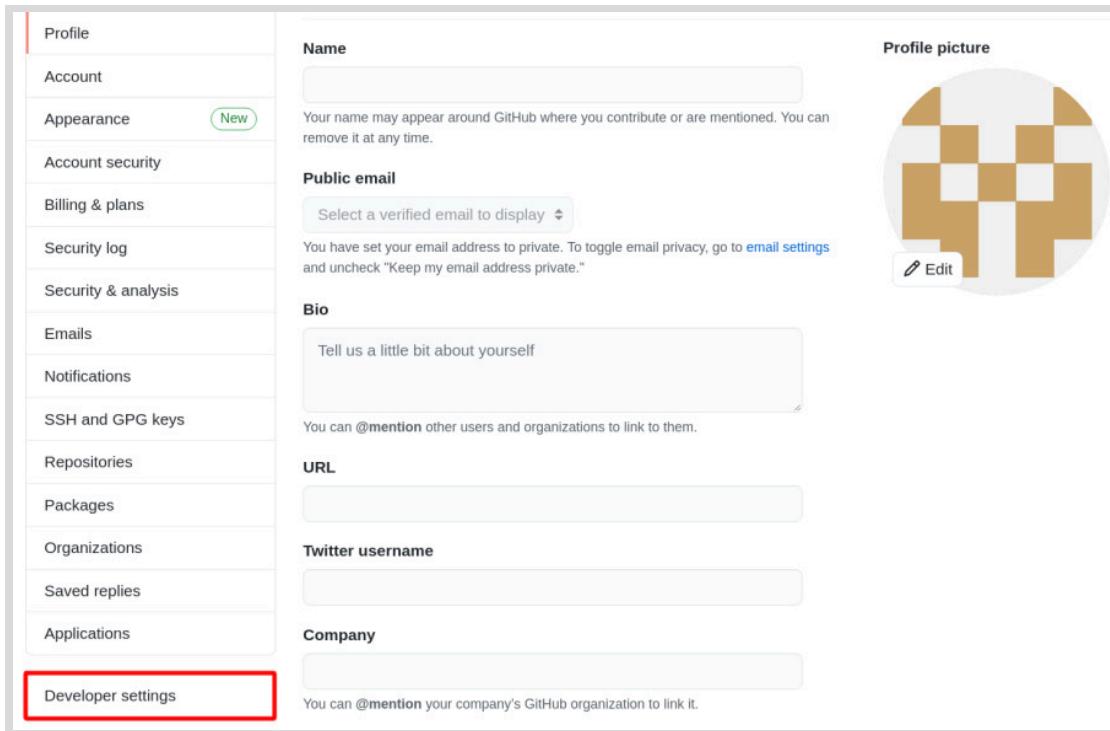


Figure 1.4: Developer settings

- 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.

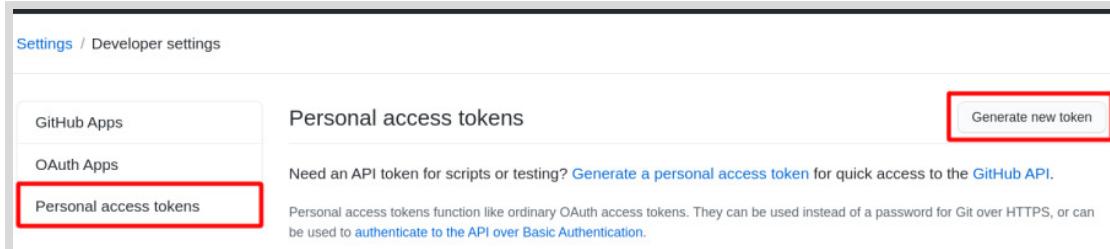


Figure 1.5: Personal access token pane

- 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 DO180
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 1.6: 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. The personal access token can not be displayed again in GitHub.

Personal access tokens

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 1.7: Generated access token

- 1.7. On workstation execute the `git config` command with the `credential.helper cache` parameters to store in cache memory your credentials for future use. The `--global` parameter applies the configuration to all of your repositories.

```
[student@workstation ~]$ git config --global credential.helper cache
```



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. Prepare your Quay.io password.

- 2.1. Configure a password for your Quay.io account. On the *Account Settings* page, click the *Change password* link. See *Appendix B, Creating a Quay Account* for further details.

► 3. Configure the workstation machine.

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

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

| Machine Type | Status | Action | Open Console |
|--------------|--------|--------|--------------|
| workstation | active | ACTION | OPEN CONSOLE |
| classroom | active | ACTION | OPEN CONSOLE |

Open a terminal on the **workstation** machine and execute the following command. Answer the interactive prompts 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
```

- 3.1. The **lab-configure** command starts by displaying a series of interactive prompts and uses sensible defaults when they are available.

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

Chapter 1 | Introducing Container Technology

```
· 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 the 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.

- 3.2. The `lab-configure` command prints all the information that you entered and attempts 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...
```

- 3.3. 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 API Endpoint...

ERROR:

```
Cannot connect to an OpenShift 4.6 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.
```

- 3.4. If everything is OK so far, the `lab-configure` attempts to access your public GitHub and Quay.io accounts.

```
...output omitted...

Verifying your GitHub account name...

Verifying your Quay.io account name...

...output omitted...
```

- 3.5. The `lab-configure` displays an error message and exits if it finds any issues. You must 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.
```

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

```
...output omitted...

Verifying your cluster configuration...

...output omitted...
```

- 3.7. If all checks pass, the `lab-configure` command saves your configuration:

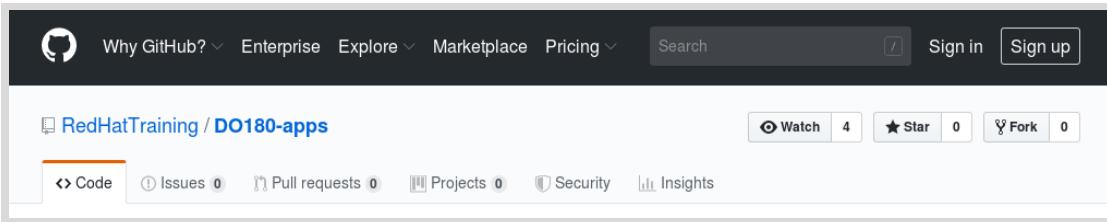
```
...output omitted...

Saving your lab configuration file...

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

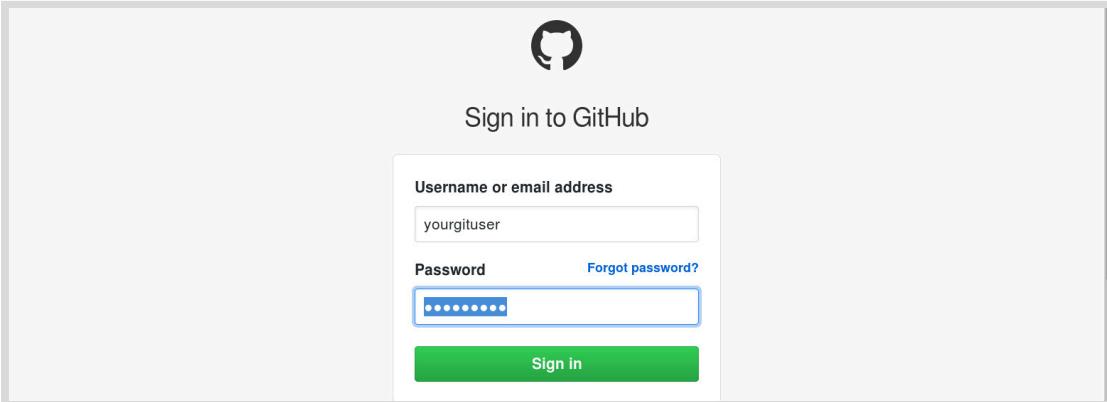
If you need to modify the configuration, rerun this or directly modify the values
in /usr/local/etc/ocp4.config.
```

- 3.8. 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.
- 4. Fork this course's sample applications into your personal GitHub account. Perform the following steps:
- 4.1. Open a web browser and navigate to <https://github.com/RedHatTraining/DO180-apps>. If you are not logged in to GitHub, click Sign in in the upper-right corner.



The screenshot shows the top navigation bar of GitHub. It includes links for "Why GitHub?", "Enterprise", "Explore", "Marketplace", "Pricing", a "Search" bar, and buttons for "Sign in" and "Sign up". Below the navigation bar, the repository "RedHatTraining / DO180-apps" is displayed. The repository card shows "Code", "Issues 0", "Pull requests 0", "Projects 0", "Security", and "Insights" sections. At the top right of the repository card, there are buttons for "Watch 4", "Star 0", "Fork 0", and "Insights".

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



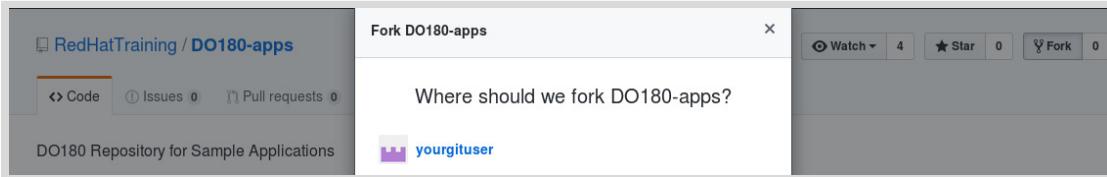
The screenshot shows the "Sign in to GitHub" page. It features a large GitHub logo at the top, followed by a form with fields for "Username or email address" (containing "yourgituser") and "Password" (redacted). There is a "Forgot password?" link and a green "Sign in" button.

- 4.3. Navigate to the RedHatTraining/D0180-apps repository and click Fork in the top right corner.



The screenshot shows the repository page for "RedHatTraining / DO180-apps". The page includes a header with "Code", "Issues 0", "Pull requests 0", "Actions", "Wiki", "Security", and "More". At the top right, there are buttons for "Watch 4", "Star 0", "Fork 0", and "Insights".

- 4.4. In the Fork D0180-apps window, click yourgituser to select your personal GitHub project.



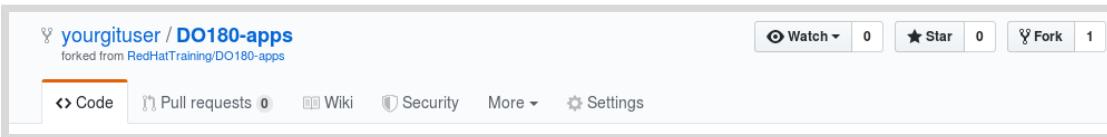
The screenshot shows a modal dialog box titled "Fork DO180-apps". It asks "Where should we fork DO180-apps?". A dropdown menu is open, showing the option "yourgituser". The background of the main GitHub interface is visible.



Important

While it is possible to rename your personal fork of the <https://github.com/RedHatTraining/DO180-apps> repository, grading scripts, helper scripts, and the example output in this course assume that you retain the name D0180-apps when your fork the repository.

- 4.5. After a few minutes, the GitHub web interface displays your new repository `yourgituser/DO180-apps`.



The screenshot shows the repository page for "yourgituser / DO180-apps". The page includes a header with "Code", "Pull requests 0", "Wiki", "Security", "More", and "Settings". At the top right, there are buttons for "Watch 0", "Star 0", "Fork 1", and "Insights". Below the header, it says "forked from RedHatTraining/DO180-apps".

- 5. Clone this course's sample applications from your personal GitHub account to your workstation machine. Perform the following steps:

- 5.1. 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/_D0180-apps
Cloning into 'D0180-apps'...
...output omitted...
```

- 5.2. Verify that /home/user/D0180-apps is a Git repository.

```
[student@workstation ~]$ cd D0180-apps
[student@workstation D0180-apps]$ git status
# On branch master
nothing to commit, working directory clean
```

- 5.3. Create a new branch to test your new personal access token.

```
[student@workstation D0180-apps]$ git checkout -b testbranch
Switched to a new branch `testbranch`
```

- 5.4. Make a change to the TEST file and then commit it to Git.

```
[student@workstation D0180-apps]$ echo "D0180" > TEST
[student@workstation D0180-apps]$ git add .
[student@workstation D0180-apps]$ git commit -am "D0180"
...output omitted...
```

- 5.5. Push the changes to your recently created testing branch.

```
[student@workstation D0180-apps]$ git push --set-upstream origin testbranch
Username for `https://github.com`: ①
Password for `https://_yourgituser_@github.com`: ②
...output omitted...
```

- ① Enter your GitHub username
- ② Enter your personal access token

- 5.6. Make other change to a text file, commit it and push it. You will notice you are no longer asked to put your user and password. This is because the `git config` command you ran in step 1.7.

```
[student@workstation D0180-apps]$ echo "OCP4.6" > TEST
[student@workstation D0180-apps]$ git add .
[student@workstation D0180-apps]$ git commit -am "OCP4.6"
[student@workstation D0180-apps]$ git push
...output omitted...
```

- 5.7. Verify that /home/user/D0180-apps contains this course's sample applications, and change back to the user's home folder.

```
[student@workstation D0180-apps]$ head README.md
# D0180-apps
...output omitted...
[student@workstation D0180-apps]$ cd ~
[student@workstation ~]$
```

Now that you have a local clone of the D0180-apps repository on the workstation machine, and you have executed the `lab-configure` command successfully, you are ready to start this course's exercises.

During this course, all exercises that build applications from source start from the `master` branch of the D0180-apps Git repository. Exercises that make changes to source code require you to create new branches to host your changes so that the `master` branch always contains a known good starting point. If for some reason you need to pause or restart an exercise and need to either save or discard changes you make into your Git branches, refer to *Appendix D, Useful Git Commands*.

This concludes the guided exercise.

Summary

In this chapter, you learned:

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

Chapter 2

Creating Containerized Services

Goal

Provision a service using container technology.

Objectives

- Create a database server from a container image.

Sections

- Provisioning Containerized Services
- Creating a MySQL Database Instance

Provisioning Containerized Services

Objectives

After completing this section, students should be able to:

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

Fetching Container Images with Podman

Applications can run inside containers, providing an isolated and controlled execution environment. Running a containerized application, that is, running an application inside a container, requires a container image and a file system bundle that provides all the application files, libraries, and dependencies that the application needs to run. Container images are available from image registries that allow users to search and retrieve container images. Podman users can use the `search` subcommand to find available images from remote or local registries.

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

After finding an image, you can use Podman to download it. Use the `pull` subcommand to direct Podman to fetch the image and save it locally for future use.

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

Container images are named based on the following syntax:

```
registry_name/user_name/image_name:tag
```

Registry Naming syntax:

- The `registry_name` is the name of the registry storing the image. It is usually the FQDN of the registry.

- The `user_name` is the name of the user or organization to which the image belongs.
- The `image_name` must be unique in user namespace.
- The `tag` identifies the image version. If the image name includes no image tag, `latest` is assumed.

**Note**

This classroom's Podman installation uses several publicly available registries, like Quay.io and Red Hat Container Catalog.

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

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

Running Containers

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

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

```
[user@demo ~]$ podman run ubi8/ubi:8.3 echo 'Hello world!'
Hello world!
```

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

```
[user@demo ~]$ podman run -d -p 8080 registry.redhat.io/rhel8/httpd-24
ff4ec6d74e9b2a7b55c49f138e56f8bc46fe2a09c23093664fea7febcd1b2
[user@demo ~]$ podman port -l
8080/tcp -> 0.0.0.0:44389
[user@demo ~]$ curl http://0.0.0.0:44389
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
...output omitted...
```

This example runs a containerized Apache HTTP server in the background. It uses the `-p 8080` option to bind HTTP server's port to a local port. Then, it uses the `podman port` command to retrieve the local port on which the container listens. Finally, it uses this port to create the target URL and fetch the root page from the Apache HTTP server. This response proves the container is still up and running after the `podman run` command.

**Note**

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

**Note**

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

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

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

**Note**

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

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

See the Podman documentation for the complete list of flags.

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

```
[user@demo ~]$ podman run -it ubi8/ubi:8.3 /bin/bash
bash-4.2# ls
...output omitted...
bash-4.2# whoami
root
bash-4.2# exit
exit
[user@demo ~]$
```

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

```
[user@demo ~]$ podman run -e GREET=Hello -e NAME=RedHat \
> ubi8/ubi:8.3 printenv GREET NAME
Hello
RedHat
[user@demo ~]$
```

The previous example starts a UBI image container that prints the two environment variables provided as parameters.

Another use case for environment variables is setting up credentials into a MySQL database server.

```
[user@demo ~]$ podman run --name mysql-custom \
> -e MYSQL_USER=redhat -e MYSQL_PASSWORD=r3dh4t \
> -e MYSQL_ROOT_PASSWORD=r3dh4t \
> -d registry.redhat.io/rhel8/mysql-80
```

Using the Red Hat Container Catalog

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

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

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

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

The screenshot shows the Red Hat Container Catalog search results for "Apache httpd". The search bar at the top contains "Apache httpd". On the left, there are filters for Provider (IBM, Red Hat, Inc.), Category (Database & Data Management, Programming Languages & Runtimes, Web Services), and Product. The results section displays three items:

- rhel8/httpd-24-rhel7** Apache httpd 2.4 by Red Hat, Inc. Platform for running Apache httpd 2.4 or building httpd-based application. Updated 11 hours ago.
- rhel8/httpd-24** Apache httpd 2.4 by Red Hat, Inc. Platform for running Apache httpd 2.4 or building httpd-based application. Updated 11 hours ago.
- ibm/couchdb3** Apache CouchDB by IBM Apache CouchDB is a database that uses JSON for documents, an HTTP API, & JavaScript/...

A "Have feedback?" button is located at the bottom right of the results area.

Figure 2.1: Red Hat Container Catalog search page

As displayed in the preceding image, searching for Apache `httpd` in the search box of the Container Catalog displays a suggested list of products and image repositories matching the search pattern. To access the Apache `httpd 2.4` image page, select `rhel8/httpd-24` from the suggested list.

After selecting the desired image, the subsequent page provides additional information about the image:

The screenshot shows the detailed view for the Apache httpd 2.4 image. At the top, it says "Standalone Image" and "Apache httpd 2.4" with the tag "rhel8/httpd-24". It includes dropdowns for "Architecture" (amd64) and "Tag" (latest). A "Provided by" section features the Red Hat logo. Below this, there's a "latest" tag with a count of 1 and a timestamp of 1-1231614609239. The page has tabs for "Overview", "Security", "Packages", "Dockerfile", and "Get this image". The "Description" tab is selected, containing a paragraph about Apache HTTP Server 2.4. To the right, there are sections for "Published" (28 days ago), "Release category" (Generally Available), "Health index" (B 0 2), and a "Have feedback?" button.

Figure 2.2: Apache httpd 2.4 (rhel8/httpd-24) overview image page

The *Apache httpd 2.4* panel displays image details and several tabs. This page states that Red Hat maintains the image repository.

Under the *Overview* tab, there are other details:

- *Description*: A summary of the image's capabilities.
- *Documentation*: References to container's author documentation.
- *Products using this container*: It indicates that Red Hat Enterprise Linux uses this image repository.

On the right side, it shows information about when the image received its latest update, the latest tag applied to the image, its health, size, and more.

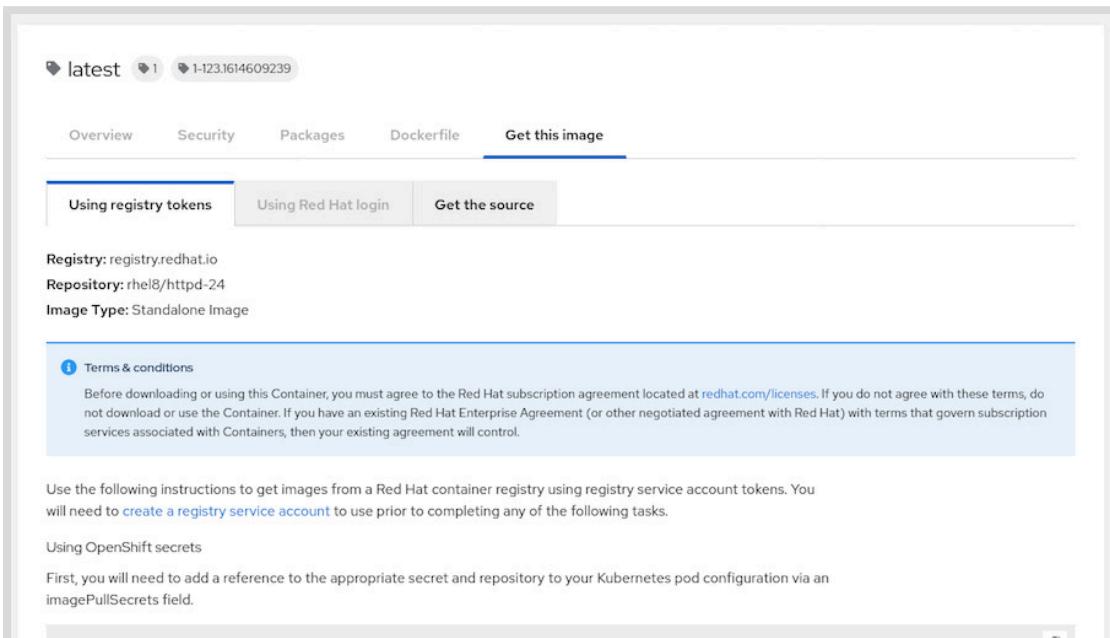


Figure 2.3: Apache httpd 2.4 (rhel8/httpd-24) latest image page

The *Get this image* tab provides the procedure to get the most current version of the image. The page provides different options to retrieve the image. Choose your preferred procedure in the tabs, and the page provides the appropriate instructions to retrieve the image.

 **References**

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

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

► Guided Exercise

Creating a MySQL Database Instance

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

Outcomes

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

Before You Begin

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

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

Instructions

► 1. Create a MySQL container instance.

- 1.1. Log in to the Red Hat Container Catalog with your Red Hat account. If you need to register with Red Hat, see the instructions in *Appendix C, Creating a Red Hat Account*.

```
[student@workstation ~]$ podman login registry.redhat.io
Username: your_username
Password: your_password
Login Succeeded!
```

- 1.2. Start a container from the Red Hat Container Catalog MySQL image.

```
[student@workstation ~]$ podman run --name mysql-basic \
> -e MYSQL_USER=user1 -e MYSQL_PASSWORD=mypa55 \
> -e MYSQL_DATABASE=items -e MYSQL_ROOT_PASSWORD=r00tpa55 \
> -d registry.redhat.io/rhel8/mysql-80:1
Trying to pull ...output omitted...
Copying blob ...output omitted...
Writing manifest to image destination
Storing signatures
`2d37682eb33a`70330259d6798bdfdc37921367f56b9c2a97339d84faa3446a03
```

This command downloads the MySQL 8.0 container image with the `1` tag, and then starts a container based on that image. It creates a database named `items`, owned by a user named `user1` with `mypa55` as the password. The database administrator password is set to `r00tpa55` and the container runs in the background.

- 1.3. Verify that the container started without errors.

```
[student@workstation ~]$ podman ps --format "{{.ID}} {{.Image}} {{.Names}}"
2d37682eb33a registry.redhat.io/rhel8/mysql-80:1 mysql-basic
```

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

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

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

- 3. Add data to the database.

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

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

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

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

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

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

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

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

```
mysql> CREATE TABLE Projects (id int NOT NULL,
-> name varchar(255) DEFAULT NULL,
-> code varchar(255) DEFAULT NULL,
-> PRIMARY KEY (id));
Query OK, 0 rows affected (0.01 sec)
```

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

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

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

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

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

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

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

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

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

Finish

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

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

This concludes the exercise.

Summary

In this chapter, you learned:

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

Chapter 3

Managing Containers

Goal

Make use of prebuilt container images to create and manage containerized services.

Objectives

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

Sections

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

Lab

- Managing Containers

Managing the Life Cycle of Containers

Objectives

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

Container Life Cycle Management with Podman

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

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

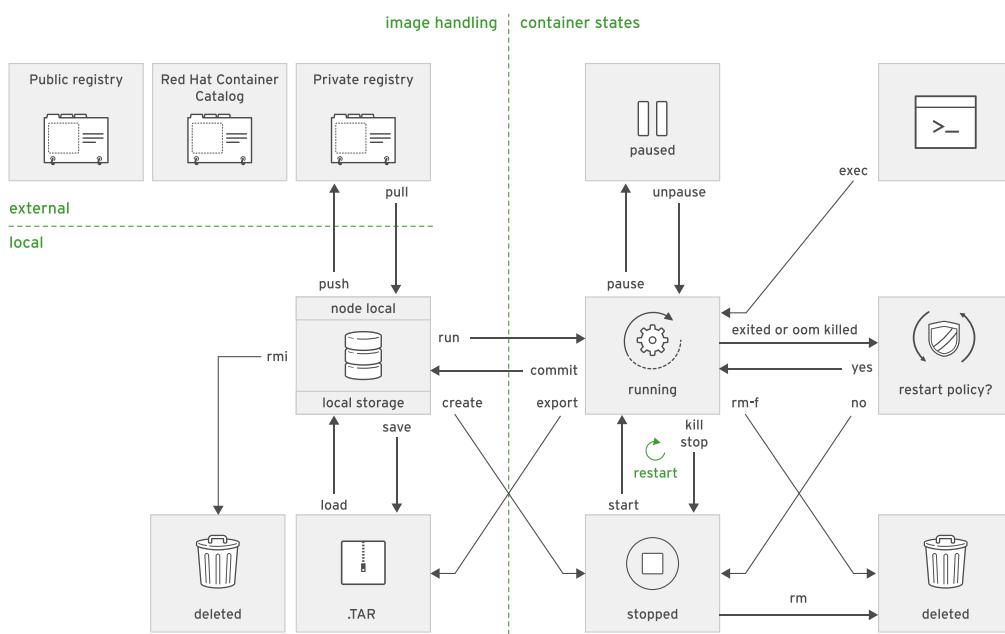


Figure 3.1: Podman managing subcommands

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

You can use these subcommands to extract information from containers and images for debugging, updating, or reporting purposes. The following figure shows a summary of the most commonly used subcommands that query information from containers and images:

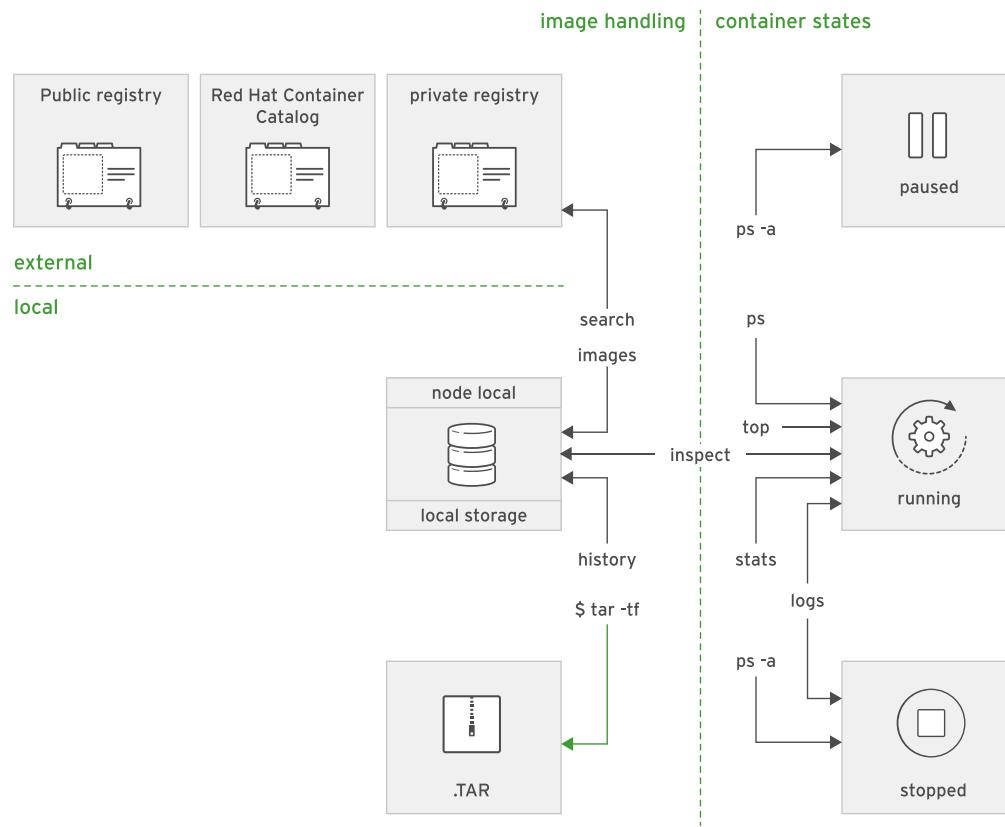


Figure 3.2: Podman query subcommands

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

Creating Containers

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

```
[user@host ~]$ podman run registry.redhat.io/rhel8/httpd-24
Trying to pull registry.redhat.io/rhel8/httpd-24...
Getting image source signatures
Copying blob sha256:23113...b0be82
72.21 MB / 72.21 MB [=====] 7s
...output omitted...AH00094: Command line: 'httpd -D FOREGROUND'
^C
```

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

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

```
[user@host ~]$ podman ps
CONTAINER ID IMAGE COMMAND ... NAMES
47c9aad6049①
registry.redhat.io/rhel8/httpd-24 "/usr/bin/run-http..." ... focused_fermat②
```

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

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

```
[user@host ~]$ podman run --name my-httpd-container \
> registry.redhat.io/rhel8/httpd-24
...output omitted...AH00094: Command line: 'httpd -D FOREGROUND'
```



Note

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

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

```
[user@host ~]$ podman run --name my-httpd-container \
> -d registry.redhat.io/rhel8/httpd-24
77d4b7b8ed1fd57449163bcb0b78d205e70d2314273263ab941c0c371ad56412
```

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

```
[user@host ~]$ podman run registry.redhat.io/rhel8/httpd-24 ls /tmp
ks-script-1j4CXN
```

The specified command must be executable inside the container image.



Note

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

Some containers need to run as an interactive shell or process. This includes containers running processes that need user input (such as entering commands), and processes that generate output through standard output. The following example starts an interactive bash shell in a `registry.redhat.io/rhel8/httpd-24` container:

```
[user@host ~]$ podman run -it registry.redhat.io/rhel8/httpd-24 /bin/bash  
bash-4.4#
```

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

Running Commands in a Container

When a container starts, it executes the entry point command. However, it may be necessary to execute other commands to manage the running container.

Some typical use case are shown below:

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

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

```
[user@host ~]$ podman exec 7ed6e671a600 cat /etc/hostname  
7ed6e671a600
```

In this example, the container ID is used to execute a command in an existing container.

Podman remembers the last container created. Developers can skip writing this container's ID or name in later Podman commands by replacing the container id by the `-l` (or `--latest`) option:

```
[user@host ~]$ podman exec my-httpd-container cat /etc/hostname  
7ed6e671a600  
[user@host ~]$ podman exec -l cat /etc/hostname  
7ed6e671a600
```

Managing Containers

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

Podman provides the following commands for managing containers:

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

```
[user@host ~]$ podman ps  
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES  
77d4b7b8ed1f registry.redhat.io/rhel8/httpd-24 "/usr/bin/run-http..." ...ago  
Up... my-htt...❶
```

- ❶ Each row describes information about the container.

Chapter 3 | Managing Containers

Each container, when created, gets a `container ID`, which is a hexadecimal number. This ID looks like an image ID but is unrelated.

The `IMAGE` field indicates container image that was used to start the container.

The `COMMAND` field indicates command executed when the container started.

The `CREATED` field indicates date and time the container was started.

The `STATUS` field indicates total container uptime, if still running, or time since terminated.

The `PORTS` field indicates ports that were exposed by the container or any port forwarding that might be configured.

The `NAMES` field indicates the container name.

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

```
[user@host ~]$ podman ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
4829d82fbbff registry.redhat.io/rhel8/httpd-24 "/usr/bin/run-httpd..." ...ago
    Exited (0)... my-httpd...
```



Note

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

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

```
[user@host ~]$ podman stop my-httpd-container
77d4b7b8ed1fd57449163bcb0b78d205e70d2314273263ab941c0c371ad56412
```

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

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

```
[user@host ~]$ podman kill my-httpd-container
77d4b7b8ed1fd57449163bcb0b78d205e70d2314273263ab941c0c371ad56412
```

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

```
[user@host ~]$ podman kill -s SIGKILL my-httpd-container
77d4b7b8ed1fd57449163bcb0b78d205e70d2314273263ab941c0c371ad56412
```

Any Unix signal can be sent to the main process. Podman accepts either the signal name or number.

The following table shows several useful signals:

| Signal | Value | Default Action | Comment |
|---------|----------|----------------|---|
| SIGHUP | 1 | Term | Hangup detected on controlling terminal or death of controlling process |
| SIGINT | 2 | Term | Interrupt from keyboard |
| SIGQUIT | 3 | Core | Quit from keyboard |
| SIGILL | 4 | Core | Illegal Instruction |
| SIGABRT | 6 | Core | Abort signal from abort(3) |
| SIGFPE | 8 | Core | Floating point exception |
| SIGKILL | 9 | Term | Kill signal |
| SIGSEGV | 11 | Core | Invalid memory reference |
| SIGPIPE | 13 | Term | Broken pipe: write to pipe with no readers |
| SIGALRM | 14 | Term | Timer signal from alarm(2) |
| SIGTERM | 15 | Term | Termination signal |
| SIGUSR1 | 30,10,16 | Term | User-defined signal 1 |
| SIGUSR2 | 31,12,17 | Term | User-defined signal 2 |
| SIGCHLD | 20,17,18 | Ign | Child stopped or terminated |
| SIGCONT | 19,18,25 | Cont | Continue if stopped |
| SIGSTOP | 17,19,23 | Stop | Stop process |
| SIGTSTP | 18,20,24 | Stop | Stop typed at tty |
| SIGTTIN | 21,21,26 | Stop | tty input for background process |
| SIGTTOU | 22,22,27 | Stop | tty output for background process |

Any Unix signal can be sent to the main process. Podman accepts either the signal name or number.



Note
Term

Terminate the process.

Core

Terminate the process and generate a core dump.

Ign

Signal is ignored.

Stop

Stop the process.

Additional useful podman commands include the following:

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

```
[user@host ~]$ podman restart my-httdp-container
77d4b7b8ed1fd57449163bcb0b78d205e70d2314273263ab941c0c371ad56412
```

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

- The **podman rm** command deletes a container and discards its state and file system.

```
[user@host ~]$ podman rm my-httdp-container
77d4b7b8ed1fd57449163bcb0b78d205e70d2314273263ab941c0c371ad56412
```

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

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

```
[user@host ~]$ podman rm -a
5fd8e98ec7eab567eabe84943fe82e99fdfc91d12c65d99ec760d5a55b8470d6
716fd687f65b0957edac73b84b3253760e915166d3bc620c4aec8e5f4eadfe8e
86162c906b44f4cb63ba2e3386554030dc6abedbce9e9fcad60aa9f8b2d5d4
```

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

```
[user@host ~]$ podman stop -a
5fd8e98ec7eab567eabe84943fe82e99fdfc91d12c65d99ec760d5a55b8470d6
716fd687f65b0957edac73b84b3253760e915166d3bc620c4aec8e5f4eadfe8e
86162c906b44f4cb63ba2e3386554030dc6abedbce9e9fcad60aa9f8b2d5d4
```



Note

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



References

Unix Posix Signals man page

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

► Guided Exercise

Managing a MySQL Container

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

Outcomes

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

Before You Begin

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

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

Instructions

- 1. Download the MySQL database container image and attempt to start it. The container does not start because several environment variables must be provided to the image.
 - 1.1. Log in to the Red Hat Container Catalog with your Red Hat account. If you need to register with Red Hat, see the instructions in *Appendix C, Creating a Red Hat Account*.

```
[student@workstation ~]$ podman login registry.redhat.io
Username: your_username
Password: your_password
Login Succeeded!
```

- 1.2. Download the MySQL database container image and attempt to start it.

```
[student@workstation ~]$ podman run --name mysql-db \
> registry.redhat.io/rhel8/mysql-80:1
Trying to pull ...output omitted...
...output omitted...
Writing manifest to image destination
Storing signatures
You must either specify the following environment variables:
  MYSQL_USER (regex: '^$')
  MYSQL_PASSWORD (regex: '^[a-zA-Z0-9_~!@#$%^&*()-=<>,.?;:|]$')
  MYSQL_DATABASE (regex: '^$')
`Or the following environment variable:
  MYSQL_ROOT_PASSWORD (regex: '^[a-zA-Z0-9_~!@#$%^&*()-=<>,.?;:|]$')
Or both.
Optional Settings:
...output omitted...

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

**Note**

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

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

- ▶ 2. Create a new container named `mysql`, and then specify each required variable using the `-e` parameter.

**Note**

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

```
[student@workstation ~]$ podman run --name mysql \
> -d -e MYSQL_USER=user1 -e MYSQL_PASSWORD=mypa55 \
> -e MYSQL_DATABASE=items -e MYSQL_ROOT_PASSWORD=r00tpa55 \
> registry.redhat.io/rhel8/mysql-80:1
```

The command displays the container ID for the `mysql` container. Below is an example of the output.

```
a49dba9ff17f2b5876001725b581fdd331c9ab8b9eda21cc2a2899c23f078509
```

- ▶ 3. Verify that the `mysql` container started correctly. Run the following command:

```
[student@workstation ~]$ podman ps --format="{{.ID}} {{.Names}} {{.Status}}"
a49dba9ff17f    mysql    Up About a minute ago
```

The command only shows the first 12 characters of the container ID displayed in the previous command.

- ▶ 4. Populate the `items` database with the `Projects` table:

- 4.1. Run `podman cp` command to copy database file to `mysql` container.

```
[student@workstation ~]$ podman cp \
> /home/student/D0180/labs/manage-lifecycle/db.sql mysql:/
```

- 4.2. Populate the `items` database with the `Projects` table.

```
[student@workstation ~]$ podman exec mysql /bin/bash \
> -c 'mysql -uuser1 -pmypa55 items < /db.sql'
mysql: [Warning] Using a password on the command line interface can be insecure.
```

- ▶ 5. Create another container using the same container image as the previous container. Interactively enter the /bin/bash shell instead of using the default command for the container image.

```
[student@workstation ~]$ podman run --name mysql-2 \
> -it registry.redhat.io/rhel8/mysql-80:1 /bin/bash
bash-4.4$
```

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

```
bash-4.4$ mysql -uroot
```

The following error displays:

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

The MySQL database server is not running because the container executed the /bin/bash command rather than starting the MySQL server.

- ▶ 7. Exit the container.

```
bash-4.4$ exit
```

- ▶ 8. Verify that the mysql-2 container is not running.

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

- ▶ 9. Query the mysql container to list all rows in the Projects table. The command instructs the bash shell to query the items database using a mysql command.

```
[student@workstation ~]$ podman exec mysql /bin/bash \
> -c 'mysql -uuser1 -pmypa55 -e "select * from items.Projects;"'
mysql: [Warning] Using a password on the command-line interface can be insecure.
+----+-----+-----+
| id | name | code |
+----+-----+-----+
| 1 | DevOps | D0180 |
```

Finish

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

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

This concludes the exercise.

Attaching Persistent Storage to Containers

Objectives

After completing this section, students should be able to:

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

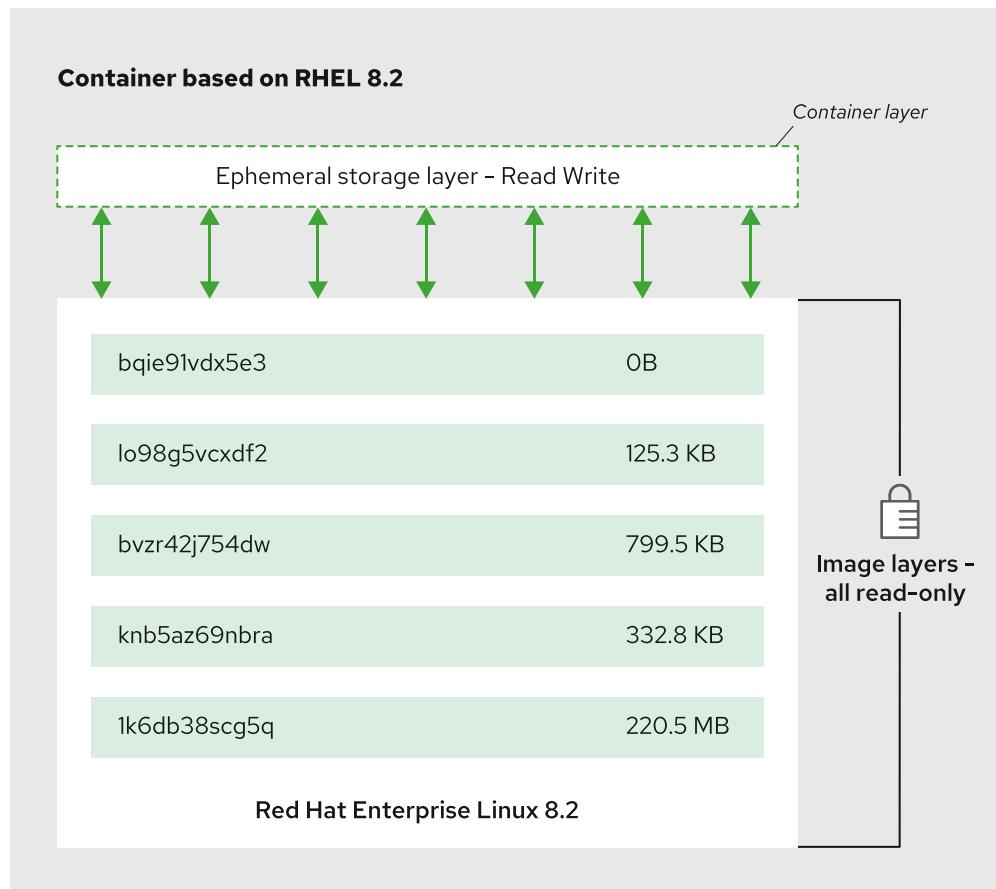
Preparing Permanent Storage Locations

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

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

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

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

**Figure 3.3: Container layers**

Containerized applications should not try to use the container storage to store persistent data, because they cannot control how long its contents will be preserved.

Even if it were possible to keep container storage indefinitely, the layered file system does not perform well for intensive I/O workloads and would not be adequate for most applications requiring persistent storage.

Reclaiming Storage

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

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

Preparing the Host Directory

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

For example, a database container can use a host directory to store database files. If this database container fails, Podman can create a new container using the same host directory, keeping the

database data available to client applications. To the database container, it does not matter where this host directory is stored from the host point of view; it could be anywhere from a local hard disk partition to a remote networked file system.

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

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

1. Create a directory:

```
[user@host ~]$ mkdir /home/student/dbfiles
```

2. The user running processes in the container must be capable of writing files to the directory. The permission should be defined with the numeric user ID (UID) from the container. For the MySQL service provided by Red Hat, the UID is 27. The `podman unshare` command provides a session to execute commands within the same user namespace as the process running inside the container.

```
[user@host ~]$ podman unshare chown -R 27:27 /home/student/dbfiles
```

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

```
[user@host ~]$ sudo semanage fcontext -a -t container_file_t '/home/student/dbfiles(/.*)?'
```

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

```
[user@host ~]$ sudo restorecon -Rv /home/student/dbfiles
```

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

Mounting a Volume

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

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

```
[user@host ~]$ podman run -v /home/student/dbfiles:/var/lib/mysql rhmap47/mysql
```

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

► Guided Exercise

Create MySQL Container with Persistent Database

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

Outcomes

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

Before You Begin

The workstation should not have any container images running.

Run the following command on workstation:

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

Instructions

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

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

- 1.2. Add the appropriate SELinux context for the /home/student/local/mysql directory and its contents.

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

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

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

- 1.4. Verify that the SELinux context type for the /home/student/local/mysql directory is container_file_t.

```
[student@workstation ~]$ ls -ldZ /home/student/local/mysql  
drwxrwxr-x. 2 student student unconfined_u:object_r:container_file_t:s0 6 May 26  
14:33 /home/student/local/mysql
```

15. Change the owner of the /home/student/local/mysql directory to the mysql user and mysql group:

```
[student@workstation ~]$ podman unshare chown 27:27 /home/student/local/mysql
```

**Note**

The user running processes in the container must be capable of writing files to the directory.

The permission should be defined with the numeric user ID (UID) from the container. For the MySQL service provided by Red Hat, the UID is 27. The `podman unshare` command provides a session to execute commands within the same user namespace as the process running inside the container.

- ▶ 2. Create a MySQL container instance with persistent storage.

- 2.1. Log in to the Red Hat Container Catalog with your Red Hat account. If you need to register with Red Hat, see the instructions in *Appendix C, Creating a Red Hat Account*.

```
[student@workstation ~]$ podman login registry.redhat.io
Username: your_username
Password: your_password
Login Succeeded!
```

- 2.2. Pull the MySQL container image.

```
[student@workstation ~]$ podman pull registry.redhat.io/rhel8/mysql-80:1
Trying to pull ...output omitted...registry.redhat.io/rhel8/mysql-80:1...output
omitted...
...output omitted...
Writing manifest to image destination
Storing signatures
4ae3a3f4f409a8912cab9fbf71d3564d011ed2e68f926d50f88f2a3a72c809c5
```

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

```
[student@workstation ~]$ podman run --name persist-db \
> -d -v /home/student/local/mysql:/var/lib/mysql/data \
> -e MYSQL_USER=user1 -e MYSQL_PASSWORD=myspa55 \
> -e MYSQL_DATABASE=items -e MYSQL_ROOT_PASSWORD=r00tpa55 \
> registry.redhat.io/rhel8/mysql-80:1
6e0ef134315b510042ca757faf869f2ba19df27790c601f95ec2fd9d3c44b95d
```

This command mounts the /home/student/local/mysql directory from the host to the /var/lib/mysql/data directory in the container. By default, the MySQL database stores data in the /var/lib/mysql/data directory.

- 2.4. Verify that the container started correctly.

```
[student@workstation ~]$ podman ps --format="{{.ID}} {{.Names}} {{.Status}}"
6e0ef134315b  persist-db  Up 3 minutes ago
```

- 3. Verify that the /home/student/local/mysql directory contains the items directory.

```
[student@workstation ~]$ ls -ld /home/student/local/mysql/items
drwxr-x---. 2 100026 100026 6 Apr  8 07:31 /home/student/local/mysql/items
```

The items directory stores data related to the items database that was created by this container. If the items directory does not exist, then the mount point was not defined correctly during container creation.



Note

Alternatively you can run the same command with podman unshare to check numeric user ID (UID) from the container.

```
[student@workstation ~]$ podman unshare ls -ld /home/student/local/mysql/items
drwxr-x---. 2 27 27 6 Apr  8 07:31 /home/student/local/mysql/items
```

Finish

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

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

This concludes the exercise.

Accessing Containers

Objectives

After completing this section, students should be able to:

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

Mapping Network Ports

Accessing a rootless container from the host network can be challenging because no IP address is available for a rootless container.

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

Consider the following example:

```
[user@host ~]$ podman run -d --name apache1 -p 8080:80 \
> registry.redhat.io/rhel8/httpd-24
```

This example creates an externally accessible container. The value 8080 colon 80 specifies that any requests to port 8080 on the host are forwarded to port 80 within the container.

You can also use the `-p` option to bind the port to the specified IP address.

```
[user@host ~]$ podman run -d --name apache2 \
> -p 127.0.0.1:8081:80 registry.redhat.io/rhel8/httpd-24
```

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

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

```
[user@host ~]$ podman run -d --name apache3 -p 127.0.0.1::80 \
> registry.redhat.io/rhel8/httpd-24
```

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

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

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

```
[user@host ~]$ podman run -d --name apache4 \
> -p 80 registry.redhat.io/rhel8/httpd-24
[user@host ~]$ podman port apache4
80/tcp -> 0.0.0.0:37068
```

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



References

Container Network Interface - networking for Linux containers

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

Cloud Native Computing Foundation

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

► Guided Exercise

Loading the Database

In this exercise, you will create a MySQL database container with port forwarding enabled. After populating a database with a SQL script, you verify the database content using three different methods.

Outcomes

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

Before You Begin

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

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

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

Instructions

- 1. Create a MySQL container instance with persistent storage and port forwarding.
 - 1.1. Log in to the Red Hat Container Catalog with your Red Hat account. If you need to register with Red Hat, see the instructions in *Appendix C, Creating a Red Hat Account*.
- ```
[student@workstation ~]$ podman login registry.redhat.io
Username: your_username
Password: your_password
Login Succeeded!
```
- 1.2. Download the MySQL database container image and then create a MySQL container instance with persistent storage and port forwarding.

```
[student@workstation ~]$ podman run --name mysqldb-port \
> -d -v /home/student/local/mysql:/var/lib/mysql/data -p 13306:3306 \
> -e MYSQL_USER=user1 -e MYSQL_PASSWORD=mypa55 \
> -e MYSQL_DATABASE=items -e MYSQL_ROOT_PASSWORD=r00tpa55 \
> registry.redhat.io/rhel8/mysql-80:1
Trying to pull ...output omitted...
Copying blob sha256:1c9f515...output omitted...
 72.70 MB / ? [=====] 5s
Copying blob sha256:1d2c4ce...output omitted...
 1.54 KB / ? [=====] 0s
Copying blob sha256:f1e961f...output omitted...
 6.85 MB / ? [=====] 0s
Copying blob sha256:9f1840c...output omitted...
 62.31 MB / ? [=====] 7s
```

```
Copying config sha256:60726...output omitted...
6.85 KB / 6.85 KB [=====] 0s
Writing manifest to image destination
Storing signatures
066630d45cb902ab533d503c83b834aa6a9f9cf88755cb68eedb8a3e8edbc5aa
```

The last line of your output and the time needed to download each image layer will differ.

The -p option configures port forwarding so that port 13306 on the local host forwards to container port 3306.



### Note

The start script creates the /home/student/local/mysql directory with the appropriate ownership and SELinux context required by the containerized database.

- ▶ 2. Verify that the mysqlDb-port container started successfully and enables port forwarding.

```
[student@workstation ~]$ podman ps --format="{{.ID}} {{.Names}} {{.Ports}}"
9941da2936a5 mysqlDb-port 0.0.0.0:13306->3306/tcp
```

- ▶ 3. Populate the database using the provided file. If there are no errors, then the command does not return any output.

```
[student@workstation ~]$ mysql -uuser1 -h 127.0.0.1 -pmypa55 \
> -P13306 items < /home/student/D0180/labs/manage-networking/db.sql
mysql: [Warning] Using a password on the command line interface can be insecure.
```

There are multiple ways to verify that the database loaded successfully. The next steps show three different methods. You only need to use one of the methods.

- ▶ 4. Verify that the database loaded successfully by executing a non-interactive command inside the container.

```
[student@workstation ~]$ podman exec -it mysqlDb-port \
> mysql -uroot items -e "SELECT * FROM Item"

| id | description | done |

| 1 | Pick up newspaper | 0 |
| 2 | Buy groceries | 1 |

```

- 5. Verify that the database loaded successfully by using port forwarding from the local host. This alternate method is optional.

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

| id | description | done |

| 1 | Pick up newspaper | 0 |
| 2 | Buy groceries | 1 |

```

- 6. Verify that the database loaded successfully by opening an interactive terminal session inside the container. This alternate method is optional.

- 6.1. Open a Bash shell inside the container.

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

- 6.2. Verify that the database contains data:

```
bash-4.4$ mysql -uroot items -e "SELECT * FROM Item"

| id | description | done |

| 1 | Pick up newspaper | 0 |
| 2 | Buy groceries | 1 |

```

- 6.3. Exit from the container:

```
bash-4.4$ exit
```

## Finish

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

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

This concludes the exercise.

## ► Lab

# Managing Containers

## Outcomes

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

## Before You Begin

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

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

## Instructions

1. Create the `/home/student/local/mysql` directory with the correct SELinux context and permissions.
2. Deploy a MySQL container instance using the following characteristics:
  - *Name*: `mysql-1`
  - *Run as daemon*: `yes`
  - *Volume*: from `/home/student/local/mysql` host folder to `/var/lib/mysql/data` container folder
  - *Container image*: `registry.redhat.io/rhel8/mysql-80:1`
  - *Port forward*: `yes`, from host port `13306` to container port `3306`
  - *Environment variables*:
    - `MYSQL_USER`: `user1`
    - `MYSQL_PASSWORD`: `mypa55`
    - `MYSQL_DATABASE`: `items`
    - `MYSQL_ROOT_PASSWORD`: `r00tpa55`
3. Load the `items` database using the `/home/student/D0180/labs/manage-review/db.sql` script.
4. Stop the container gracefully.

**Important**

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

5. Create a new container with the following characteristics:
  - *Name*: `mysql-2`
  - *Run as a daemon*: yes
  - *Volume*: from `/home/student/local/mysql` host folder to `/var/lib/mysql/data` container folder
  - *Container image*: `registry.redhat.io/rhel8/mysql-80:1`
  - *Port forward*: yes, from host port 13306 to container port 3306
  - *Environment variables*:
    - `MYSQL_USER`: `user1`
    - `MYSQL_PASSWORD`: `mypa55`
    - `MYSQL_DATABASE`: `items`
    - `MYSQL_ROOT_PASSWORD`: `r00tpa55`
6. Save the list of all containers (including stopped ones) to the `/tmp/my-containers` file.
7. Access the Bash shell inside the container and verify that the `items` database and the `Item` table are still available. Confirm also that the table contains data.
8. Using port forwarding, insert a new row into the `Item` table. The row should have a `description` value of `Finished lab`, and a `done` value of `1`.
9. Because the first container is not required any more, remove it to release resources.

**Evaluation**

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

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

**Finish**

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

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

This concludes the lab.

## ► Solution

# Managing Containers

### Outcomes

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

### Before You Begin

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

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

### Instructions

1. Create the `/home/student/local/mysql` directory with the correct SELinux context and permissions.

- 1.1. Create the `/home/student/local/mysql` directory.

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

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

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

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

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

- 1.4. Change the owner of the `/home/student/local/mysql` directory to match the `mysql` user and `mysql` group for the `registry.redhat.io/rhel8/mysql-80:1` container image:

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

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

- `Name: mysql-1`

- *Run as daemon:* yes
- *Volume:* from /home/student/local/mysql host folder to /var/lib/mysql/data container folder
- *Container image:* registry.redhat.io/rhel8/mysql-80:1
- *Port forward:* yes, from host port 13306 to container port 3306
- *Environment variables:*
  - MYSQL\_USER: user1
  - MYSQL\_PASSWORD: mypa55
  - MYSQL\_DATABASE: items
  - MYSQL\_ROOT\_PASSWORD: r00tpa55

2.1. Log in to the Red Hat Container Catalog with your Red Hat account.

```
[student@workstation ~]$ podman login registry.redhat.io
Username: your_username
Password: your_password
Login Succeeded!
```

2.2. Create and start the container.

```
[student@workstation ~]$ podman run --name mysql-1 -p 13306:3306 \
> -d -v /home/student/local/mysql:/var/lib/mysql/data \
> -e MYSQL_USER=user1 -e MYSQL_PASSWORD=mypa55 \
> -e MYSQL_DATABASE=items -e MYSQL_ROOT_PASSWORD=r00tpa55 \
> registry.redhat.io/rhel8/mysql-80:1
Trying to pull ...output omitted...
...output omitted...
Writing manifest to image destination
Storing signatures
6l6azfaa55x866e7eb18b1fd0423a5461d8f24c147b1ad8668b76e6167587cdd
```

2.3. Verify that the container was started correctly.

```
[student@workstation ~]$ podman ps --format="{{.ID}} {{.Names}}"
6l6azfaa55x8 mysql-1
```

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

3.1. Load the database using the SQL commands in /home/student/D0180/labs/manage-review/db.sql.

```
[student@workstation ~]$ mysql -uuser1 -h 127.0.0.1 \
> -pmypa55 -P13306 items < /home/student/D0180/labs/manage-review/db.sql
mysql: [Warning] Using a password on the command-line interface can be insecure.
```

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

**Note**

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

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

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

```

4. Stop the container gracefully.

**Important**

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

- 4.1. Use the `podman` command to stop the container.

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

5. Create a new container with the following characteristics:

- *Name*: `mysql-2`
- *Run as a daemon*: yes
- *Volume*: from `/home/student/local/mysql` host folder to `/var/lib/mysql/data` container folder
- *Container image*: `registry.redhat.io/rhel8/mysql-80:1`
- *Port forward*: yes, from host port 13306 to container port 3306
- *Environment variables*:
  - `MYSQL_USER`: `user1`
  - `MYSQL_PASSWORD`: `mypa55`
  - `MYSQL_DATABASE`: `items`
  - `MYSQL_ROOT_PASSWORD`: `r00tpa55`

- 5.1. Create and start the container.

```
[student@workstation ~]$ podman run --name mysql-2 \
> -d -v /home/student/local/mysql:/var/lib/mysql/data \
> -p 13306:3306 \
> -e MYSQL_USER=user1 -e MYSQL_PASSWORD=mypa55 \
> -e MYSQL_DATABASE=items -e MYSQL_ROOT_PASSWORD=r00tpa55 \
> registry.redhat.io/rhel8/mysql-80:1
281c0e2790e54cd5a0b8e2a8cb6e3969981b85cde8ac611bf7ea98ff78bdffbb
```

5.2. Verify that the container was started correctly.

```
[student@workstation ~]$ podman ps --format="{{.ID}} {{.Names}}"
281c0e2790e5 mysql-2
```

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

6.1. Use the podman command to save the information in /tmp/my-containers.

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

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

7.1. Access the Bash shell inside the container.

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

7.2. Connect to the MySQL server.

```
bash-4.4$ mysql -uroot
```

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

```
mysql> show databases;

| Database |

| information_schema |
| items |
| mysql |
| performance_schema |
| sys |

5 rows in set (0.03 sec)
```

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

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

| Tables_in_items |
```

```

| Item |

1 row in set (0.01 sec)
```

- 7.5. View the data from the table.

```
mysql> SELECT * FROM Item;

| id | description | done |

| 1 | Pick up newspaper | 0 |
| 2 | Buy groceries | 1 |

```

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

```
mysql> exit
Bye
bash-4.4$ exit
```

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

- 8.1. Connect to the MySQL database.

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

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

mysql>
```

- 8.2. Insert the new row.

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

- 8.3. Exit from the MySQL client.

```
mysql> exit
Bye
```

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

- 9.1. Use the following command to remove the container:

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

## Evaluation

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

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

## Finish

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

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

This concludes the lab.

# Summary

---

In this chapter, you learned:

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



## Chapter 4

# Managing Container Images

### Goal

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

### Objectives

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

### Sections

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

### Lab

- Managing Container Images

# Accessing Registries

---

## Objectives

After completing this section, students should be able to:

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

## Public Registries

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

Podman searches for and downloads container images from public and private registries. Red Hat Container Catalog is the public image registry managed by Red Hat. It hosts a large set of container images, including those provided by major open source projects, such as Apache, MySQL, and Jenkins. All images in the Container Catalog are vetted by the Red Hat internal security team and all components have been rebuilt by Red Hat to avoid known security vulnerabilities.

Red Hat container images provide the following benefits:

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

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

While Red Hat Container Catalog images are trusted and verified, Quay.io offers live images regularly updated by creators. Quay.io users can create their namespaces, with fine-grained

access control, and publish the images they create to that namespace. Container Catalog users rarely or never push new images, but consume trusted images generated by the Red Hat team.

## Private Registries

Image creators or maintainers want to make their images publicly available. However, there are other image creators who prefer to keep their images private due to:

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

In some cases, private images are preferred. Private registries give image creators the control over their images placement, distribution and usage.

## Configuring Registries in Podman

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

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



### Note

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

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

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

## Accessing Registries

Podman provides commands that enable you to search for images. Images registries can also be accessed via an API. The following discusses both approaches.

### Searching for Images in Registries

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

```
[user@host ~]$ podman search [OPTIONS] <term>
```

The following table displays useful options available for the `search` subcommand:

| Option                                       | Description                                                                                                                                                                                                                                                                                                                          |
|----------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>--limit &lt;number&gt;</code>          | Limits the number of listed images per registry.                                                                                                                                                                                                                                                                                     |
| <code>--filter &lt;filter=value&gt;</code>   | Filter output based on conditions provided. Supported filters are: <code>stars=&lt;number&gt;</code> : Show only images with at least this number of stars. <code>is-automated=&lt;true false&gt;</code> : Show only images automatically built. <code>is-official=&lt;true false&gt;</code> : Show only images flagged as official. |
| <code>--tls-verify &lt;true false&gt;</code> | Enables or disables HTTPS certificate validation for all used registries.                                                                                                                                                                                                                                                            |
| <code>--list-tags</code>                     | List the available tags in the repository for the specified image.                                                                                                                                                                                                                                                                   |

## Registry HTTP API

A remote registry exposes web services that provide an application programming interface (API) to the registry. Podman uses these interfaces to access and interact with remote repositories. Many registries conform to the Docker Registry HTTP API v2 specification, which exposes a standardized REST interface for registry interaction. You can use this REST interface to directly interact with a registry, instead of using Podman.

Examples for using this API with `curl` commands are shown below:

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

```
[user@host ~]$ curl -Ls https://myserver/v2/_catalog?n=3
{"repositories": ["centos/httpd", "do180/custom-httpd", "hello-openshift"]}
```



### Note

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

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

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

```
[user@host ~]$ curl -Ls \
> https://quay.io/v2/redhattraining/httpd-parent/tags/list \
> | python -m json.tool
{
 "name": "redhattraining/httpd-parent",
 "tags": [
 "latest",
 "2.4"
]
}
```



### Note

Quay.io offers a dedicated API to interact with repositories beyond what is specified in Docker Repository API. See <https://docs.quay.io/api/> for details.

## Registry Authentication

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

```
[user@host ~]$ podman login -u username \
> -p password registry.access.redhat.com
Login Succeeded!
```

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

```
[user@host ~]$ curl -u username:password -Ls \
> "https://sso.redhat.com/auth/realms/rhcc/protocol/redhat-docker-v2/auth?
service=docker-registry"
>{"token":"eyJh...05G8",
"access_token":"eyJh...mgL4",
"expires_in":...output omitted...}[user@host ~]$
```

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

```
[user@host ~]$ curl -H "Authorization: Bearer eyJh...mgL4" \
> -Ls https://registry.redhat.io/v2/rhel8/mysql-80/tags/list \
> | python -mjson.tool
{
 "name": "rhel8/mysql-80",
 "tags": [
 "1.0",
 "1.2",
 ...output omitted...
```

**Note**

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

## Pulling Images

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

```
[user@host ~]$ podman pull [OPTIONS] [REGISTRY[:PORT]/]NAME[:TAG]
```

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

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

```
[user@host ~]$ podman pull quay.io/bitnami/nginx
```

**Note**

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

## Listing Local Copies of Images

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

**Note**

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

Podman provides an `images` subcommand to list all the container images stored locally:

```
[user@host ~]$ podman images
REPOSITORY TAG IMAGE ID CREATED SIZE
registry.redhat.io/rhel8/mysql-80 latest ad5a0e6d030f 3 weeks ago 588 MB
```

## Image Tags

An image tag is a mechanism to support multiple releases of the same image. This feature is useful when multiple versions of the same software are provided, such as a production-ready container or the latest updates of the same software developed for community evaluation. Any Podman subcommand that requires a container image name accepts a tag parameter to differentiate

between multiple tags. If an image name does not contain a tag, then the tag value defaults to `latest`. For example, to pull an image with the tag `1` from `rhel8/mysql-80`, use the following command:

```
[user@host ~]$ podman pull registry.redhat.io/rhel8/mysql-80:1
```

To start a new container based on the `rhel8/mysql-80:1` image, use the following command:

```
[user@host ~]$ podman run registry.redhat.io/rhel8/mysql-80:1
```



## References

### **Red Hat Container Catalog**

<https://registry.redhat.io>

### **Quay.io**

<https://quay.io>

### **Docker Registry HTTP API V2**

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

### **RFC7235 - HTTP/1.1: Authentication**

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

## ► Quiz

# Working With Registries

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

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

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

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

### Image names/tags per registry

| Registry           | Image      |
|--------------------|------------|
| registry.redhat.io | nginx/1.16 |
|                    | mysql/8.0  |
|                    | httpd/2.4  |
| quay.io            | mysql/5.7  |
|                    | httpd/2.4  |

No images are locally available.

- 1. Which two commands display mysql images available for download from registry.redhat.io? (Choose two.)
  - a. podman search registry.redhat.io/mysql
  - b. podman images
  - c. podman pull mysql
  - d. podman search mysql
  
- 2. Which command is used to list all available image tags for the httpd container image?
  - a. podman search --list-tags httpd
  - b. podman images httpd
  - c. podman pull --all-tags=true httpd
  - d. There is no podman command available to search for tags.

► 3. Which two commands pull the httpd image with the 2.4 tag? (Choose two.)

- a. podman pull httpd:2.4
- b. podman pull httpd:latest
- c. podman pull quay.io/httpd
- d. podman pull registry.redhat.io/httpd:2.4

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

```
podman pull registry.redhat.io/httpd:2.4
podman pull quay.io/mysql:8.0
```

- a. quay.io/httpd:2.4 registry.redhat.io/mysql:8.0
- b. registry.redhat.io/httpd:2.4 registry.redhat.io/mysql:8.0
- c. registry.redhat.io/httpd:2.4 No image will be downloaded for mysql.
- d. quay.io/httpd:2.4 No image will be downloaded for mysql.

## ► Solution

# Working With Registries

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

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

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

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

### Image names/tags per registry

| Registry           | Image      |
|--------------------|------------|
| registry.redhat.io | nginx/1.16 |
|                    | mysql/8.0  |
|                    | httpd/2.4  |
| quay.io            | mysql/5.7  |
|                    | httpd/2.4  |

No images are locally available.

- 1. Which two commands display mysql images available for download from registry.redhat.io? (Choose two.)
  - a. podman search registry.redhat.io/mysql
  - b. podman images
  - c. podman pull mysql
  - d. podman search mysql
  
- 2. Which command is used to list all available image tags for the httpd container image?
  - a. podman search --list-tags httpd
  - b. podman images httpd
  - c. podman pull --all-tags=true httpd
  - d. There is no podman command available to search for tags.

► 3. Which two commands pull the httpd image with the 2.4 tag? (Choose two.)

- a. podman pull httpd:2.4
- b. podman pull httpd:latest
- c. podman pull quay.io/httpd
- d. podman pull registry.redhat.io/httpd:2.4

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

```
podman pull registry.redhat.io/httpd:2.4
podman pull quay.io/mysql:8.0
```

- a. quay.io/httpd:2.4 registry.redhat.io/mysql:8.0
- b. registry.redhat.io/httpd:2.4 registry.redhat.io/mysql:8.0
- c. registry.redhat.io/httpd:2.4 No image will be downloaded for mysql.
- d. quay.io/httpd:2.4 No image will be downloaded for mysql.

# Manipulating Container Images

## Objectives

After completing this section, students should be able to:

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

## Introduction

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

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



### Note

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

## Saving and Loading Images

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

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

```
[user@host ~]$ podman save [-o FILE_NAME] IMAGE_NAME[:TAG]
```

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

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

```
[user@host ~]$ podman save \
> -o mysql.tar registry.redhat.io/rhel8/mysql-80
```

Note the use of the `-o` option in this example.

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

```
[user@host ~]$ podman load [-i FILE_NAME]
```

The `load` command is the opposite of the `save` command.

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

```
[user@host ~]$ podman load -i mysql.tar
```

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



### Note

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

## Deleting Images

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



### Note

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

To delete an image from the local storage, run the `podman rmi` command.

```
[user@host ~]$ podman rmi [OPTIONS] IMAGE [IMAGE...]
```

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

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

## Deleting All Images

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

```
[user@host ~]$ podman rmi -a
```

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

## Modifying Images

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



### Warning

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

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

```
[user@host ~]$ podman commit [OPTIONS] CONTAINER \
> [REPOSITORY[:PORT]/]IMAGE_NAME[:TAG]
```

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

| Option                    | Description                                                                                   |
|---------------------------|-----------------------------------------------------------------------------------------------|
| <code>--author ""</code>  | Identifies who created the container image.                                                   |
| <code>--message ""</code> | Includes a commit message to the registry.                                                    |
| <code>--format</code>     | Selects the format of the image. Valid options are <code>oci</code> and <code>docker</code> . |



### Note

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

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

```
[user@host ~]$ podman ps
CONTAINER ID IMAGE ...
87bdfcc7c656 mysql ...output omitted... mysql-basic
```

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

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

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



### Note

The `diff` command only reports added, changed, or deleted files to the container file system. Files that are mounted to a running container are not considered part of the container file system. To retrieve the list of mounted files and directories for a running container, use the `podman inspect` command:

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

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

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

```
[user@host ~]$ podman commit mysql-basic mysql-custom
```

The previous example creates a new image named `mysql-custom`.

## Tagging Images

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

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



### Note

Usually, the tags are used by container developers to distinguish between multiple versions of the same software. Multiple tags are provided to identify a release easily. The official MySQL container image website uses the version as the tag's name (8.0). Also, the same image might have a second tag with the minor version to minimize the need to get the latest release for a specific version.

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

```
[user@host ~]$ podman tag [OPTIONS] IMAGE[:TAG] \
> [REGISTRYHOST/] [USERNAME/]NAME[:TAG]
```

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

```
[user@host ~]$ podman tag mysql-custom devops/mysql
```

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

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

```
[user@host ~]$ podman tag mysql-custom devops/mysql:snapshot
```

## Removing Tags from Images

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

```
[user@host ~]$ podman rmi devops/mysql:snapshot
```



### Note

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

## Best Practices for Tagging Images

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

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

## Publishing Images to a Registry

To publish an image to a registry, it must reside in the Podman local storage and be tagged for identification purposes. To push the image to the registry, use the `push` subcommand:

```
[user@host ~]$ podman push [OPTIONS] IMAGE [DESTINATION]
```

For example, to push the `bitnami/nginx` image to its repository, use the following command:

```
[user@host ~]$ podman push quay.io/bitnami/nginx
```

The previous example pushes the image to Quay.io.



### References

#### Podman site

<https://podman.io/>

## ► Guided Exercise

# Creating a Custom Apache Container Image

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

## Outcomes

You should be able to create a custom container image.

## Before You Begin

Make sure you have completed *Guided Exercise: Configuring the Classroom Environment* from Chapter 1 before executing any command of this practice.

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

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

## Instructions

- ▶ 1. Log in to your Quay.io account and start a container by using the image available at `quay.io/redhattraining/httpd-parent`. The `-p` option allows you to specify a redirect port. In this case, Podman forwards incoming requests on TCP port 8180 of the host to TCP port 80 of the container.

```
[student@workstation ~]$ podman login quay.io
Username: your_quay_username
Password: your_quay_password
Login Succeeded!
[student@workstation ~]$ podman run -d --name official-httpd \
> -p 8180:80 quay.io/redhattraining/httpd-parent
...output omitted...
Writing manifest to image destination
Storing signatures
`3a6baecaff2b`4e8c53b026e04847dda5976b773ade1a3a712b1431d60ac5915d
```

Your last line of output is different from the last line shown above. Note the first twelve characters.

- ▶ 2. Create an HTML page on the `official-httpd` container.
  - 2.1. Access the shell of the container by using the `podman exec` command and create an HTML page.

```
[student@workstation ~]$ podman exec -it official-httdp /bin/bash
bash-4.4# echo "DO180 Page" > /var/www/html/do180.html
```

- 2.2. Exit from the container.

```
bash-4.4# exit
```

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

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

You should see the following output:

```
DO180 Page
```

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

```
[student@workstation ~]$ podman diff official-httdp
C /etc
C /root
A /root/.bash_history
...output omitted...
C /tmp
C /var
C /var/log
C /var/log/httdp
A /var/log/httdp/access_log
A /var/log/httdp/error_log
C /var/www
C /var/www/html
A /var/www/html/do180.html
```



### Note

Often, web server container images label the /var/www/html directory as a volume. In these cases, any files added to this directory are not considered part of the container file system, and would not show in the output of the podman diff command. The quay.io/redhattraining/httdp-parent container image does not label the /var/www/html directory as a volume. As a result, the change to the /var/www/html/do180.html file is considered a change to the underlying container file system.

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

- 4.1. Stop the official-httdp container.

```
[student@workstation ~]$ podman stop official-httdp
'3a6baecaff2b`4e8c53b026e04847dda5976b773ade1a3a712b1431d60ac5915d
```

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

```
[student@workstation ~]$ podman commit \
> -a 'Your Name' official-httdp do180-custom-httdp
Getting image source signatures
Skipping fetch of repeat blob sha256:071d8bd765171080d01682844524be57ac9883e...
...output omitted...
Copying blob sha256:1e19be875ce6f5b9dece378755eb9df96ee205abfb4f165c797f59a9...
15.00 KB / 15.00 KB [=====] 0s
Copying config sha256:8049dc2e7d0a0b1a70fc0268ad236399d9f5fb686ad4e31c7482cc...
2.99 KB / 2.99 KB [=====] 0s
Writing manifest to image destination
Storing signatures
`31c3ac78e9d4`137c928da23762e7d32b00c428eb1036cab1caeeb399befef2a23
```

- 4.3. List the available container images.

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

The expected output is similar to the following:

| REPOSITORY                          | TAG    | IMAGE ID            | CREATED | SIZE |
|-------------------------------------|--------|---------------------|---------|------|
| localhost/do180-custom-httdp        | latest | <b>31c3ac78e9d4</b> | ...     | ...  |
| quay.io/redhattraining/httdp-parent | latest | 2cc07fbb5000        | ...     | ...  |

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

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

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

```
[student@workstation ~]$ podman tag do180-custom-httdp \
> quay.io/${RHT_OCP4_QUAY_USER}/do180-custom-httdp:v1.0
```

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

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

| REPOSITORY                                        | TAG    | IMAGE ID            | ... |
|---------------------------------------------------|--------|---------------------|-----|
| localhost/do180-custom-httdp                      | latest | <b>31c3ac78e9d4</b> | ... |
| quay.io/\${RHT_OCP4_QUAY_USER}/do180-custom-httdp | v1.0   | <b>31c3ac78e9d4</b> | ... |
| quay.io/redhattraining/httdp-parent               | latest | 2cc07fbb5000        | ... |

## 5.4. Publish the image to your Quay.io registry.

```
[student@workstation ~]$ podman push \
> quay.io/${RHT_OCP4_QUAY_USER}/do180-custom-httd:v1.0
Getting image source signatures
Copying blob sha256:071d8bd765171080d01682844524be57ac9883e53079b6ac66707e19...
200.44 MB / 200.44 MB [=====] 1m38s
...output omitted...
Copying config sha256:31c3ac78e9d4137c928da23762e7d32b00c428eb1036cab1caeab3...
2.99 KB / 2.99 KB [=====] 0s
Writing manifest to image destination
Copying config sha256:31c3ac78e9d4137c928da23762e7d32b00c428eb1036cab1caeab3...
0 B / 2.99 KB [-----] 0s
Writing manifest to image destination
Storing signatures
```

**Note**

Pushing the do180-custom-httd image creates a private repository in your Quay.io account. Currently, private repositories are disallowed by Quay.io free plans. You can either create the public repository prior to pushing the image, or change the repository to public afterwards.

5.5. Verify that the image is available from Quay.io. The `podman search` command requires the image to be indexed by Quay.io. That may take some hours to occur, so use the `podman pull` command to fetch the image. This proves that the image is available.

```
[student@workstation ~]$ podman pull \
> -q quay.io/${RHT_OCP4_QUAY_USER}/do180-custom-httd:v1.0
31c3ac78e9d4137c928da23762e7d32b00c428eb1036cab1caeab3
```

## ▶ 6. Create a container from the newly published image.

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

```
[student@workstation ~]$ podman run -d --name test-httd -p 8280:80 \
> ${RHT_OCP4_QUAY_USER}/do180-custom-httd:v1.0
c0f04e906bb12bd0e514cbd0e581d2746e04e44a468dfbc85bc29ffcc5acd16c
```

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

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

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

**Finish**

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

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

This concludes the guided exercise.

## ▶ Lab

# Managing Images

## Outcomes

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

## Before You Begin

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

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

## Instructions

1. Use the `podman pull` command to download the `quay.io/redhattraining/nginx:1.17` container image. This image is a copy of the official container image available at `docker.io/library/nginx:1.17`.  
Ensure that you successfully downloaded the image.
2. Start a new container using the Nginx image, according to the specifications listed in the following list.
  - *Name*: `official-nginx`
  - *Run as daemon*: `yes`
  - *Container image*: `nginx`
  - *Port forward*: from host port 8080 to container port 80.
3. Log in to the container using the `podman exec` command. Replace the contents of the `index.html` file with `D0180`. The web server directory is located at `/usr/share/nginx/html`.  
After the file has been updated, exit the container and use the `curl` command to access the web page.
4. Stop the running container and commit your changes to create a new container image. Give the new image a name of `do180/mynginx` and a tag of `v1.0-SNAPSHOT`. Use the following specifications:
  - Image name: `do180/mynginx`
  - Image tag: `v1.0-SNAPSHOT`
  - Author name: *your name*
5. Start a new container using the updated Nginx image, according to the specifications listed in the following list.
  - *Name*: `official-nginx-dev`
  - *Run as daemon*: `yes`

## Chapter 4 | Managing Container Images

- Container image: do180/mynginx:v1.0-SNAPSHOT
  - Port forward: from host port 8080 to container port 80.
6. Log in to the container using the `podman exec` command, and introduce a final change. Replace the contents of the file `/usr/share/nginx/html/index.html` file with D0180 Page.
- After the file has been updated, exit the container and use the `curl` command to verify the changes.
7. Stop the running container and commit your changes to create the final container image. Give the new image a name of `do180/mynginx` and a tag of `v1.0`. Use the following specifications:
- Image name: `do180/mynginx`
  - Image tag: `v1.0`
  - Author name: *your name*
8. Remove the development image `do180/mynginx:v1.0-SNAPSHOT` from local image storage.
9. Use the image tagged `do180/mynginx:v1.0` to create a new container with the following specifications:
- Container name: `my-nginx`
  - Run as daemon: yes
  - Container image: `do180/mynginx:v1.0`
  - Port forward: from host port 8280 to container port 80

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

## Evaluation

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

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

## Finish

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

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

This concludes the lab.

## ► Solution

# Managing Images

### Outcomes

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

### Before You Begin

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

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

### Instructions

1. Use the `podman pull` command to download the `quay.io/redhattraining/nginx:1.17` container image. This image is a copy of the official container image available at `docker.io/library/nginx:1.17`.  
Ensure that you successfully downloaded the image.

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

```
[student@workstation ~]$ podman pull quay.io/redhattraining/nginx:1.17
Trying to pull quay.io/redhattraining/nginx:1.17...
...output omitted...
Storing signatures
9beeba249f3ee158d3e495a6ac25c5667ae2de8a43ac2a8bfd2bf687a58c06c9
```

- 1.2. Ensure that the container image exists on the local system by running the `podman images` command.

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

This command produces output similar to the following:

| REPOSITORY                   | TAG  | IMAGE ID     | CREATED      | SIZE  |
|------------------------------|------|--------------|--------------|-------|
| quay.io/redhattraining/nginx | 1.17 | 9beeba249f3e | 6 months ago | 428MB |

2. Start a new container using the Nginx image, according to the specifications listed in the following list.
  - *Name*: `official-nginx`
  - *Run as daemon*: `yes`
  - *Container image*: `nginx`
  - *Port forward*: from host port 8080 to container port 80.

- 2.1. On workstation, use the `podman run` command to create a container named `official-nginx`.

```
[student@workstation ~]$ podman run --name official-nginx \
> -d -p 8080:80 quay.io/redhattraining/nginx:1.17
b9d5739af239914b371025c38340352ac1657358561e7ebbd5472dfd5ff97788
```

3. Log in to the container using the `podman exec` command. Replace the contents of the `index.html` file with D0180. The web server directory is located at `/usr/share/nginx/html`.

After the file has been updated, exit the container and use the `curl` command to access the web page.

- 3.1. Log in to the container by using the `podman exec` command.

```
[student@workstation ~]$ podman exec -it official-nginx /bin/bash
root@b9d5739af239:/#
```

- 3.2. Update the `index.html` file located at `/usr/share/nginx/html`. The file should read D0180.

```
root@b9d5739af239:/# echo 'D0180' > /usr/share/nginx/html/index.html
```

- 3.3. Exit the container.

```
root@b9d5739af239:/# exit
```

- 3.4. Use the `curl` command to ensure that the `index.html` file is updated.

```
[student@workstation ~]$ curl 127.0.0.1:8080
D0180
```

4. Stop the running container and commit your changes to create a new container image. Give the new image a name of `do180/mynginx` and a tag of `v1.0-SNAPSHOT`. Use the following specifications:

- Image name: `do180/mynginx`
- Image tag: `v1.0-SNAPSHOT`
- Author name: `your name`

- 4.1. Use the ``podman stop`` command to stop the `official-nginx` container.

```
[student@workstation ~]$ podman stop official-nginx
b9d5739af239914b371025c38340352ac1657358561e7ebbd5472dfd5ff97788
```

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

```
[student@workstation ~]$ podman commit -a 'Your Name' \
> official-nginx do180/mynginx:v1.0-SNAPSHOT
Getting image source signatures
...output omitted...
Storing signatures
d6d10f52e258e4e88c181a56c51637789424e9261b208338404e82a26c960751
```

- 4.3. List the available container images to locate your newly created image.

```
[student@workstation ~]$ podman images
REPOSITORY TAG IMAGE ID CREATED ...
localhost/do180/mynginx v1.0-SNAPSHOT d6d10f52e258 30 seconds ago ...
quay.io/redhattraining/nginx 1.17 9beeba249f3e 6 months ago ...
```

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

- *Name*: official-nginx-dev
- *Run as daemon*: yes
- *Container image*: do180/mynginx:v1.0-SNAPSHOT
- *Port forward*: from host port 8080 to container port 80.

- 5.1. On workstation, use the podman run command to create a container named official-nginx-dev.

```
[student@workstation ~]$ podman run --name official-nginx-dev \
> -d -p 8080:80 do180/mynginx:v1.0-SNAPSHOT
cfa21f02a77d0e46e438c255f83dea2dfb89bb5aa72413a28866156671f0bbbb
```

6. Log in to the container using the podman exec command, and introduce a final change. Replace the contents of the file /usr/share/nginx/html/index.html file with D0180 Page.

After the file has been updated, exit the container and use the curl command to verify the changes.

- 6.1. Log in to the container by using the podman exec command.

```
[student@workstation ~]$ podman exec -it official-nginx-dev /bin/bash
root@cfa21f02a77d:#
```

- 6.2. Update the index.html file located at /usr/share/nginx/html. The file should read D0180 Page.

```
root@cfa21f02a77d:# echo 'D0180 Page' > /usr/share/nginx/html/index.html
```

- 6.3. Exit the container.

```
root@cfa21f02a77d:# exit
```

- 6.4. Use the curl command to ensure that the index.html file is updated.

```
[student@workstation ~]$ curl 127.0.0.1:8080
DO180 Page
```

7. Stop the running container and commit your changes to create the final container image. Give the new image a name of do180/mynginx and a tag of v1.0. Use the following specifications:

- Image name: do180/mynginx
- Image tag: v1.0
- Author name: *your name*

- 7.1. Use the ` podman sto`p command to stop the official-nginx-dev container.

```
[student@workstation ~]$ podman stop official-nginx-dev
cfa21f02a77d0e46e438c255f83dea2dfb89bb5aa72413a28866156671f0bbbb
```

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

```
[student@workstation ~]$ podman commit -a 'Your Name' \
> official-nginx-dev do180/mynginx:v1.0
Getting image source signatures
...output omitted...
Storing signatures
90915976c33de534e06778a74d2a8969c25ef5f8f58c0c1ab7aeaac19abd18af
```

- 7.3. List the available container images in order to locate your newly created image.

```
[student@workstation ~]$ podman images
REPOSITORY TAG IMAGE ID CREATED ...
localhost/do180/mynginx v1.0 90915976c33d 6 seconds ago ...
localhost/do180/mynginx v1.0-SNAPSHOT d6d10f52e258 8 minutes ago ...
quay.io/redhattraining/nginx 1.17 9beeba249f3e 6 months ago ...
```

8. Remove the development image do180/mynginx:v1.0-SNAPSHOT from local image storage.

- 8.1. Despite being stopped, the official-nginx-dev is still present. Display the container with the podman ps command with the -a flag.

```
[student@workstation ~]$ podman ps -a \
> --format="{{.ID}} {{.Names}} {{.Status}}"
cfa21f02a77d official-nginx-dev Exited (0) 9 minutes ago
b9d5739af239 official-nginx Exited (0) 12 minutes ago
```

- 8.2. Remove the container with the podman rm command.

```
[student@workstation ~]$ podman rm official-nginx-dev
cfa21f02a77d0e46e438c255f83dea2dfb89bb5aa72413a28866156671f0bbbb
```

- 8.3. Verify that the container is deleted by resubmitting the same `podman ps` command.

```
[student@workstation ~]$ podman ps -a \
> --format="{{.ID}} {{.Names}} {{.Status}}"
b9d5739af239 official-nginx Exited (0) 12 minutes ago
```

- 8.4. Use the `` podman rm`` command to remove the `do180/mynginx:v1.0-SNAPSHOT` image.

```
[student@workstation ~]$ podman rmi do180/mynginx:v1.0-SNAPSHOT
Untagged: localhost/do180/mynginx:v1.0-SNAPSHOT
```

- 8.5. Verify that the image is no longer present by listing all images using the `podman images` command.

```
[student@workstation ~]$ podman images
REPOSITORY TAG IMAGE ID CREATED SIZE
localhost/do180/mynginx v1.0 90915976c33d 5 minutes ago 131MB
quay.io/redhattraining/nginx 1.17 9beeba249f3e 6 months ago 131MB
```

9. Use the image tagged `do180/mynginx:v1.0` to create a new container with the following specifications:

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

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

- 9.1. Use the `` podman run`` command to create the `my-nginx` container, according to the specifications.

```
[student@workstation ~]$ podman run -d --name my-nginx \
> -p 8280:80 do180/mynginx:v1.0
51958c8ec8d2613bd26f85194c66ca96c95d23b82c43b23b0f0fb9fded74da20
```

- 9.2. Use the `curl` command to ensure that the `index.html` page is available and returns the custom content.

```
[student@workstation ~]$ curl 127.0.0.1:8280
DO180 Page
```

## Evaluation

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

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

## Finish

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

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

This concludes the lab.

# Summary

---

In this chapter, you learned:

- The Red Hat Container Catalog provides tested and certified images at [registry.redhat.io](https://registry.redhat.io).
- Podman can interact with remote container registries to search, pull, and push container images.
- Image tags are a mechanism to support multiple releases of a container image.
- Podman provides commands to manage container images both in local storage and as compressed files.
- Use the `podman commit` command to create an image from a container.



## Chapter 5

# Creating Custom Container Images

### Goal

Design and code a Containerfile to build a custom container image.

### Objectives

- Describe the approaches for creating custom container images.
- Create a container image using common Containerfile commands.

### Sections

- Designing Custom Container Images (and Quiz)
- Building Custom Container Images with Containerfiles (and Guided Exercise)

### Lab

- Creating Custom Container Images

# Designing Custom Container Images

## Objectives

After completing this section, students should be able to:

- Describe the approaches for creating custom container images.
- Find existing Containerfiles to use as a starting point for creating a custom container image.
- Define the role played by the Red Hat Software Collections Library (RHSCl) in designing container images from the Red Hat registry.
- Describe the Source-to-Image (S2I) alternative to Containerfiles.

## Reusing Existing Containerfiles

One method of creating container images previously discussed requires that you create a container, modify it to meet the requirements of the application to run in it, and then commit the changes to an image. This option, although straightforward, is only suitable for using or testing very specific changes. It does not follow best software practices, like maintainability, automation of building, and repeatability.

Containerfiles are another option for creating container images that addresses these limitations. Containerfiles are easy to share, version control, reuse, and extend.

Containerfiles also make it easy to extend one image, called a child image, from another image, called a parent image. A child image incorporates everything in the parent image and all changes and additions made to create it.



### Note

For this remainder of this course, a file referred to as a Containerfile can be a file named either *Containerfile* or *Dockerfile*. Both share the same syntax. *Containerfile* is the default name used by OCI compliant tools.

To share and reuse images, many popular applications, languages, and frameworks are already available in public image registries, such as Quay.io [<https://quay.io>]. It is not trivial to customize an application configuration to follow recommended practices for containers, and so starting from a proven parent image usually saves a lot of work.

Using a high-quality parent image enhances maintainability, especially if the parent image is kept updated by its author to account for bug fixes and security issues.

Creating a Containerfile for building a child image from an existing container image is often used in the following typical scenarios:

- Add new runtime libraries, such as database connectors.
- Include organization-wide customization such as SSL certificates and authentication providers.
- Add internal libraries to be shared as a single image layer by multiple container images for different applications.

Changing an existing Containerfile to create a new image can be a sensible approach in other scenarios. For example:

- Trim the container image by removing unused material (such as man pages, or documentation found in /usr/share/doc).
- Lock either the parent image or some included software package to a specific release to lower risk related to future software updates.

Two sources of container images to use, either as parent images or for modification of the Containerfiles, are Docker Hub and the Red Hat Software Collections Library (RHSCl).

## Working with the Red Hat Software Collections Library

Red Hat Software Collections Library (RHSCl), or simply Software Collections, is Red Hat's solution for developers who require the latest development tools that usually do not fit the standard RHEL release schedule.

Red Hat Enterprise Linux (RHEL) provides a stable environment for enterprise applications. This requires RHEL to keep the major releases of upstream packages at the same level to prevent API and configuration file format changes. Security and performance fixes are backported from later upstream releases, but new features that would break backwards compatibility are not backported.

RHSCl allows software developers to use the latest version without impacting RHEL, because RHSCl packages do not replace or conflict with default RHEL packages. Default RHEL packages and RHSCl packages are installed side by side.



### Note

All RHEL subscribers have access to the RHSCl. To enable a particular software collection for a specific user or application environment (for example, MySQL 5.7, which is named rh-mysql57), enable the RHSCl software Yum repositories and follow a few simple steps.

## Finding Containerfiles from the Red Hat Software Collections Library

RHSCl is the source of most container images provided by the Red Hat image registry for use by RHEL Atomic Host and OpenShift Container Platform customers.

Red Hat provides RHSCl Containerfiles and related sources in the `rhscl-dockerfiles` package available from the RHSCl repository. Community users can get Containerfiles for CentOS-based equivalent container images from GitHub's repository at <https://github.com/sclorg?q=-container>.



### Note

Many RHSCl container images include support for Source-to-Image (S2I), best known as an OpenShift Container Platform feature. Having support for S2I does not affect the use of these container images with Docker.

## Container Images in Red Hat Container Catalog (RHCC)

Critical applications require trusted containers. The Red Hat Container Catalog is a repository of reliable, tested, certified, and curated collection of container images built on versions of Red Hat Enterprise Linux (RHEL) and related systems. Container images available through RHCC have undergone a quality-assurance process. All components have been rebuilt by Red Hat to avoid known security vulnerabilities. They are upgraded on a regular basis so that they contain the required version of software even when a new image is not yet available. Using RHCC, you can browse and search for images, and you can access information about each image, such as its version, contents, and usage.

### Searching for Images Using Quay.io

Quay.io is an advanced container repository from CoreOS optimized for team collaboration. You can search for container images using <https://quay.io/search>.

Clicking on an image's name provides access to the image information page, including access to all existing tags for the image and the command to pull the image.

### Finding Containerfiles on Docker Hub

Be careful with images from Docker Hub. Anyone can create a Docker Hub account and publish container images there. There are no general assurances about quality and security; images on Docker Hub range from professionally supported to one-time experiments. Each image has to be evaluated individually.

After searching for an image, the documentation page might provide a link to its Containerfile. For example, the first result when searching for `mysql` is the documentation page for the MySQL official image at [https://hub.docker.com/\\_/mysql](https://hub.docker.com/_/mysql).

On that page, under the `Supported tags and respective Dockerfile links` section, each of the tags points to the `docker-library` GitHub project, which hosts Containerfiles for images built by the Docker community automatic build system.

The direct URL for the Docker Hub MySQL 8.0 Containerfile tree is <https://github.com/docker-library/mysql/blob/master/8.0>.

### Describing How to Use the OpenShift Source-to-Image Tool

Source-to-Image (S2I) provides an alternative to using Containerfiles to create new container images that can be used either as a feature from OpenShift or as the standalone `s2i` utility. S2I allows developers to work using their usual tools, instead of learning Containerfile syntax and using operating system commands such as `yum`. The `S2I` utility usually creates slimmer images with fewer layers.

S2I uses the following process to build a custom container image for an application:

1. Start a container from a base container image called the builder image. This image includes a programming language runtime and essential development tools, such as compilers and package managers.
2. Fetch the application source code, usually from a Git server, and send it to the container.
3. Build the application binary files inside the container.

4. Save the container, after some clean up, as a new container image, which includes the programming language runtime and the application binaries.

The builder image is a regular container image following a standard directory structure and providing scripts that are called during the S2I process. Most of these builder images can also be used as base images for Containerfiles, outside of the S2I process.

The `s2i` command is used to run the S2I process outside of OpenShift, in a Docker-only environment. It can be installed on a RHEL system from the `source-to-image` RPM package, and on other platforms, including Windows and Mac OS, from the installers available in the S2I project on GitHub.



## References

### **Red Hat Software Collections Library (RHSCl)**

<https://access.redhat.com/documentation/en/red-hat-software-collections/>

### **Red Hat Container Catalog (RHCC)**

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

### **RHSCl Dockerfiles on GitHub**

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

### **Using Red Hat Software Collections Container Images**

<https://access.redhat.com/articles/1752723>

### **Quay.io**

<https://quay.io/search>

### **Docker Hub**

<https://hub.docker.com/>

### **Docker Library GitHub project**

<https://github.com/docker-library>

### **The S2I GitHub project**

<https://github.com/openshift/source-to-image>

## ► Quiz

# Approaches to Container Image Design

Choose the correct answers to the following questions:

- ▶ 1. **Which method for creating container images is recommended by the containers community?**
  - a. Run commands inside a basic OS container, commit the container, and then save or export it as a new container image.
  - b. Run commands from a Containerfile and push the generated container image to an image registry.
  - c. Create the container image layers manually from tar files.
  - d. Run the `podman build` command to process a container image description in YAML format.
  
- ▶ 2. **What are two advantages of using the standalone S2I process as an alternative to Containerfiles? (Choose two.)**
  - a. Requires no additional tools apart from a basic Podman setup.
  - b. Creates smaller container images with fewer layers.
  - c. Reuses high-quality builder images.
  - d. Automatically updates the child image as the parent image changes (for example, with security fixes).
  - e. Creates images compatible with OpenShift, unlike container images created from Docker tools.
  
- ▶ 3. **What are two typical scenarios for creating a Containerfile to build a child image from an existing image? (Choose two.)**
  - a. Adding new runtime libraries.
  - b. Setting constraints on a container's access to the host machine's CPU.
  - c. Adding internal libraries to be shared as a single image layer by multiple container images for different applications.

## ► Solution

# Approaches to Container Image Design

Choose the correct answers to the following questions:

- ▶ 1. **Which method for creating container images is recommended by the containers community?**
  - a. Run commands inside a basic OS container, commit the container, and then save or export it as a new container image.
  - b. Run commands from a Containerfile and push the generated container image to an image registry.
  - c. Create the container image layers manually from tar files.
  - d. Run the podman build command to process a container image description in YAML format.
  
- ▶ 2. **What are two advantages of using the standalone S2I process as an alternative to Containerfiles? (Choose two.)**
  - a. Requires no additional tools apart from a basic Podman setup.
  - b. Creates smaller container images with fewer layers.
  - c. Reuses high-quality builder images.
  - d. Automatically updates the child image as the parent image changes (for example, with security fixes).
  - e. Creates images compatible with OpenShift, unlike container images created from Docker tools.
  
- ▶ 3. **What are two typical scenarios for creating a Containerfile to build a child image from an existing image? (Choose two.)**
  - a. Adding new runtime libraries.
  - b. Setting constraints on a container's access to the host machine's CPU.
  - c. Adding internal libraries to be shared as a single image layer by multiple container images for different applications.

# Building Custom Container Images with Containerfiles

## Objectives

After completing this section, students should be able to create a container image using common Containerfile commands.

## Building Base Containers

A Containerfile is a mechanism to automate the building of container images. Building an image from a Containerfile is a three-step process.

1. Create a working directory
2. Write the Containerfile
3. Build the image with Podman

### Create a Working Directory

The working directory is the directory containing all files needed to build the image. Creating an empty working directory is good practice to avoid incorporating unnecessary files into the image. For security reasons, the root directory, /, should never be used as a working directory for image builds.

### Write the Containerfile Specification

A Containerfile is a text file named either *Containerfile* or *Dockerfile* that contains the instructions needed to build the image. The basic syntax of a Containerfile follows:

```
Comment
INSTRUCTION arguments
```

Lines that begin with a hash, or pound, symbol (#) are comments. *INSTRUCTION* states for any Containerfile instruction keyword. Instructions are not case-sensitive, but the convention is to make instructions all uppercase to improve visibility.

The first non-comment instruction must be a FROM instruction to specify the base image. Containerfile instructions are executed into a new container using this image and then committed to a new image. The next instruction (if any) executes into that new image. The execution order of instructions is the same as the order of their appearance in the Containerfile.



#### Note

The ARG instruction can appear before the FROM instruction, but ARG instructions are outside the objectives for this section.

Each Containerfile instruction runs in an independent container using an intermediate image built from every previous command. This means each instruction is independent from other instructions

in the Containerfile. The following is an example Containerfile for building a simple Apache web server container:

```
This is a comment line ①
FROM ubi8/ubi:8.3 ②
LABEL description="This is a custom httpd container image" ③
MAINTAINER John Doe <jdoe@xyz.com> ④
RUN yum install -y httpd ⑤
EXPOSE 80 ⑥
ENV LogLevel "info" ⑦
ADD http://someserver.com/filename.pdf /var/www/html ⑧
COPY ./src/ /var/www/html/ ⑨
USER apache ⑩
ENTRYPOINT ["/usr/sbin/httpd"] ⑪
CMD ["-D", "FOREGROUND"] ⑫
```

- ① Lines that begin with a hash, or pound, sign (#) are comments.
- ② The `FROM` instruction declares that the new container image extends `ubi8/ubi:8.3` container base image. Containerfiles can use any other container image as a base image, not only images from operating system distributions. Red Hat provides a set of container images that are certified and tested and highly recommends using these container images as a base.
- ③ The `LABEL` is responsible for adding generic metadata to an image. A `LABEL` is a simple key-value pair.
- ④ `MAINTAINER` indicates the `Author` field of the generated container image's metadata. You can use the `podman inspect` command to view image metadata.
- ⑤ `RUN` executes commands in a new layer on top of the current image. The shell that is used to execute commands is `/bin/sh`.
- ⑥ `EXPOSE` indicates that the container listens on the specified network port at runtime. The `EXPOSE` instruction defines metadata only; it does not make ports accessible from the host. The `-p` option in the `podman run` command exposes container ports from the host.
- ⑦ `ENV` is responsible for defining environment variables that are available in the container. You can declare multiple `ENV` instructions within the Containerfile. You can use the `env` command inside the container to view each of the environment variables.
- ⑧ `ADD` instruction copies files or folders from a local or remote source and adds them to the container's file system. If used to copy local files, those must be in the working directory. `ADD` instruction unpacks local `.tar` files to the destination image directory.
- ⑨ `COPY` copies files from the working directory and adds them to the container's file system. It is not possible to copy a remote file using its URL with this Containerfile instruction.
- ⑩ `USER` specifies the username or the UID to use when running the container image for the `RUN`, `CMD`, and `ENTRYPOINT` instructions. It is a good practice to define a different user other than `root` for security reasons.
- ⑪ `ENTRYPOINT` specifies the default command to execute when the image runs in a container. If omitted, the default `ENTRYPOINT` is `/bin/sh -c`.

- ⑫ CMD provides the default arguments for the ENTRYPPOINT instruction. If the default ENTRYPPOINT applies (`/bin/sh -c`), then CMD forms an executable command and parameters that run at container start.

## CMD and ENTRYPPOINT

ENTRYPPOINT and CMD instructions have two formats:

- Exec form (using a JSON array):

```
ENTRYPPOINT ["command", "param1", "param2"]
CMD ["param1", "param2"]
```

- Shell form:

```
ENTRYPPOINT command param1 param2
CMD param1 param2
```

Exec form is the preferred form. Shell form wraps the commands in a `/bin/sh -c` shell, creating a sometimes unnecessary shell process. Also, some combinations are not allowed or may not work as expected. For example, if ENTRYPPOINT is `["ping"]` (exec form) and CMD is `localhost` (shell form), then the expected executed command is `ping localhost`, but the container tries `ping /bin/sh -c localhost`, which is a malformed command.

The Containerfile should contain at most one ENTRYPPOINT and one CMD instruction. If more than one of each is present, then only the last instruction takes effect. CMD can be present without specifying an ENTRYPPOINT. In this case, the base image's ENTRYPPOINT applies, or the default ENTRYPPOINT if none is defined.

Podman can override the CMD instruction when starting a container. If present, all parameters for the `podman run` command after the image name form the CMD instruction. For example, the following instruction causes the running container to display the current time.

```
ENTRYPPOINT ["/bin/date", "+%H:%M"]
```

The ENTRYPPOINT defines both the command to be executed and the parameters. So the CMD instruction cannot be used. The following example provides the same functionality as the preceding one, with the added benefit of the CMD instruction being overwritable when a container starts.

```
ENTRYPPOINT ["/bin/date"]
CMD ["+%H:%M"]
```

In both cases, when a container starts without providing a parameter, the current time is displayed:

```
[student@workstation ~]$ sudo podman run -it do180/rhel
11:41
```

In the second case, if a parameter appears after the image name in the `podman run` command, it overwrites the CMD instruction. The following command displays the current day of the week instead of the time:

```
[student@workstation demo-basic]$ sudo podman run -it do180/rhel +%A
Tuesday
```

Another approach is using the default ENTRYPPOINT and the CMD instruction to define the initial command. The following instruction displays the current time, with the added benefit of being able to be overridden at run time.

```
CMD ["date", "+%H:%M"]
```

## ADD and COPY

The ADD and COPY instructions have two forms:

- The Shell form:

```
ADD <_source_ >... <_destination_ >
COPY <_source_ >... <_destination_>
```

- The Exec form:

```
ADD ["<_source_ >", ... "<_destination_ >"]
COPY ["<_source_ >", ... "<_destination_>"]
```

If the source is a file system path, it must be inside the working directory.

The ADD instruction also allows you to specify a resource using a URL.

```
ADD http://someserver.com/filename.pdf /var/www/html
```

If the source is a compressed file, then the ADD instruction decompresses the file to the *destination* folder. The COPY instruction does not have this functionality.



### Warning

Both the ADD and COPY instructions copy the files, retaining permissions, with `root` as the owner, even if the `USER` instruction is specified. Red Hat recommends using a RUN instruction after the copy to change the owner and avoid permission denied errors.

## Layering Image

Each instruction in a Containerfile creates a new image layer. Having too many instructions in a Containerfile causes too many layers, resulting in large images. For example, consider the following RUN instructions in a Containerfile:

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

## Chapter 5 | Creating Custom Container Images

This example shows the creation of a container image having three layers (one for each RUN instruction). Red Hat recommends minimizing the number of layers. You can achieve the same objective by creating a single layer image using the `&&` conjunction in the RUN instruction.

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

The problem with this approach is that the readability of the Containerfile decays. Use the `\` escape code to insert line breaks and improve readability. You can also indent lines to align the commands:

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

This example creates only one layer and the readability improves. RUN, COPY, and ADD instructions create new image layers, but RUN can be improved this way.

Red Hat recommends applying similar formatting rules to other instructions accepting multiple parameters, such as LABEL and ENV:

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

```
ENV MYSQL_ROOT_PASSWORD="my_password" \
 MYSQL_DATABASE "my_database"
```

## Building Images with Podman

The `podman build` command processes the Containerfile and builds a new image based on the instructions it contains. The syntax for this command is as follows:

```
$ podman build -t NAME:TAG DIR
```

*DIR* is the path to the working directory. It can be the current directory as designated by a dot (`.`) if the working directory is the current directory. *NAME:TAG* is a name with a tag given to the new image. If *TAG* is not specified, then the image is automatically tagged as `latest`.



### Note

The current working directory is by default the path for the Containerfile, but you can specify a different directory using the `-f` flag. For more information you can check the Best practices for writing Dockerfiles [[https://docs.docker.com/develop/develop-images/dockerfile\\_best-practices/](https://docs.docker.com/develop/develop-images/dockerfile_best-practices/)].



## References

### Dockerfile Reference Guide

<https://docs.docker.com/engine/reference/builder/>

### Creating base images

<https://docs.docker.com/engine/userguide/eng-image/baseimages/>

## ► Guided Exercise

# Creating a Basic Apache Container Image

In this exercise, you will create a basic Apache container image.

## Outcomes

You should be able to create a custom Apache container image built on a Red Hat Universal Base Image 8 image.

## Before You Begin

Run the following command to download the relevant lab files and to verify that Docker is running:

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

## Instructions

### ► 1. Create the Apache Containerfile.

- 1.1. Open a terminal on **workstation**. Using your preferred editor, create a new Containerfile.

```
[student@workstation
~]$ vim /home/student/D0180/labs/dockerfile-create/Containerfile
```

- 1.2. Use UBI 8.3 as a base image by adding the following **FROM** instruction at the top of the new Containerfile:

```
FROM ubi8/ubi:8.3
```

- 1.3. Beneath the **FROM** instruction, include the **MAINTAINER** instruction to set the **Author** field in the new image. Replace the values to include your name and email address.

```
MAINTAINER Your Name <_youremail_>
```

- 1.4. Below the **MAINTAINER** instruction, add the following **LABEL** instruction to add description metadata to the new image:

```
LABEL description="A custom Apache container based on UBI 8"
```

- 1.5. Add a **RUN** instruction with a **yum install** command to install Apache on the new container.

```
RUN yum install -y httpd && \
 yum clean all
```

- 1.6. Add a RUN instruction to replace contents of the default HTTPD home page.

```
RUN echo "Hello from Containerfile" > /var/www/html/index.html
```

- 1.7. Use the EXPOSE instruction beneath the RUN instruction to document the port that the container listens to at runtime. In this instance, set the port to 80, because it is the default for an Apache server.

```
EXPOSE 80
```



### Note

The EXPOSE instruction does not actually make the specified port available to the host; rather, the instruction serves as metadata about the ports on which the container is listening.

- 1.8. At the end of the file, use the following CMD instruction to set `httpd` as the default entry point:

```
CMD ["httpd", "-D", "FOREGROUND"]
```

- 1.9. Verify that your Containerfile matches the following before saving and proceeding with the next steps:

```
FROM ubi8/ubi:8.3

MAINTAINER Your Name <_youremail_>

LABEL description="A custom Apache container based on UBI 8"

RUN yum install -y httpd && \
 yum clean all

RUN echo "Hello from Containerfile" > /var/www/html/index.html

EXPOSE 80

CMD ["httpd", "-D", "FOREGROUND"]
```

- 2. Build and verify the Apache container image.

- 2.1. Use the following commands to create a basic Apache container image using the newly created Containerfile:

```
[student@workstation ~]$ cd /home/student/D0180/labs/dockerfile-create
[student@workstation dockerfile-create]$ podman build --layers=false \
> -t do180/apache .
```

```

STEP 1: FROM ubi8/ubi:8.3
Getting image source signatures ①
Copying blob sha256:...output omitted...
71.46 MB / 71.46 MB [=====] 18s
...output omitted...
Storing signatures
STEP 2: MAINTAINER Your Name <youremail>
STEP 3: LABEL description="A custom Apache container based on UBI 8"
STEP 4: RUN yum install -y httpd && yum clean all
Loaded plugins: ovl, product-id, search-disabled-repos, subscription-manager
...output omitted...
STEP 5: RUN echo "Hello from Containerfile" > /var/www/html/index.html
STEP 6: EXPOSE 80
STEP 7: CMD ["httpd", "-D", "FOREGROUND"]
STEP 8: COMMIT ...output omitted... localhost/do180/apache:latest
Getting image source signatures
...output omitted...
Storing signatures
b49375fa8ee1e549dc1b72742532f01c13e0ad5b4a82bb088e5befbe59377bcf

```

- ① The container image listed in the FROM instruction is only downloaded if it is not already present in local storage.



### Note

Podman creates many anonymous intermediate images during the build process. They are not be listed unless -a is used. Use the --layers=false option of the build subcommand to instruct Podman to delete intermediate images.

- After the build process has finished, run `podman images` to see the new image in the image repository.

| REPOSITORY                          | TAG    | IMAGE ID     | CREATED               |
|-------------------------------------|--------|--------------|-----------------------|
| localhost/do180/apache              | latest | 8ebfe343e08c | 15 seconds ago 234 MB |
| registry.access.redhat.com/ubi8/ubi | 8.3    | 4199acc83c6a | 6 weeks ago 213 MB    |

- 3. Run the Apache container.

- Use the following command to run a container using the Apache image:

```
[student@workstation dockerfile-create]$ podman run --name lab-apache \
> -d -p 10080:80 do180/apache
fa1d1c450e8892ae085dd8bbf763edac92c41e6ffaa7ad6ec6388466809bb391
```

- Run the `podman ps` command to see the running container.

```
[student@workstation dockerfile-create]$ podman ps
CONTAINER ID IMAGE COMMAND ...output omitted...
fa1d1c450e88 localhost/do180/apache:latest httpd -D FOREGROU...output omitted...
```

- 3.3. Use the curl command to verify that the server is running.

```
[student@workstation dockerfile-create]$ curl 127.0.0.1:10080
Hello from Containerfile
```

## Finish

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

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

This concludes the guided exercise.

## ▶ Lab

# Creating Custom Container Images

In this lab, you will create a Containerfile to build a custom Apache Web Server container image. The custom image will be based on a RHEL 8.3 UBI image and serve a custom `index.html` page.

## Outcomes

You should be able to create a custom Apache Web Server container that hosts static HTML files.

## Before You Begin

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

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

## Instructions

1. Review the provided Containerfile stub in the `/home/student/D0180/labs/dockerfile-review/` folder. Edit the `Containerfile` and ensure that it meets the following specifications:
  - The base image is `ubi8/ubi:8.3`
  - Sets the desired author name and email ID with the `MAINTAINER` instruction
  - Sets the environment variable `PORT` to `8080`
  - Install Apache (`httpd` package).
  - Change the Apache configuration file `/etc/httpd/conf/httpd.conf` to listen to port `8080` instead of the default port `80`.
  - Change ownership of the `/etc/httpd/logs` and `/run/httpd` folders to user and group `apache` (UID and GID are `48`).
  - Expose the value set in the `PORT` environment variable so that container users know how to access the Apache Web Server.
  - Copy the contents of the `src/` folder in the lab directory to the Apache DocumentRoot file (`/var/www/html/`) inside the container.

The `src` folder contains a single `index.html` file that prints a `Hello World!` message.

- Start the Apache `httpd` daemon in the foreground using the `CMD` instruction and the following command:

```
httpd -D FOREGROUND
```

2. Build the custom Apache image with the name do180/custom-apache.
3. Create a new container in detached mode with the following characteristics:
  - Name: `containerfile`
  - Container image: `do180/custom-apache`
  - Port forward: from host port 20080 to container port 8080
  - Run as a daemon: yes

Verify that the container is ready and running.
4. Verify that the server is serving the HTML file.

## Evaluation

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

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

## Finish

From workstation, run the `lab dockerfile-review finish` command to complete this lab.

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

This concludes the lab.

## ► Solution

# Creating Custom Container Images

In this lab, you will create a Containerfile to build a custom Apache Web Server container image. The custom image will be based on a RHEL 8.3 UBI image and serve a custom `index.html` page.

## Outcomes

You should be able to create a custom Apache Web Server container that hosts static HTML files.

## Before You Begin

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

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

## Instructions

1. Review the provided Containerfile stub in the `/home/student/D0180/labs/dockerfile-review/` folder. Edit the `Containerfile` and ensure that it meets the following specifications:
  - The base image is `ubi8/ubi:8.3`
  - Sets the desired author name and email ID with the `MAINTAINER` instruction
  - Sets the environment variable `PORT` to `8080`
  - Install Apache (`httpd` package).
  - Change the Apache configuration file `/etc/httpd/conf/httpd.conf` to listen to port `8080` instead of the default port `80`.
  - Change ownership of the `/etc/httpd/logs` and `/run/httpd` folders to user and group `apache` (UID and GID are `48`).
  - Expose the value set in the `PORT` environment variable so that container users know how to access the Apache Web Server.
  - Copy the contents of the `src/` folder in the lab directory to the Apache DocumentRoot file (`/var/www/html/`) inside the container.

The `src` folder contains a single `index.html` file that prints a `Hello World!` message.

- Start the Apache `httpd` daemon in the foreground using the `CMD` instruction and the following command:

```
httpd -D FOREGROUND
```

**Chapter 5 |** Creating Custom Container Images

- 1.1. Use your preferred editor to modify the Containerfile located in the /home/student/D0180/labs/dockerfile-review/ folder.

```
[student@workstation ~]$ cd /home/student/D0180/labs/dockerfile-review/
[student@workstation dockerfile-review]$ vim Containerfile
```

- 1.2. Set the base image for the Containerfile to ubi8/ubi:8.3.

```
FROM ubi8/ubi:8.3
```

- 1.3. Set your name and email with a MAINTAINER instruction.

```
MAINTAINER Your Name <youremail>
```

- 1.4. Create an environment variable called PORT and set it to 8080.

```
ENV PORT 8080
```

- 1.5. Install Apache server.

```
RUN yum install -y httpd && \
 yum clean all
```

- 1.6. Change the Apache HTTP Server configuration file to listen on port 8080 and change ownership of the server working folders with a single RUN instruction.

```
RUN sed -ri -e "/^Listen 80/c\Listen ${PORT}" /etc/httpd/conf/httpd.conf && \
 chown -R apache:apache /etc/httpd/logs/ && \
 chown -R apache:apache /run/httpd/
```

- 1.7. Use the USER instruction to run the container as the apache user. Use the EXPOSE instruction to document the port that the container listens to at runtime. In this instance, set the port to the PORT environment variable, which is the default for an Apache server.

```
USER apache

Expose the custom port that you provided in the ENV var
EXPOSE ${PORT}
```

- 1.8. Copy all the files from the src folder to the Apache DocumentRoot path at /var/www/html.

```
Copy all files under src/ folder to Apache DocumentRoot (/var/www/html)
COPY ./src/ /var/www/html/
```

- 1.9. Finally, insert a CMD instruction to run httpd in the foreground, and then save the Containerfile.

```
Start Apache in the foreground
CMD ["httpd", "-D", "FOREGROUND"]
```

## 2. Build the custom Apache image with the name do180/custom-apache.

### 2.1. Verify the Containerfile for the custom Apache image.

The Containerfile for the custom Apache image should look similar to the following:

```
FROM ubi8/ubi:8.3

MAINTAINER Your Name <youremail>

ENV PORT 8080

RUN yum install -y httpd && \
 yum clean all

RUN sed -ri -e "/^Listen 80/c\Listen ${PORT}" /etc/httpd/conf/httpd.conf && \
 chown -R apache:apache /etc/httpd/logs/ && \
 chown -R apache:apache /run/httpd/

USER apache

Expose the custom port that you provided in the ENV var
EXPOSE ${PORT}

Copy all files under src/ folder to Apache DocumentRoot (/var/www/html)
COPY ./src/ /var/www/html/

Start Apache in the foreground
CMD ["httpd", "-D", "FOREGROUND"]
```

### 2.2. Run a podman build command to build the custom Apache image and name it do180/custom-apache.

```
[student@workstation dockerfile-review]$ podman build --layers=false \
> -t do180/custom-apache .
STEP 1: FROM ubi8/ubi:8.3
...output omitted...
STEP 2: MAINTAINER username <username@example.com>
STEP 3: ENV PORT 8080
STEP 4: RUN yum install -y httpd && yum clean all
...output omitted...
STEP 5: RUN sed -ri -e "/^Listen 80/c\Listen ${PORT}" /etc/httpd/conf/httpd.conf
&& chown -R apache:apache /etc/httpd/logs/ && chown -R apache:apache /
run/httpd/
STEP 6: USER apache
STEP 7: EXPOSE ${PORT}
STEP 8: COPY ./src/ /var/www/html/
STEP 9: CMD ["httpd", "-D", "FOREGROUND"]
STEP 10: COMMIT ...output omitted... localhost/do180/custom-apache:latest
...output omitted...
```

- 2.3. Run the `podman images` command to verify that the custom image is built successfully.

```
[student@workstation dockerfile-review]$ podman images
REPOSITORY TAG IMAGE ID CREATED
SIZE
localhost/do180/custom-apache latest 08fcf6d92b16 3 minutes ago
234 MB
registry.access.redhat.com/ubi8/ubi 8.3 4199acc83c6a 6 weeks ago
213 MB
```

3. Create a new container in detached mode with the following characteristics:

- Name: `containerfile`
- Container image: `do180/custom-apache`
- Port forward: from host port 20080 to container port 8080
- Run as a daemon: yes

Verify that the container is ready and running.

- 3.1. Create and run the container.

```
[student@workstation dockerfile-review]$ podman run -d \
> --name containerfile -p 20080:8080 do180/custom-apache
367823e35c4a...
```

- 3.2. Verify that the container is ready and running.

```
[student@workstation dockerfile-review]$ podman ps
... IMAGE COMMAND ... PORTS NAMES
... do180/custom... "httpd -D ..." ... 0.0.0.0:20080->8080/tcp containerfile
```

4. Verify that the server is serving the HTML file.

- 4.1. Run a `curl` command on `127.0.0.1:20080`

```
[student@workstation dockerfile-review]$ curl 127.0.0.1:20080
```

The output should be as follows:

```
<html>
<header><title>D0180 Hello!</title></header>
<body>
 Hello World! The containerfile-review lab works!
</body>
</html>
```

## Evaluation

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

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

## Finish

From workstation, run the `lab dockerfile-review finish` command to complete this lab.

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

This concludes the lab.

# Summary

---

In this chapter, you learned:

- A `Containerfile` contains instructions that specify how to construct a container image.
- Container images provided by Red Hat Container Catalog or Quay.io are a good starting point for creating custom images for a specific language or technology.
- Building an image from a `Containerfile` is a three-step process:
  1. Create a working directory.
  2. Specify the build instructions in a `Containerfile` file.
  3. Build the image with the `podman build` command.
- The Source-to-Image (S2I) process provides an alternative to `Containerfiles`. S2I implements a standardized container image build process for common technologies from application source code. This allows developers to focus on application development and not `Containerfile` development.



## Chapter 6

# Deploying Containerized Applications on OpenShift

### Goal

Deploy single container applications on OpenShift Container Platform.

### Objectives

- Describe the architecture of Kubernetes and Red Hat OpenShift Container Platform.
- Create standard Kubernetes resources.
- Create a route to a service.

### Sections

- Creating Kubernetes Resources
- Deploying a Database Server on OpenShift
- Creating Routes
- Exposing a Service as a Route
- Creating Applications with Source-to-Image
- Creating a Containerized Application with Source-to-Image

### Lab

- Lab: Deploying Containerized Applications on OpenShift

# Creating Kubernetes Resources

## Objectives

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

## The Red Hat OpenShift Container Platform (RHOC) Command-line Tool

The main method of interacting with an RHOC cluster is using the `oc` command. The basic usage of the command is through its subcommands in the following syntax:

```
$> oc <command>
```

Before interacting with a cluster, most operations require the user to log in. The syntax to log in is shown below:

```
$> oc login <clusterUrl>
```

## Describing Pod Resource Definition Syntax

RHOC runs containers inside Kubernetes pods, and to create a pod from a container image, OpenShift needs a *pod resource definition*. This can be provided either as a JSON or YAML text file, or can be generated from defaults by the `oc new-app` command or the OpenShift web console.

A pod is a collection of containers and other resources. An example of a WildFly application server pod definition in YAML format is shown below:

```
apiVersion: v1
kind: Pod1
metadata:
 name: wildfly2
 labels:
 name: wildfly3
spec:
 containers:
 - resources:
 limits:
 cpu: 0.5
 image: do276/todojee4
 name: wildfly
 ports:
 - containerPort: 80805
 name: wildfly
 env:6
 - name: MYSQL_ENV_MYSQL_DATABASE
 value: items
```

```

- name: MYSQL_ENV_MYSQL_USER
 value: user1
- name: MYSQL_ENV_MYSQL_PASSWORD
 value: mypa55

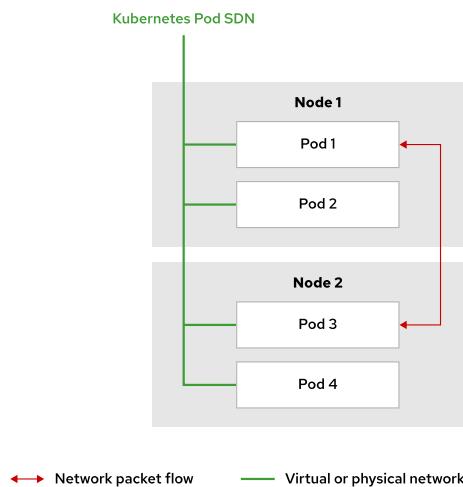
```

- ➊ Declares a Kubernetes pod resource type.
- ➋ A unique name for a pod in Kubernetes that allows administrators to run commands on it.
- ➌ Creates a label with a key named name that other resources in Kubernetes, usually as service, can use to find it.
- ➍ Defines the container image name.
- ➎ A container-dependent attribute identifying which port on the container is exposed.
- ➏ Defines a collection of environment variables.

Some pods may require environment variables that can be read by a container. Kubernetes transforms all the name and value pairs to environment variables. For instance, the MYSQL\_ENV\_MYSQL\_USER variable is declared internally by the Kubernetes runtime with a value of user1, and is forwarded to the container image definition. Because the container uses the same variable name to get the user's login, the value is used by the WildFly container instance to set the username that accesses a MySQL database instance.

## Describing Service Resource Definition Syntax

Kubernetes provides a virtual network that allows pods from different compute nodes to connect. But, Kubernetes provides no easy way for a pod to discover the IP addresses of other pods:

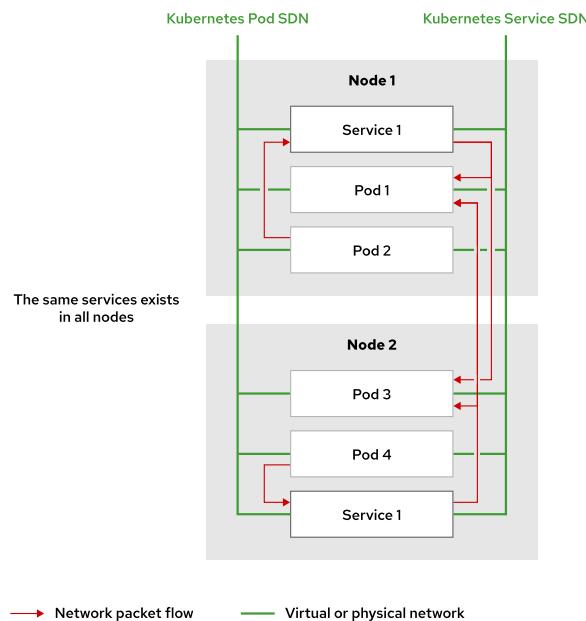


**Figure 6.1: Basic Kubernetes networking**

If Pod 3 fails and is restarted, it could return with a different IP address. This would cause Pod 1 to fail when attempting to communicate with Pod 3. A service layer provides the abstraction required to solve this problem.

Services are essential resources to any OpenShift application. They allow containers in one pod to open network connections to containers in another pod. A pod can be restarted for many reasons, and it gets a different internal IP address each time. Instead of a pod having to discover the IP

address of another pod after each restart, a service provides a stable IP address for other pods to use, no matter what compute node runs the pod after each restart:



**Figure 6.2: Kubernetes services networking**

Most real-world applications do not run as a single pod. They need to scale horizontally, so many pods run the same containers from the same pod resource definition to meet growing user demand. A service is tied to a set of pods, providing a single IP address for the whole set, and a load-balancing client request among member pods.

The set of pods running behind a service is managed by a Deployment resource. A Deployment resource embeds a ReplicationController that manages how many pod copies (replicas) have to be created, and creates new ones if any of them fail. Deployment and ReplicationController resources are explained later in this chapter.

The following example shows a minimal service definition in JSON syntax:

```
{
 "kind": "Service", ①
 "apiVersion": "v1",
 "metadata": {
 "name": "quotedb" ②
 },
 "spec": {
 "ports": [③
 {
 "port": 3306,
 "targetPort": 3306
 }
],
 "selector": {
 "name": "mysqldb" ④
 }
 }
}
```

```
 }
 }
}
```

- ➊ The kind of Kubernetes resource. In this case, a Service.
- ➋ A unique name for the service.
- ➌ `ports` is an array of objects that describes network ports exposed by the service. The `targetPort` attribute has to match a `containerPort` from a pod container definition, and the `port` attribute is the port that is exposed by the service. Clients connect to the service port and the service forwards packets to the pod `targetPort`.
- ➍ `selector` is how the service finds pods to forward packets to. The target pods need to have matching labels in their metadata attributes. If the service finds multiple pods with matching labels, it load balances network connections between them.

Each service is assigned a unique IP address for clients to connect to. This IP address comes from another internal OpenShift SDN, distinct from the pods' internal network, but visible only to pods. Each pod matching the `selector` is added to the service resource as an endpoint.

## Discovering Services

An application typically finds a service IP address and port by using environment variables. For each service inside an OpenShift project, the following environment variables are automatically defined and injected into containers for all pods inside the same project:

- `SVC_NAME_SERVICE_HOST` is the service IP address.
- `SVC_NAME_SERVICE_PORT` is the service TCP port.



### Note

The `SVC_NAME` part of the variable is changed to comply with DNS naming restrictions: letters are capitalized and underscores (`_`) are replaced by dashes (`-`).

Another way to discover a service from a pod is by using the OpenShift internal DNS server, which is visible only to pods. Each service is dynamically assigned an SRV record with an FQDN of the form:

```
SVC_NAME .PROJECT_NAME.svc.cluster.local
```

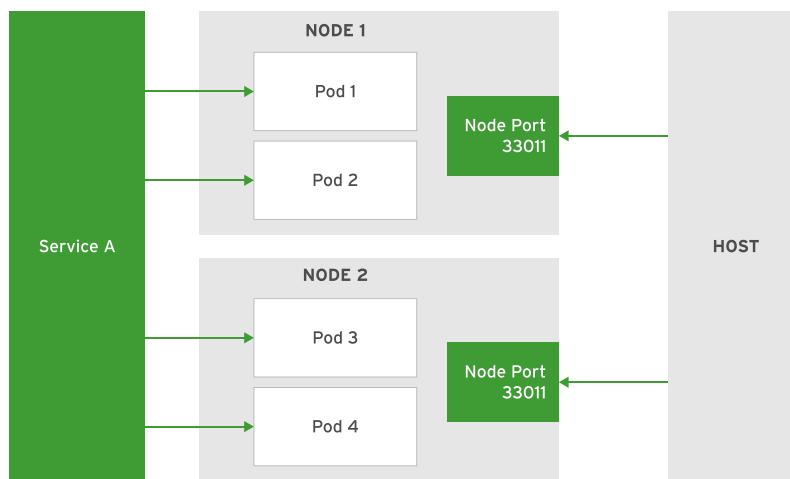
When discovering services using environment variables, a pod has to be created and started only after the service is created. If the application was written to discover services using DNS queries, however, it can find services created after the pod was started.

There are two ways for an application to access the service from outside an OpenShift cluster:

1. **NodePort type:** This is an older Kubernetes-based approach, where the service is exposed to external clients by binding to available ports on the compute node host, which then proxies connections to the service IP address. Use the `oc edit svc` command to edit service attributes and specify `NodePort` as the value for `type`, and provide a port value for the `nodePort` attribute. OpenShift then proxies connections to the service via the public IP address of the compute node host and the port value set in `NodePort`.

2. OpenShift Routes: This is the preferred approach in OpenShift to expose services using a unique URL. Use the `oc expose` command to expose a service for external access or expose a service from the OpenShift web console.

Figure 6.3 illustrates how NodePort services allow external access to Kubernetes services. OpenShift routes are covered in more detail later in this course.



**Figure 6.3: Alternative method for external access to a Kubernetes service**

OpenShift provides the `oc port-forward` command for forwarding a local port to a pod port. This is different from having access to a pod through a service resource:

- The port-forwarding mapping exists only on the workstation where the `oc` client runs, while a service maps a port for all network users.
- A service load balances connections to potentially multiple pods, whereas a port-forwarding mapping forwards connections to a single pod.



### Note

Red Hat discourages the use of the NodePort approach to avoid exposing the service to direct connections. Mapping via port-forwarding in OpenShift is considered a more secure alternative.

The following example demonstrates the use of the `oc port-forward` command:

```
[user@host ~]$ oc port-forward mysql-openshift-1-glqrp 3306:3306
```

The command forwards port 3306 from the developer machine to port 3306 on the db pod, where a MySQL server (inside a container) accepts network connections.



### Note

When running this command, make sure you leave the terminal window running. Closing the window or canceling the process stops the port mapping.

## Creating Applications

Simple applications, complex multitier applications, and microservice applications can be described by a single resource definition file. This single file would contain many pod definitions, service definitions to connect the pods, replication controllers or Deployment to horizontally scale the application pods, PersistentVolumeClaims to persist application data, and anything else needed that can be managed by OpenShift.

The `oc new-app` command can be used with the `-o json` or `-o yaml` option to create a skeleton resource definition file in JSON or YAML format, respectively. This file can be customized and used to create an application using the `oc create -f <filename>` command, or merged with other resource definition files to create a composite application.

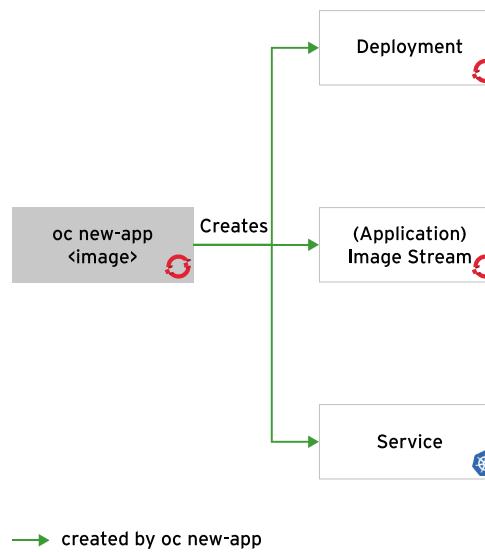
The `oc new-app` command can create application pods to run on OpenShift in many different ways. It can create pods from existing docker images, from Dockerfiles, and from raw source code using the Source-to-Image (S2I) process.

Run the `oc new-app -h` command to understand all the different options available for creating new applications on OpenShift.

The following command creates an application based on an image, `mysql`, from Docker Hub, with the label set to `db=mysql`:

```
[user@host ~]$ oc new-app mysql MYSQL_USER=user MYSQL_PASSWORD=pass
 MYSQL_DATABASE=testdb -l db=mysql
```

The following figure shows the Kubernetes and OpenShift resources created by the `oc new-app` command when the argument is a container image:



**Figure 6.4: Resources created for a new application**

The following command creates an application based on an image from a private Docker image registry:

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

The following command creates an application based on source code stored in a Git repository:

```
oc new-app https://github.com/openshift/ruby-hello-world --name=ruby-hello
```

You will learn more about the Source-to-Image (S2I) process, its associated concepts, and more advanced ways to use `oc new-app` to build applications for OpenShift in the next section.

The following command creates an application based on an existing template:

```
$ oc new-app \
> --template=mysql-persistent \
> -p MYSQL_USER=user1 -p MYSQL_PASSWORD=mypa55 -p MYSQL_DATABASE=testdb \
> -p MYSQL_ROOT_PASSWORD=r00tpa55 -p VOLUME_CAPACITY=10Gi
...output omitted...
```

**Note**

You will learn more about templates on the next chapter.

## Managing Persistent Storage

In addition to the specification of custom images, you can create persistent storage and attach it to your application. In this way you can make sure your data is not lost when deleting your pods. To list the `PersistentVolume` objects in a cluster, use the `oc get pv` command:

```
[admin@host ~]$ oc get pv
NAME CAPACITY ACCESS MODES RECLAIM POLICY STATUS CLAIM ...
pv0001 1Mi RWO Retain Available ...
pv0002 10Mi RWX Recycle Available ...
...output omitted...
```

To see the YAML definition for a given `PersistentVolume`, use the `oc get` command with the `-o yaml` option:

```
[admin@host ~]$ oc get pv pv0001 -o yaml
apiVersion: v1
kind: PersistentVolume
metadata:
 creationTimestamp: ...value omitted...
 finalizers:
 - kubernetes.io/pv-protection
 labels:
 type: local
 name: pv0001
 resourceVersion: ...value omitted...
 selfLink: /api/v1/persistentvolumes/pv0001
 uid: ...value omitted...
spec:
```

```
accessModes:
- ReadWriteOnce
capacity:
 storage: 1Mi
hostPath:
 path: /data/pv0001
 type: ""
persistentVolumeReclaimPolicy: Retain
status:
 phase: Available
```

To add more `PersistentVolume` objects to a cluster, use the `oc create` command:

```
[admin@host ~]$ oc create -f pv1001.yaml
```



### Note

The above `pv1001.yaml` file must contain a persistent volume definition, similar in structure to the output of the `oc get pv pv-name -o yaml` command.

## Requesting Persistent Volumes

When an application requires storage, you create a `PersistentVolumeClaim` (PVC) object to request a dedicated storage resource from the cluster pool. The following content from a file named `pvc.yaml` is an example definition for a PVC:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
 name: myapp
spec:
 accessModes:
 - ReadWriteOnce
 resources:
 requests:
 storage: 1Gi
```

The PVC defines storage requirements for the application, such as capacity or throughput. To create the PVC, use the `oc create` command:

```
[admin@host ~]$ oc create -f pvc.yaml
```

After you create a PVC, OpenShift attempts to find an available `PersistentVolume` resource that satisfies the PVC's requirements. If OpenShift finds a match, it binds the `PersistentVolume` object to the `PersistentVolumeClaim` object. To list the PVCs in a project, use the `oc get pvc` command:

```
[admin@host ~]$ oc get pvc
NAME STATUS VOLUME CAPACITY ACCESS MODES STORAGECLASS AGE
myapp Bound pv0001 1Gi RWO 6s
```

The output indicates whether a persistent volume is bound to the PVC, along with attributes of the PVC (such as capacity).

To use the persistent volume in an application pod, define a volume mount for a container that references the `PersistentVolumeClaim` object. The application pod definition below references a `PersistentVolumeClaim` object to define a volume mount for the application:

```
apiVersion: "v1"
kind: "Pod"
metadata:
 name: "myapp"
 labels:
 name: "myapp"
spec:
 containers:
 - name: "myapp"
 image: openshift/myapp
 ports:
 - containerPort: 80
 name: "http-server"
 volumeMounts:
 - mountPath: "/var/www/html"
 name: "pvol" ①
 volumes:
 - name: "pvol" ②
 persistentVolumeClaim:
 claimName: "myapp" ③
```

- ① This section declares that the `pvol` volume mounts at `/var/www/html` in the container file system.
- ② This section defines the `pvol` volume.
- ③ The `pvol` volume references the `myapp` PVC. If OpenShift associates an available persistent volume to the `myapp` PVC, then the `pvol` volume refers to this associated volume.

## Managing OpenShift Resources at the Command Line

There are several essential commands used to manage OpenShift resources as described below.

Use the `oc get` command to retrieve information about resources in the cluster. Generally, this command outputs only the most important characteristics of the resources and omits more detailed information.

The `oc get RESOURCE_TYPE` command displays a summary of all resources of the specified type. The following illustrates example output of the `oc get pods` command.

| NAME          | READY | STATUS  | RESTARTS | AGE |
|---------------|-------|---------|----------|-----|
| nginx-1-5r583 | 1/1   | Running | 0        | 1h  |
| myapp-1-l44m7 | 1/1   | Running | 0        | 1h  |

## oc get all

Use the `oc get all` command to retrieve a summary of the most important components of a cluster. This command iterates through the major resource types for the current project and prints out a summary of their information:

| NAME             | DOCKER REPO                            |             |                 | TAGS                       | UPDATED           |
|------------------|----------------------------------------|-------------|-----------------|----------------------------|-------------------|
| is/nginx         | 172.30.1.1:5000/basic-kubernetes/nginx |             |                 | latest                     | About an hour ago |
| <br>             |                                        |             |                 |                            |                   |
| NAME             | REVISION                               | DESIRED     | CURRENT         | TRIGGERED BY               |                   |
| dc/nginx         | 1                                      | 1           | 1               | config,image(nginx:latest) |                   |
| <br>             |                                        |             |                 |                            |                   |
| NAME             | DESIRED                                | CURRENT     | READY           | AGE                        |                   |
| rc/nginx-1       | 1                                      | 1           | 1               | 1h                         |                   |
| <br>             |                                        |             |                 |                            |                   |
| NAME             | CLUSTER-IP                             | EXTERNAL-IP | PORT(S)         | AGE                        |                   |
| svc/nginx        | 172.30.72.75                           | <none>      | 80/TCP, 443/TCP | 1h                         |                   |
| <br>             |                                        |             |                 |                            |                   |
| NAME             | READY                                  | STATUS      | RESTARTS        | AGE                        |                   |
| po/nginx-1-ypp8t | 1/1                                    | Running     | 0               | 1h                         |                   |

## oc describe

If the summaries provided by `oc get` are insufficient, use the `oc describe RESOURCE_TYPE RESOURCE_NAME` command to retrieve additional information. Unlike the `oc get` command, there is no way to iterate through all the different resources by type. Although most major resources can be described, this functionality is not available across all resources. The following is an example output from describing a pod resource:

```
Name: mysql-openshift-1-glqrp
Namespace: mysql-openshift
Priority: 0
Node: cluster-worker-1/172.25.250.52
Start Time: Fri, 15 Feb 2019 02:14:34 +0000
Labels: app=mysql-openshift
 deployment=mysql-openshift-1
...output omitted...
Status: Running
IP: 10.129.0.85
```

## oc get

The `oc get RESOURCE_TYPE RESOURCE_NAME` command can be used to export a resource definition. Typical use cases include creating a backup, or to aid in the modification of a definition. The `-o yaml` option prints out the object representation in YAML format, but this can be changed to JSON format by providing a `-o json` option.

## oc create

This command creates resources from a resource definition. Typically, this is paired with the `oc get RESOURCE_TYPE RESOURCE_NAME -o yaml` command for editing definitions.

## oc edit

This command allows the user to edit resources of a resource definition. By default, this command opens a `vi` buffer for editing the resource definition.

## oc delete

The `oc delete RESOURCE_TYPE name` command removes a resource from an OpenShift cluster. Note that a fundamental understanding of the OpenShift architecture is needed here, because deleting managed resources such as pods results in new instances of those resources being automatically created. When a project is deleted, it deletes all of the resources and applications contained within it.

## oc exec

The `oc exec CONTAINER_ID options` command executes commands inside a container. You can use this command to run interactive and noninteractive batch commands as part of a script.

## Labelling Resources

When working with many resources in the same project, it is often useful to group those resources by application, environment, or some other criteria. To establish these groups, you define labels for the resources in your project. Labels are part of the `metadata` section of a resource, and are defined as key/value pairs, as shown in the following example:

```
apiVersion: v1
kind: Service
metadata:
 ...contents omitted...
 labels: app: nexus template: nexus-persistent-template
 name: nexus
 ...contents omitted...
```

Many `oc` subcommands support a `-l` option to process resources from a label specification. For the `oc get` command, the `-l` option acts as a selector to only retrieve objects that have a matching label:

```
$ oc get svc,deployments -l app=nexus
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
service/nexus ClusterIP 172.30.29.218 <none> 8081/TCP 4h

NAME REVISION DESIRED CURRENT ...
deployment.apps.openshift.io/nexus 1 1 1 ...
```



### Note

Although any label can appear in resources, both the `app` and `template` keys are common for labels. By convention, the `app` key indicates the application related to this resource. The `template` key labels all resources generated by the same template with the template's name.

When using templates to generate resources, labels are especially useful. A template resource has a `labels` section separated from the `metadata.labels` section. Labels defined in the `labels`

section do not apply to the template itself, but are added to every resource generated by the template:

```
apiVersion: template.openshift.io/v1
kind: Template
labels: app: nexus template: nexus-persistent-template
metadata:
...contents omitted...
labels: maintainer: redhat
 name: nexus-persistent
...contents omitted...
objects:
- apiVersion: v1
 kind: Service
 metadata:
 name: nexus
labels: version: 1
...contents omitted...
```

The previous example defines a template resource with a single label: `maintainer: redhat`. The template generates a service resource with three labels: `app: nexus`, `template: nexus-persistent-template`, and `version: 1`.



## References

Additional information about pods and services is available in the *Pods and Services* section of the OpenShift Container Platform documentation:

### Architecture

[https://access.redhat.com/documentation/en-us/openshift\\_container\\_platform/4.6/html-single/architecture/index](https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6/html-single/architecture/index)

Additional information about creating images is available in the OpenShift Container Platform documentation:

### Creating Images

[https://access.redhat.com/documentation/en-us/openshift\\_container\\_platform/4.6/html-single/images/index](https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6/html-single/images/index)

Labels and label selectors details are available in *Working with Kubernetes Objects* section for the Kubernetes documentation:

### Labels and Selectors

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

## ► Guided Exercise

# Deploying a Database Server on OpenShift

In this exercise, you will create and deploy a MySQL database pod on OpenShift using the `oc new-app` command.

## Outcomes

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

## Before You Begin

Make sure you have completed *Guided Exercise: Configuring the Classroom Environment* from *Chapter 1* before executing any command of this practice.

On workstation, run the following command to set up the environment:

```
[student@workstation ~]$ lab openshift-resources 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 OpenShift cluster.

```
[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 that contains your RHOCP developer username for the resources you create during this exercise:

```
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-mysql-openshift
Now using project ...output omitted...
```

### ► 2. Create a new application from the mysql-persistent template using the `oc new-app` command.

This image requires that you use the `-p` option to set the `MYSQL_USER`, `MYSQL_PASSWORD`, `MYSQL_DATABASE`, `MYSQL_ROOT_PASSWORD` and `VOLUME_CAPACITY` environment variables.

Use the `--template` option with the `oc new-app` command to specify a template with persistent storage so that OpenShift does not try and pull the image from the internet:

```
[student@workstation ~]$ oc new-app \
> --template=mysql-persistent \
> -p MYSQL_USER=user1 -p MYSQL_PASSWORD=mypa55 -p MYSQL_DATABASE=testdb \
> -p MYSQL_ROOT_PASSWORD=r00tpa55 -p VOLUME_CAPACITY=10Gi
--> Deploying template "openshift/mysql-persistent" to project
${RHT_OCP4_DEV_USER}-mysql-openshift
...output omitted...
--> Creating resources ...
 secret "mysql" created
 service "mysql" created
 persistentvolumeclaim "mysql" created
 deploymentconfig.apps.openshift.io "mysql" created
--> Success
 Application is not exposed. You can expose services to the outside world by
executing one or more of the commands below:
 'oc expose service/mysql'
Run 'oc status' to view your app.
```

- ▶ 3. Verify that the MySQL pod was created successfully and view the details about the pod and its service.
- 3.1. Run the `oc status` command to view the status of the new application and verify that the deployment of the MySQL image was successful:

```
[student@workstation ~]$ oc status
In project ${RHT_OCP4_DEV_USER}-mysql-openshift on server ...

svc/mysql - 172.30.151.91:3306
...output omitted...
deployment #1 deployed 6 minutes ago - 1 pod
```

- 3.2. List the pods in this project to verify that the MySQL pod is ready and running:

```
[student@workstation ~]$ oc get pods
NAME READY STATUS RESTARTS AGE
mysql-1-5vfn4 1/1 Running 0 109s
```

**Note**

Notice the name of the running pod. You need this information to be able to log in to the MySQL database server later.

- 3.3. Use the `oc describe` command to view more details about the pod:

```
[student@workstation ~]$ oc describe pod mysql-1-5vfn4
Name: mysql-1-5vfn4
Namespace: ${RHT_OCP4_DEV_USER}-mysql-openshift
Priority: 0
Node: master01/192.168.50.10
Start Time: Mon, 29 Mar 2021 16:42:13 -0400
Labels: deployment=mysql-1
...output omitted...
Status: Running
IP: 10.10.0.34
...output omitted...
```

- 3.4. List the services in this project and verify that the service to access the MySQL pod was created:

```
[student@workstation ~]$ oc get svc
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
mysql ClusterIP 172.30.151.91 <none> 3306/TCP 10m
```

- 3.5. Retrieve the details of the `mysql` service using the `oc describe` command and note that the service type is `ClusterIP` by default:

```
[student@workstation ~]$ oc describe service mysql
Name: mysql
Namespace: ${RHT_OCP4_DEV_USER}-mysql-openshift
Labels: app=mysql-persistent
 app.kubernetes.io/component=mysql-persistent
 app.kubernetes.io/instance=mysql-persistent
 template=mysql-persistent-template
Annotations: openshift.io/generated-by: OpenShiftNewApp
...output omitted...
Selector: name=mysql
Type: ClusterIP
IP: 172.30.151.91
Port: 3306-tcp 3306/TCP
TargetPort: 3306/TCP
Endpoints: 10.10.0.34:3306
Session Affinity: None
Events: <none>
```

- 3.6. List the persistent storage claims in this project:

```
[student@workstation ~]$ oc get pvc
NAME STATUS VOLUME CAPACITY ...
STORAGECLASS
mysql Bound pvc-e9bf0b1f-47df-4500-afb6-77e826f76c15 10Gi ...
standard
```

- 3.7. Retrieve the details of the `mysql` pvc using the `oc describe` command:

```
[student@workstation ~]$ oc describe pvc/mysql
Name: mysql
Namespace: ${RHT_OCP4_DEV_USER}-mysql-openshift
StorageClass: standard
Status: Bound
Volume: pvc-e9bf0b1f-47df-4500-afb6-77e826f76c15
Labels: app=mysql-persistent
 app.kubernetes.io/component=mysql-persistent
 app.kubernetes.io/instance=mysql-persistent
 template=mysql-persistent-template
Annotations: openshift.io/generated-by: OpenShiftNewApp
...output omitted...
Capacity: 10Gi
Access Modes: RWO
VolumeMode: Filesystem
Mounted By: mysql-1-5vfn4
```

- ▶ 4. Connect to the MySQL database server and verify that the database was created successfully.
- 4.1. From the workstation machine, configure port forwarding between workstation and the database pod running on OpenShift using port 3306. The terminal will hang after executing the command.

```
[student@workstation ~]$ oc port-forward mysql-1-5vfn4 3306:3306
Forwarding from 127.0.0.1:3306 -> 3306
Forwarding from [::1]:3306 -> 3306
```

- 4.2. From the workstation machine open another terminal and connect to the MySQL server using the MySQL client.

```
[student@workstation ~]$ mysql -uuser1 -pmypa55 --protocol tcp -h localhost
mysql: [Warning] Using a password on the command line interface can be insecure.
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 13
Server version: 8.0.21 Source distribution

Copyright (c) 2000, 2021, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective

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

mysql>
```

- 4.3. Verify the creation of the testdb database.

```
mysql> show databases;

| Database |

| information_schema |
| testdb |

2 rows in set (0.00 sec)
```

#### 4.4. Exit from the MySQL prompt:

```
mysql> exit
Bye
```

Close the terminal and return to the previous one. Finish the port forwarding process by pressing **Ctrl+C**.

```
Forwarding from 127.0.0.1:3306 -> 3306
Forwarding from [::1]:3306 -> 3306
Handling connection for 3306
^C
```

#### ► 5. Delete the project to remove all the resources within the project:

```
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-mysql-openshift
```

## Finish

On workstation, run the `lab openshift-resources finish` script to complete this lab.

```
[student@workstation ~]$ lab openshift-resources finish
```

This concludes the exercise.

# Creating Routes

---

## Objectives

After completing this section, students should be able to expose services using OpenShift routes.

## Working with Routes

Services allow for network access between pods inside an OpenShift instance, and routes allow for network access to pods from users and applications outside the OpenShift instance.

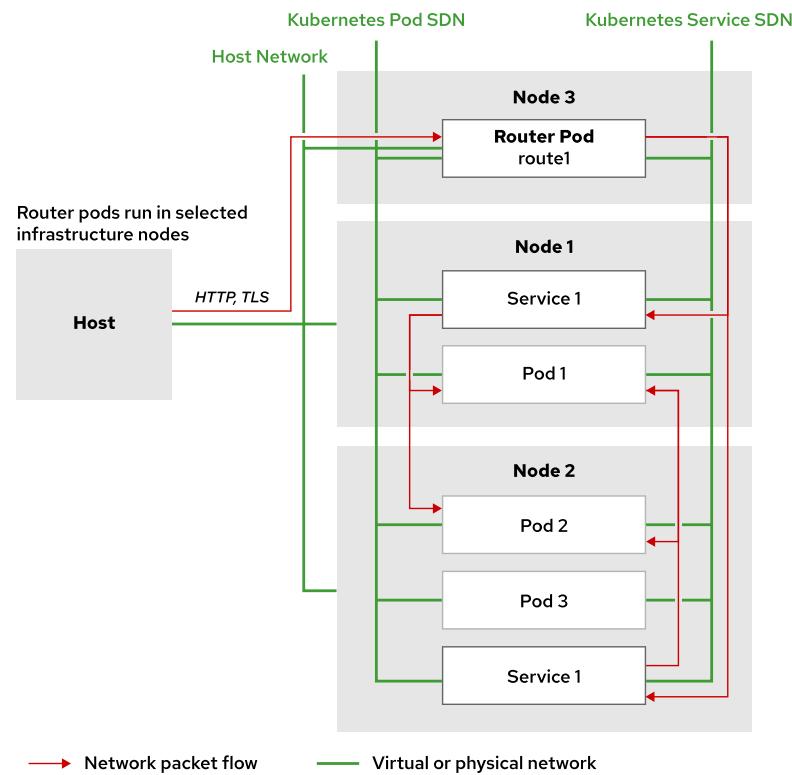


Figure 6.5: OpenShift routes and Kubernetes services

A route connects a public-facing IP address and DNS host name to an internal-facing service IP. It uses the service resource to find the endpoints; that is, the ports exposed by the service.

OpenShift routes are implemented by a cluster-wide router service, which runs as a containerized application in the OpenShift cluster. OpenShift scales and replicates router pods like any other OpenShift application.

**Note**

In practice, to improve performance and reduce latency, the OpenShift router connects directly to the pods using the internal pod software-defined network (SDN).

The router service uses *HAProxy* as the default implementation.

An important consideration for OpenShift administrators is that the public DNS host names configured for routes need to point to the public-facing IP addresses of the nodes running the router. Router pods, unlike regular application pods, bind to their nodes' public IP addresses instead of to the internal pod SDN.

The following example shows a minimal route defined using JSON syntax:

```
{
 "apiVersion": "v1",
 "kind": "Route",
 "metadata": {
 "name": "quoteapp"
 },
 "spec": {
 "host": "quoteapp.apps.example.com",
 "to": {
 "kind": "Service",
 "name": "quoteapp"
 }
 }
}
```

The `apiVersion`, `kind`, and `metadata` attributes follow standard Kubernetes resource definition rules. The `Route` value for `kind` shows that this is a route resource, and the `metadata.name` attribute gives this particular route the identifier `quoteapp`.

As with pods and services, the main part is the `spec` attribute, which is an object containing the following attributes:

- `host` is a string containing the FQDN associated with the route. DNS must resolve this FQDN to the IP address of the OpenShift router. The details to modify DNS configuration are outside the scope of this course.
- `to` is an object stating the resource this route points to. In this case, the route points to an OpenShift Service with the name set to `quoteapp`.

**Note**

Names of different resource types do not collide. It is perfectly legal to have a route named `quoteapp` that points to a service also named `quoteapp`.

**Important**

Unlike services, which use selectors to link to pod resources containing specific labels, a route links directly to the service resource name.

## Creating Routes

Use the `oc create` command to create route resources, just like any other OpenShift resource. You must provide a JSON or YAML resource definition file, which defines the route, to the `oc create` command.

The `oc new-app` command does not create a route resource when building a pod from container images, Dockerfiles, or application source code. After all, `oc new-app` does not know if the pod is intended to be accessible from outside the OpenShift instance or not.

Another way to create a route is to use the `oc expose service` command, passing a service resource name as the input. The `--name` option can be used to control the name of the route resource. For example:

```
$ oc expose service quotedb --name quote
```

By default, routes created by `oc expose` generate DNS names of the form: `route-name-project-name.default-domain`

Where:

- `route-name` is the name assigned to the route. If no explicit name is set, OpenShift assigns the route the same name as the originating resource (for example, the service name).
- `project-name` is the name of the project containing the resource.
- `default-domain` is configured on the OpenShift Control Plane and corresponds to the wildcard DNS domain listed as a prerequisite for installing OpenShift.

For example, creating a route named `quote` in project named `test` from an OpenShift instance where the wildcard domain is `cloudapps.example.com` results in the FQDN `quote-test.cloudapps.example.com`.



### Note

The DNS server that hosts the wildcard domain knows nothing about route host names. It merely resolves any name to the configured IP addresses. Only the OpenShift router knows about route host names, treating each one as an HTTP virtual host. The OpenShift router blocks invalid wildcard domain host names that do not correspond to any route and returns an HTTP 404 error.

## Leveraging the Default Routing Service

The default routing service is implemented as an HAProxy pod. Router pods, containers, and their configuration can be inspected just like any other resource in an OpenShift cluster:

```
$ oc get pod --all-namespaces | grep router
openshift-ingress router-default-746b5cfb65-f6sdm 1/1 Running 1 4d
```

Note that you can query information on the default router using the associated label as shown here.

By default, router is deployed in `openshift-ingress` project. Use `oc describe pod` command to get the routing configuration details:

```
$ oc describe pod router-default-746b5cfb65-f6sdm
Name: router-default-746b5cfb65-f6sdm
Namespace: openshift-ingress
...output omitted...
Containers:
 router:
 ...output omitted...
 Environment:
 STATS_PORT: 1936
 ROUTER_SERVICE_NAMESPACE: openshift-ingress
 DEFAULT_CERTIFICATE_DIR: /etc/pki/tls/private
 ROUTER_SERVICE_NAME: default
 ROUTER_CANONICAL_HOSTNAME: apps.cluster.lab.example.com
...output omitted...
```

The subdomain, or default domain to be used in all default routes, takes its value from the ROUTER\_CANONICAL\_HOSTNAME entry.



## References

Additional information about the architecture of routes in OpenShift is available in the *Architecture* and *Developer Guide* sections of the **OpenShift Container Platform documentation**.

[https://access.redhat.com/documentation/en-us/openshift\\_container\\_platform/4.6/](https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6/)

## ► Guided Exercise

# Exposing a Service as a Route

In this exercise, you will create, build, and deploy an application on an OpenShift cluster and expose its service as a route.

## Outcomes

You should be able to expose a service as a route for a deployed OpenShift application.

## Before You Begin

Make sure you have completed *Guided Exercise: Configuring the Classroom Environment* from Chapter 1 before executing any command of this practice.

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

```
[student@workstation ~]$ lab openshift-routes 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 OpenShift cluster.

```
[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 that contains your RHOCP developer username for the resources you create during this exercise.

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

► 2. Create a new PHP application using the `quay.io/redhattraining/php-hello-dockerfile` image.

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

**Important**

The following example uses a backslash (\) to indicate that the second line is a continuation of the first line. If you wish to ignore the backslash, you can type the entire command in one line.

```
[student@workstation ~]$ oc new-app \
> --docker-image=quay.io/redhattraining/php-hello-dockerfile \
> --name php-helloworld
--> Found container image 4b696cc (2 years old) from quay.io for "quay.io/
redhattraining/php-hello-dockerfile"
...output omitted...
--> Creating resources ...
...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/php-helloworld'
Run 'oc status' to view your app.
```

- 2.2. Wait until the application finishes deploying by monitoring the progress with the `oc get pods -w` command:

```
[student@workstation ~]$ oc get pods -w
NAME READY STATUS RESTARTS AGE
php-helloworld-74bb86f6cb-zt6wl 1/1 Running 0 5s
^C
```

Your exact output may differ in names, status, timing, and order. The container in `Running` status with a random suffix (`74bb86f6cb-zt6wl` in the example) contains the application and shows it is up and running. Alternatively, monitor the deployment logs with the `oc logs -f php-helloworld-74bb86f6cb-zt6wl` command. Press `Ctrl + C` to exit the command if necessary.

```
[student@workstation ~]$ oc logs -f php-helloworld-74bb86f6cb-zt6wl
AH00558: httpd: Could not reliably determine the server's fully qualified domain
name, using 10.129.5.124. Set the 'ServerName' directive globally to suppress
this message
[09-Aug-2021 22:09:45] NOTICE: [pool www] 'user' directive is ignored when FPM is
not running as root
[09-Aug-2021 22:09:45] NOTICE: [pool www] 'group' directive is ignored when FPM is
not running as root
`^C`
```

Your exact output may differ.

- 2.3. Review the service for this application using the `oc describe` command:

```
[student@workstation ~]$ oc describe svc/php-helloworld
Name: php-helloworld
Namespace: extrdp-route
```

```

Labels: app=php-helloworld
 app.kubernetes.io/component=php-helloworld
 app.kubernetes.io/instance=php-helloworld
Annotations: openshift.io/generated-by: OpenShiftNewApp
Selector: deployment=php-helloworld
Type: ClusterIP
IP: 172.30.100.236
Port: 8080-tcp 8080/TCP
TargetPort: 8080/TCP
Endpoints: 10.129.5.124:8080
Session Affinity: None
Events: <none>

```

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

- ▶ 3. Expose the service, which creates a route. Use the default name and fully qualified domain name (FQDN) for the route:

```

[student@workstation ~]$ oc expose svc/php-helloworld
route.route.openshift.io/php-helloworld exposed
[student@workstation ~]$ oc describe route
Name: php-helloworld
Namespace: extrdp-route
Created: 16 seconds ago
Labels: app=php-helloworld
 app.kubernetes.io/component=php-helloworld
 app.kubernetes.io/instance=php-helloworld
Annotations: openshift.io/host.generated=true
Requested Host: php-helloworld-extrdp-route.apps.na46-stage2.dev.nextcle.com
 exposed on router default (host apps.na46-
stage2.dev.nextcle.com) 16 seconds ago
Path: <none>
TLS Termination: <none>
Insecure Policy: <none>
Endpoint Port: 8080-tcp

Service: php-helloworld
Weight: 100 (100%)
Endpoints: 10.129.5.124:8080

```

- 4. Access the service from a host external to the cluster to verify that the service and route are working.

```
[student@workstation ~]$ curl \
> php-helloworld-${RHT_OCP4_DEV_USER}-route.${RHT_OCP4_WILDCARD_DOMAIN}
Hello, World! PHP version is 7.2.11
```

**Note**

The output of the PHP application depends on the actual image version. It may be different from yours.

Notice the FQDN is comprised of the application name and project name by default. The remainder of the FQDN, the subdomain, is defined when OpenShift is installed.

- 5. Replace this route with a route named xyz.

5.1. Delete the current route:

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

**Note**

Deleting the route is optional. You can have multiple routes for the same service, provided they have different names.

5.2. Create a route for the service with a name of \${RHT\_OCP4\_DEV\_USER}-xyz.

```
[student@workstation ~]$ oc expose svc/php-helloworld \
> --name=${RHT_OCP4_DEV_USER}-xyz
route.route.openshift.io/${RHT_OCP4_DEV_USER}-xyz exposed
[student@workstation ~]$ oc describe route
Name: extrdp-xyz
Namespace: extrdp-route
Created: 23 seconds ago
Labels: app=php-helloworld
 app.kubernetes.io/component=php-helloworld
 app.kubernetes.io/instance=php-helloworld
Annotations: openshift.io/host.generated=true
Requested Host:extrdp-xyz-extrdp-route.apps.na46-stage2.dev.nextcle.com
 exposed on router default (host apps.na46-
stage2.dev.nextcle.com) 22 seconds ago
Path: <none>
TLS Termination: <none>
Insecure Policy: <none>
Endpoint Port: 8080-tcp

Service: php-helloworld
Weight: 100 (100%)
Endpoints: 10.129.5.124:8080
```

Your exact output may differ. Note the new FQDN that was generated based on the new route name. Both the route name and the project name contain your user name, hence it appears twice in the route FQDN.

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

```
[student@workstation ~]$ curl \
> ${RHT_OCP4_DEV_USER}-xyz-${RHT_OCP4_DEV_USER}-route.${RHT_OCP4_WILDCARD_DOMAIN}
Hello, World! PHP version is 7.2.11
```

## Finish

On `workstation`, run the `lab openshift-routes finish` script to complete this exercise.

```
[student@workstation ~]$ lab openshift-routes finish
```

This concludes the guided exercise.

# Creating Applications with Source-to-Image

---

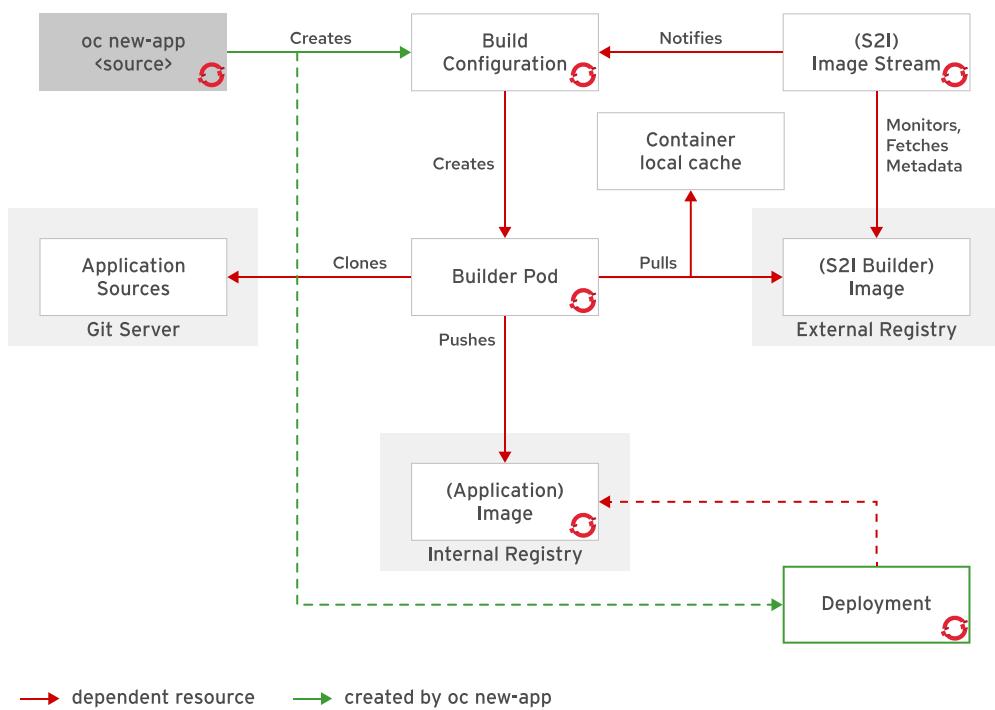
## Objectives

After completing this section, students should be able to deploy an application using the Source-to-Image (S2I) facility of OpenShift Container Platform.

## The Source-to-Image (S2I) Process

Source-to-Image (S2I) is a tool that makes it easy to build container images from application source code. This tool takes an application's source code from a Git repository, injects the source code into a base container based on the language and framework desired, and produces a new container image that runs the assembled application.

This figure shows the resources created by the `oc new-app` command when the argument is an application source code repository. Notice that S2I also creates a Deployment and all its dependent resources:



**Figure 6.6: Deployment and dependent resources**

S2I is the primary strategy used for building applications in OpenShift Container Platform. The main reasons for using source builds are:

- User efficiency: Developers do not need to understand Dockerfiles and operating system commands such as `yum install`. They work using their standard programming language tools.

- Patching: S2I allows for rebuilding all the applications consistently if a base image needs a patch due to a security issue. For example, if a security issue is found in a PHP base image, then updating this image with security patches updates all applications that use this image as a base.
- Speed: With S2I, the assembly process can perform a large number of complex operations without creating a new layer at each step, resulting in faster builds.
- Ecosystem: S2I encourages a shared ecosystem of images where base images and scripts can be customized and reused across multiple types of applications.

## Describing Image Streams

OpenShift deploys new versions of user applications into pods quickly. To create a new application, in addition to the application source code, a base image (the S2I builder image) is required. If either of these two components gets updated, OpenShift creates a new container image. Pods created using the older container image are replaced by pods using the new image.

Even though it is evident that the container image needs to be updated when application code changes, it may not be evident that the deployed pods also need to be updated should the builder image change.

The image stream resource is a configuration that names specific container images associated with image stream tags, an alias for these container images. OpenShift builds applications against an image stream. The OpenShift installer populates several image streams by default during installation. To determine available image streams, use the `oc get` command, as follows:

```
$ oc get is -n openshift
NAME IMAGE REPOSITORY TAGS
cli ...svc:5000/openshift/cli latest
dotnet ...svc:5000/openshift/dotnet 2.1,...,3.1-el7,latest
dotnet-runtime ...svc:5000/openshift/dotnet-runtime 2.1,...,3.1-el7,latest
httpd ...svc:5000/openshift/httpd 2.4,2.4-el7,2.4-el8,latest
jenkins ...svc:5000/openshift/jenkins 2,latest
mariadb ...svc:5000/openshift/mariadb 10.3,10.3-el7,10.3-el8,latest
mongodb ...svc:5000/openshift/mongodb 3.6,latest
mysql ...svc:5000/openshift/mysql 8.0,8.0-el7,8.0-el8,latest
nginx ...svc:5000/openshift/nginx 1.10,1.12,1.8,latest
nodejs ...svc:5000/openshift/nodejs 10,...,12-ubi7,12-ubi8
perl ...svc:5000/openshift/perl 5.26,...,5.30,5.30-el7
php ...svc:5000/openshift/php 7.2-ubi8,...,7.3-ubi8,latest
postgresql ...svc:5000/openshift/postgresql 10,10-el7,...,12-el7,12-el8
python ...svc:5000/openshift/python 2.7,2.7-ubi7,...,3.6-ubi8,3.8
redis ...svc:5000/openshift/redis 5,5-el7,5-el8,latest
ruby ...svc:5000/openshift/ruby 2.5,2.5-ubi7,...,2.6,2.6-ubi7
...
```



### Note

Your OpenShift instance may have more or fewer image streams depending on local additions and OpenShift point releases.

OpenShift detects when an image stream changes and takes action based on that change. If a security issue arises in the `rhel8/nodejs-10` image, it can be updated in the image repository, and OpenShift can automatically trigger a new build of the application code.

It is likely that an organization chooses several supported base S2I images from Red Hat, but may also create their own base images.

## Building an Application with S2I and the CLI

Building an application with S2I can be accomplished using the OpenShift CLI.

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

```
$ oc new-app php~http://my.git.server.com/my-app ①②
--name=myapp ③
```

- ① The image stream used in the process appears to the left of the tilde (~).
- ② The URL after the tilde indicates the location of the source code's Git repository.
- ③ Sets the application name.



### Note

Instead of using the tilde, you can set the image stream by using the `-i` option or `--image-stream` for the full version.

```
$ oc new-app -i php http://services.lab.example.com/app --name=myapp
```

The `oc new-app` command allows creating applications using source code from a local or remote Git repository. If only a source repository is specified, `oc new-app` tries to identify the correct image stream to use for building the application. In addition to application code, S2I can also identify and process Dockerfiles to create a new image.

The following example creates an application using the Git repository in the current directory:

```
$ oc new-app .
```



### Important

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

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

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

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

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

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

| Language | Files                                   |
|----------|-----------------------------------------|
| Ruby     | <code>Rakefile Gemfile config.ru</code> |
| Java EE  | <code>pom.xml</code>                    |
| Node.js  | <code>app.json package.json</code>      |
| PHP      | <code>index.php composer.json</code>    |
| Python   | <code>requirements.txt config.py</code> |
| Perl     | <code>index.pl cpanfile</code>          |

After a language is detected, the new-app command searches for image stream tags that have support for the detected language, or an image stream that matches the name of the detected language.

Create a JSON resource definition file by using the -o json parameter and output redirection:

```
$ oc -o json new-app php-http://services.lab.example.com/app \
> --name=myapp > s2i.json
```

This JSON definition file creates a list of resources. The first resource is the image stream:

```
...output omitted...
{
 "kind": "ImageStream", ❶
 "apiVersion": "image.openshift.io/v1",
 "metadata": {
 "name": "myapp", ❷
 "creationTimestamp": null
 "labels": {
 "app": "myapp"
 },
 "annotations": {
 "openshift.io/generated-by": "OpenShiftNewApp"
 }
 },
 "spec": {
 "lookupPolicy": {
 "local": false
 }
 },
 "status": {
 "dockerImageRepository": ""
 }
},
...output omitted...
```

- ❶ Define a resource type of image stream.

- ❷ Name the image stream `myapp`.

The build configuration (`bc`) is responsible for defining input parameters and triggers that are executed to transform the source code into a runnable image. The `BuildConfig` (`BC`) is the second resource, and the following example provides an overview of the parameters used by OpenShift to create a runnable image.

```
...output omitted...
{
 "kind": "BuildConfig", ❶
 "apiVersion": "build.openshift.io/v1",
 "metadata": {
 "name": "myapp", ❷
 "creationTimestamp": null,
 "labels": {
 "app": "myapp"
 },
 "annotations": {
 "openshift.io/generated-by": "OpenShiftNewApp"
 }
 },
 "spec": {
 "triggers": [
 {
 "type": "GitHub",
 "github": {
 "secret": "S5_4BZpPabM6KrIuPBvI"
 }
 },
 {
 "type": "Generic",
 "generic": {
 "secret": "3q8K8JNDoRzhjoz1KgMz"
 }
 },
 {
 "type": "ConfigChange"
 },
 {
 "type": "ImageChange",
 "imageChange": {}
 }
],
 "source": {
 "type": "Git",
 "git": {
 "uri": "http://services.lab.example.com/app" ❸
 }
 },
 "strategy": {
 "type": "Source", ❹
 "sourceStrategy": {
 "from": {
 "kind": "ImageStreamTag",
 "namespace": "openshift",
 }
 }
 }
 }
}
```

```

 "name": "php:7.3" ⑤
 }
}
},
"output": {
 "to": {
 "kind": "ImageStreamTag",
 "name": "myapp:latest" ⑥
 }
},
"resources": {},
"postCommit": {},
"nodeSelector": null
},
"status": {
 "lastVersion": 0
}
},
...output omitted...

```

- ① Define a resource type of `BuildConfig`.
- ② Name the `BuildConfig` `myapp`.
- ③ Define the address to the source code Git repository.
- ④ Define the strategy to use S2I.
- ⑤ Define the builder image as the `php:7.3` image stream.
- ⑥ Name the output image stream `myapp:latest`.

The third resource is the deployment object that is responsible for customizing the deployment process in OpenShift. It may include parameters and triggers that are necessary to create new container instances, and are translated into a replication controller from Kubernetes. Some of the features provided by Deployment objects are:

- User customizable strategies to transition from the existing deployments to new deployments.
- Have as many active replica set as wanted and possible.
- Replication scaling depends of the sizes of old and new replica sets.

```

...output omitted...
{
 "kind": "Deployment", ①
 "apiVersion": "apps/v1",
 "metadata": {
 "name": "myapp", ②
 "creationTimestamp": null,
 "labels": {
 "app": "myapp",
 "app.kubernetes.io/component": "myapp",
 "app.kubernetes.io/instance": "myapp"
 },
 "annotations": {

```

```

 "image.openshift.io/triggers": "[{\\"from\\":{\\\"kind\\":
 \\\"ImageStreamTag\\\",\\\"name\\\":\\\"myapp:1
 atest\\\"},\\\"fieldPath\\\":\\\"spec.template.spec.containers[?(@.name==\\\"myapp\\
 \\\")].image\\\"}]", ❸
 "openshift.io/generated-by": "OpenShiftNewApp"
 }
 },
 "spec": {
 "replicas": 1,
 "selector": {
 "matchLabels": {
 "deployment": "myapp"
 }
 },
 "template": {
 "metadata": {
 "creationTimestamp": null,
 "labels": {
 "deployment": "myapp" ❹
 },
 "annotations": {
 "openshift.io/generated-by": "OpenShiftNewApp"
 }
 },
 "spec": {
 "containers": [
 {
 "name": "myapp", ❺
 "image": " ", ❻
 "ports": [
 {
 "containerPort": 8080,
 "protocol": "TCP"
 },
 {
 "containerPort": 8443,
 "protocol": "TCP"
 }
],
 "resources": {}
 }
]
 },
 "strategy": {}
 },
 "status": {}
 },
 ...output omitted...

```

- ❶ Define a resource type of Deployment.
- ❷ Name the Deployment myapp.

- ③ An image change trigger causes the creation of a new deployment each time a new version of the `myapp:latest` image is available in the repository.
- ④ A configuration change trigger causes a new deployment to be created any time the replication controller template changes.
- ⑤ Defines the container image to deploy: `myapp:latest`.
- ⑥ Specifies the container ports. A configuration change trigger causes a new deployment to be created any time the replication controller template changes.

The last item is the service, already covered in previous chapters:

```
...output omitted...
{
 "kind": "Service",
 "apiVersion": "v1",
 "metadata": {
 "name": "myapp",
 "creationTimestamp": null,
 "labels": {
 "app": "myapp"
 "app.kubernetes.io/component": "myapp",
 "app.kubernetes.io/instance": "myapp"
 },
 "annotations": {
 "openshift.io/generated-by": "OpenShiftNewApp"
 }
 },
 "spec": {
 "ports": [
 {
 "name": "8080-tcp",
 "protocol": "TCP",
 "port": 8080,
 "targetPort": 8080
 },
 {
 "name": "8443-tcp",
 "protocol": "TCP",
 "port": 8443,
 "targetPort": 8443
 }
],
 "selector": {
 "deployment": "myapp"
 }
 },
 "status": {
 "loadBalancer": {}
 }
}
```

**Note**

By default, the `oc new-app` command does not create a route. You can create a route after creating the application. However, a route is automatically created when using the web console because it uses a template.

After creating a new application, the build process starts. Use the `oc get builds` command to see a list of application builds:

```
$ oc get builds
NAME TYPE FROM STATUS STARTED DURATION
php-helloworld-1 Source Git@9e17db8 Running 13 seconds ago
```

OpenShift allows viewing the build logs. The following command shows the last few lines of the build log:

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

**Important**

If the build is not Running yet, or OpenShift has not deployed the `s2i-build` pod yet, the above command throws an error. Just wait a few moments and retry it.

Trigger a new build with the `oc start-build build_config_name` command:

```
$ oc get buildconfig
NAME TYPE FROM LATEST
myapp Source Git 1
```

```
$ oc start-build myapp
build "myapp-2" started
```

## Relationship Between Build and Deployment

The `BuildConfig` pod is responsible for creating the images in OpenShift and pushing them to the internal container registry. Any source code or content update typically requires a new build to guarantee the image is updated.

The `Deployment` pod is responsible for deploying pods to OpenShift. The outcome of a `Deployment` pod execution is the creation of pods with the images deployed in the internal container registry. Any existing running pod may be destroyed, depending on how the `Deployment` resource is set.

The `BuildConfig` and `Deployment` resources do not interact directly. The `BuildConfig` resource creates or updates a container image. The `Deployment` reacts to this new image or updated image event and creates pods from the container image.



## References

### Source-to-Image (S2I) Build

[https://access.redhat.com/documentation/en-us/openshift\\_container\\_platform/4.6/html-single/builds/build-strategies#builds-strategy-s2i-build\\_understanding-image-builds](https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6/html-single/builds/build-strategies#builds-strategy-s2i-build_understanding-image-builds)

### S2I GitHub repository

<https://github.com/openshift/source-to-image>

## ► Guided Exercise

# Creating a Containerized Application with Source-to-Image

In this exercise, you will build an application from source code and deploy the application to an OpenShift cluster.

## Outcomes

You should be able to:

- Build an application from source code using the OpenShift command-line interface.
- Verify the successful deployment of the application using the OpenShift command-line interface.

## Before You Begin

Make sure you have completed *Guided Exercise: Configuring the Classroom Environment* from *Chapter 1* before executing any command of this practice.

Run the following command to download the relevant lab files and configure the environment:

```
[student@workstation ~]$ lab openshift-s2i start
```

## Instructions

- 1. Inspect the PHP source code for the sample application and create and push a new branch named s2i to use during this exercise.
- 1.1. Enter your local clone of the D0180-apps Git repository and checkout the master branch of the course's repository to ensure you start this exercise from a known good state:

```
[student@workstation openshift-s2i]$ cd ~/D0180-apps
[student@workstation D0180-apps]$ git checkout master
...output omitted...
```

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

```
[student@workstation D0180-apps]$ git checkout -b s2i
Switched to a new branch 's2i'
[student@workstation D0180-apps]$ git push -u origin s2i
...output omitted...
* [new branch] s2i -> s2i
Branch 's2i' set up to track remote branch 's2i' from 'origin'.
```

13. Review the PHP source code of the application, inside the the `php-helloworld` folder.  
Open the `index.php` file in the `/home/student/D0180-apps/php-helloworld` folder:

```
<?php
print "Hello, World! php version is " . PHP_VERSION . "\n";
?>
```

The application implements a simple response which returns the PHP version it is running.

► **2.** Prepare the lab environment.

- 2.1. Load your classroom environment configuration.

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

```
[student@workstation D0180-apps]$ source /usr/local/etc/ocp4.config
```

- 2.2. Log in to the OpenShift cluster.

```
[student@workstation D0180-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 that contains your RHOCP developer username for the resources you create during this exercise:

```
[student@workstation D0180-apps]$ oc new-project ${RHT_OCP4_DEV_USER}-s2i
```

► **3.** Create a new PHP application using Source-to-Image from the `php-helloworld` directory using the `s2i` branch you created in the previous step in your fork of the D0180-apps Git repository.

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



**Important**

The following example uses the number sign (#) to select a specific branch from the git repository, in this case the `s2i` branch created in the previous step.

```
[student@workstation D0180-apps]$ oc new-app php:7.3 --name=php-helloworld \
> https://github.com/${RHT_OCP4_GITHUB_USER}/D0180-apps#s2i \
> --context-dir php-helloworld
```

- 3.2. Wait for the build to complete and the application to deploy. Verify that the build process starts with the `oc get pods` command.

```
[student@workstation openshift-s2i]$ oc get pods
NAME READY STATUS RESTARTS AGE
php-helloworld-1-build 1/1 Running 0 5s
```

- 3.3. Examine the logs for this build. Use the build pod name for this build, `php-helloworld-1-build`.

```
[student@workstation D0180-apps]$ oc logs --all-containers \
> -f php-helloworld-1-build
...output omitted...

Writing manifest to image destination
Storing signatures
Generating dockerfile with builder image image-registry.openshift-image-...
php@sha256:3206...37b4
Adding transient rw bind mount for /run/secrets/rhsm
STEP 1: FROM image-registry.openshift-image-registry.svc:5000...

...output omitted...

STEP 8: RUN /usr/libexec/s2i/assemble

...output omitted...

Pushing image .../php-helloworld:latest ...
Getting image source signatures
...output omitted...

Writing manifest to image destination
Storing signatures
Successfully pushed .../php-helloworld@sha256:3f1c...c454
Push successful
Cloning "https://github.com/${RHT_OCP4_GITHUB_USER}/D0180-apps" ...
Commit: 9a042f7e3650ef38ad07af83b74f57c7a7d1820c (Added start up script)

...output omitted...
```

Notice that the clone of the Git repository is the first step of the build. Next, the Source-to-Image process built a new image called `s2i/php-helloworld:latest`. The last step in the build process is to push this image to the OpenShift private registry.

- 3.4. Review the Deployment for this application:

```
[student@workstation D0180-apps]$ oc describe deployment/php-helloworld
Name: php-helloworld
Namespace: ${RHT_OCP4_DEV_USER}-s2i
CreationTimestamp: Tue, 30 Mar 2021 12:54:59 -0400
Labels: app=php-helloworld
 app.kubernetes.io/component=php-helloworld
 app.kubernetes.io/instance=php-helloworld
Annotations: deployment.kubernetes.io/revision: 2
```

```
image.openshift.io/triggers:
[{"from":{"kind":"ImageStreamTag","name":"php-helloworld:latest"},"fieldPath":"spec.template.spec.containers[?(@.name==\"php-helloworld\")]...
Selector: openshift.io/generated-by: OpenShiftNewApp
Replicas: deployment=php-helloworld
1 desired | 1 updated | 1 total | 1 available | 0
unavailable
StrategyType: RollingUpdate
MinReadySeconds: 0
RollingUpdateStrategy: 25% max unavailable, 25% max surge
Pod Template:
Labels: deployment=php-helloworld
Annotations: openshift.io/generated-by: OpenShiftNewApp
Containers:
php-helloworld:
Ports: 8080/TCP, 8443/TCP
Host Ports: 0/TCP, 0/TCP
Environment: <none>
Mounts: <none>
Volumes: <none>
Conditions:
Type Status Reason
---- ----
Available True MinimumReplicasAvailable
Progressing True NewReplicaSetAvailable
OldReplicaSets: <none>
NewReplicaSet: php-helloworld-6f5d4c47ff (1/1 replicas created)
...output omitted...
```

## 3.5. Add a route to test the application:

```
[student@workstation D0180-apps]$ oc expose service php-helloworld \
> --name ${RHT_OCP4_DEV_USER}-helloworld
route.route.openshift.io/${RHT_OCP4_DEV_USER}-helloworld exposed
```

## 3.6. Find the URL associated with the new route:

```
[student@workstation D0180-apps]$ oc get route -o jsonpath='{..spec.host}{"\n"}'
${RHT_OCP4_DEV_USER}-helloworld-${RHT_OCP4_DEV_USER}-s2i.
${RHT_OCP4_WILDCARD_DOMAIN}
```

## 3.7. Test the application by sending an HTTP GET request to the URL you obtained in the previous step:

```
[student@workstation D0180-apps]$ curl -s \
> ${RHT_OCP4_DEV_USER}-helloworld-${RHT_OCP4_DEV_USER}-s2i.\
> ${RHT_OCP4_WILDCARD_DOMAIN}
Hello, World! php version is 7.3.20
```

- ▶ 4. Explore starting application builds by changing the application in its Git repository and executing the proper commands to start a new Source-to-Image build.

- 4.1. Enter the source code directory.

```
[student@workstation D0180-apps]$ cd ~/D0180-apps/php-helloworld
```

- 4.2. Edit the `index.php` file as shown below:

```
<?php
print "Hello, World! php version is " . PHP_VERSION . "\n";
print "A change is a coming!\n";
?>
```

Save the file.

- 4.3. Commit the changes and push the code back to the remote Git repository:

```
[student@workstation php-helloworld]$ git add .
[student@workstation php-helloworld]$ git commit -m 'Changed index page contents.'
[s2i b1324aa] changed index page contents
 1 file changed, 1 insertion(+)
[student@workstation php-helloworld]$ git push origin s2i
...output omitted...
Counting objects: 7, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 417 bytes | 0 bytes/s, done.
Total 4 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/${RHT_OCP4_GITHUB_USER}/D0180-apps
 f7cd896..b1324aa s2i -> s2i
```

- 4.4. Start a new Source-to-Image build process and wait for it to build and deploy:

```
[student@workstation php-helloworld]$ oc start-build php-helloworld
build.build.openshift.io/php-helloworld-2 started
[student@workstation php-helloworld]$ oc logs php-helloworld-2-build -f
...output omitted...

Successfully pushed .../php-helloworld:latest@sha256:74e757a4c0edaeda497dab7...
Push successful
```



#### Note

Logs may take some seconds to be available after the build starts. If the previous command fails, wait a bit and try again.

- 4.5. After the second build has completed use the `oc get pods` command to verify that the new version of the application is running.

```
[student@workstation php-helloworld]$ oc get pods
NAME READY STATUS RESTARTS AGE
php-helloworld-1-build 0/1 Completed 0 11m
php-helloworld-1-deploy 0/1 Completed 0 10m
php-helloworld-2-build 0/1 Completed 0 45s
php-helloworld-2-deploy 0/1 Completed 0 16s
php-helloworld-2-wq9wz 1/1 Running 0 13s
```

4.6. Test that the application serves the new content:

```
[student@workstation php-helloworld]$ curl -s \
> ${RHT_OCP4_DEV_USER}-helloworld-${RHT_OCP4_DEV_USER}-s2i.\
> ${RHT_OCP4_WILDCARD_DOMAIN}
Hello, World! php version is 7.3.20
A change is a coming!
```

## Finish

On workstation, run the `lab openshift-s2i finish` script to complete this lab.

```
[student@workstation php-helloworld]$ lab openshift-s2i finish
```

This concludes the guided exercise.

## ► Lab

# Deploying Containerized Applications on OpenShift

### Outcomes

You should be able to create an OpenShift application and access it through a web browser.

### Before You Begin

Make sure you have completed *Guided Exercise: Configuring the Classroom Environment* from *Chapter 1* before executing any command of this practice.

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

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

### Instructions

1. Prepare the lab environment.
2. Create a temperature converter application named `temps` written in PHP using the `php:7.3` image stream tag. The source code is in the Git repository at <https://github.com/RedHatTraining/D0180-apps/> in the `temps` directory. You may use the OpenShift command-line interface or the web console to create the application.
3. Verify that you can access the application in a web browser at `http://temps- ${RHT_OCP4_DEV_USER}-ocp.${RHT_OCP4_WILDCARD_DOMAIN}`.

### Evaluation

On `workstation`, run the `lab openshift-review grade` command to grade your work. Correct any reported failures and rerun the script until successful.

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

### Finish

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

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

This concludes the lab.

## ► Solution

# Deploying Containerized Applications on OpenShift

### Outcomes

You should be able to create an OpenShift application and access it through a web browser.

### Before You Begin

Make sure you have completed *Guided Exercise: Configuring the Classroom Environment* from *Chapter 1* before executing any command of this practice.

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

```
[student@workstation ~]$ lab openshift-review 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 OpenShift cluster.

```
[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 named "\${RHT\_OCP4\_DEV\_USER}-ocp" for the resources you create during this exercise:

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

2. Create a temperature converter application named `temps` written in PHP using the `php:7.3` image stream tag. The source code is in the Git repository at <https://github.com/RedHatTraining/D0180-apps/> in the `temps` directory. You may use the OpenShift command-line interface or the web console to create the application.

- 2.1. If using the command-line interface, run the following commands:

```
[student@workstation ~]$ oc new-app \
> php:7.3~https://github.com/RedHatTraining/D0180-apps \
> --context-dir temps --name temps
--> Found image 688c0bd (2 months old) in image stream "openshift/php" under tag
"7.3" for "php:7.3"

Apache 2.4 with PHP 7.3

PHP 7.3 available as container is a base platform ...output omitted...
...output omitted...

--> Creating resources ...
imagestream.image.openshift.io "temps" created
buildconfig.build.openshift.io "temps" created
deployment.apps "temps" created
service "temps" created
--> Success
Build scheduled, use 'oc logs -f bc/temps' to track its progress.
Application is not exposed. You can expose services to the outside world by
executing one or more of the commands below:
'oc expose svc/temps'
Run 'oc status' to view your app.
```

## 2.2. Monitor progress of the build.

```
[student@workstation ~]$ oc logs -f bc/temps
Cloning "https://github.com/RedHatTraining/D0180-apps" ...
Commit: f7cd8963ef353d9173c3a21dcccf402f3616840b (Initial commit, including all
apps previously in course)
...output omitted...
Successfully pushed image-registry.openshift-image-registry.svc:5000/
${RHT_OCP4_DEV_USER}-temps
Push successful
```

## 2.3. Verify that the application is deployed.

```
[student@workstation ~]$ oc get pods -w
NAME READY STATUS RESTARTS AGE
temps-1-build 0/1 Completed 0 91s
temps-57d678bbdd-dlz9c 1/1 Running 0 58s
```

Press Ctrl+C to exit the `oc get pods -w` command.

## 2.4. Expose the `temps` service to create an external route for the application.

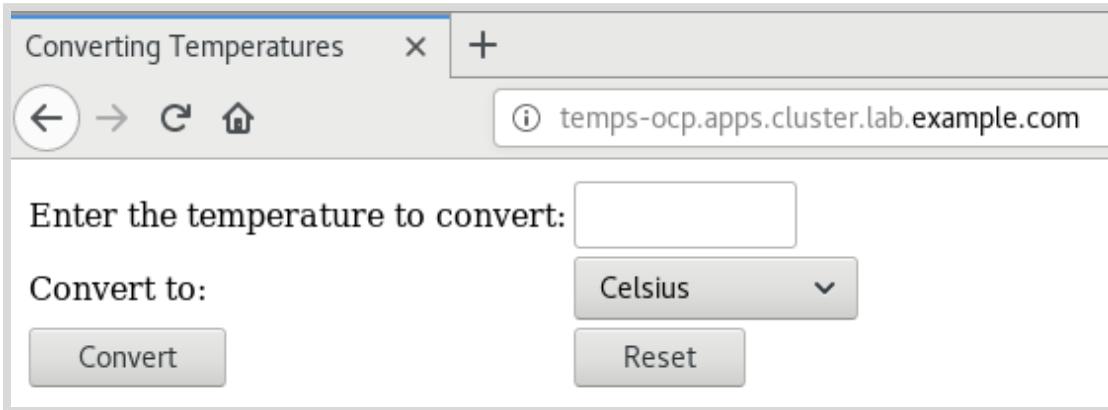
```
[student@workstation ~]$ oc expose svc/temps
route.route.openshift.io/temps exposed
```

- Verify that you can access the application in a web browser at `http://temps-${RHT_OCP4_DEV_USER}-ocp.${RHT_OCP4_WILDCARD_DOMAIN}`.

### 3.1. Determine the URL for the route.

```
[student@workstation ~]$ oc get route/temps
NAME HOST/PORT
temps temps-$${RHT_OCP4_DEV_USER} - ocp.$${RHT_OCP4_WILDCARD_DOMAIN} ...
...
```

- 3.2. Verify that the temperature converter application works by opening a web browser and navigating to the URL displayed in the previous step.



## Evaluation

On workstation, run the `lab openshift-review grade` command to grade your work. Correct any reported failures and rerun the script until successful.

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

## Finish

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

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

This concludes the lab.

# Summary

---

In this chapter, you learned:

- OpenShift Container Platform stores definitions of each OpenShift or Kubernetes resource instance as an object in the cluster's distributed database service, etcd. Common resource types are: Pod, Persistent Volume (PV), Persistent Volume Claim (PVC), Service (SVC), Route, Deployment, DeploymentConfig and Build Configuration (BC).
- Use the OpenShift command-line client oc to:
  - Create, change, and delete projects.
  - Create application resources inside a project.
  - Delete, inspect, edit, and export resources inside a project.
  - Check logs from application pods, deployments, and build operations.
- The oc new-app command can create application pods in many different ways: from an existing container image hosted on an image registry, from Dockerfiles, and from source code using the Source-to-Image (S2I) process.
- Source-to-Image (S2I) is a tool that makes it easy to build a container image from application source code. This tool retrieves source code from a Git repository, injects the source code into a selected container image based on a specific language or technology, and produces a new container image that runs the assembled application.
- A Route connects a public-facing IP address and DNS host name to an internal-facing service IP. While services allow for network access between pods inside an OpenShift instance, routes allow for network access to pods from users and applications outside the OpenShift instance.
- You can create, build, deploy and monitor applications using the OpenShift web console.

## Chapter 7

# Deploying 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

- Managing Applications with the Web Console
- Managing Applications with the Web Console
- Managing Applications with the CLI
- Managing Applications with the CLI

# Managing Applications with the Web Console

---

## Objectives

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

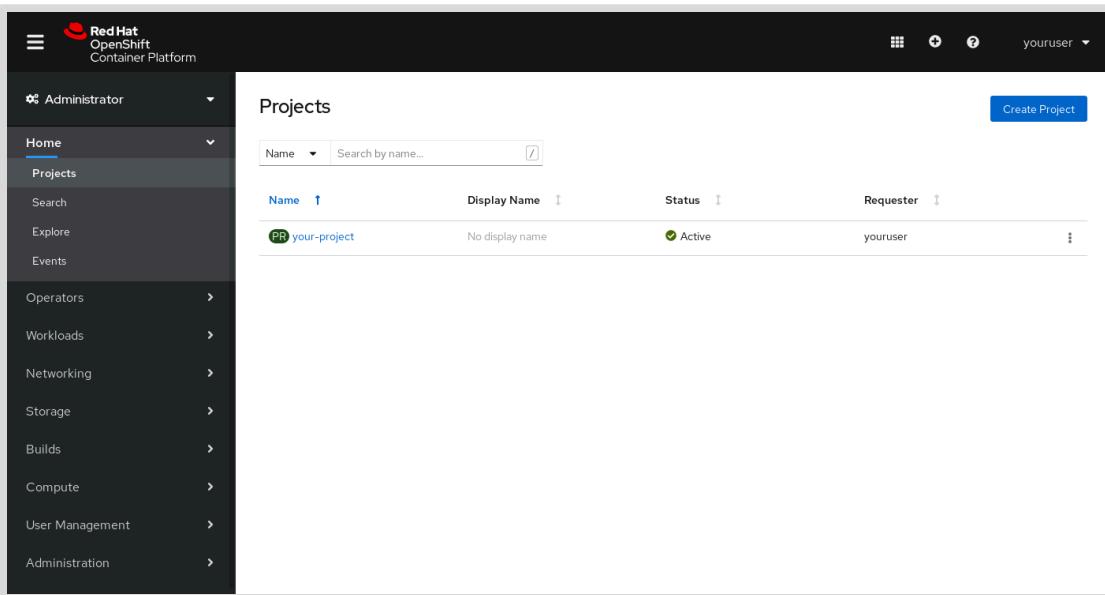
- Deploy an application from a binary image and manage its resources.
- View pod and build logs.
- Edit resource definitions.

## 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 containing a "Administrator" dropdown, a "Home" dropdown set to "Projects", and links for "Search", "Explore", and "Events". To the right of the sidebar is the main content area titled "Projects". It features a search bar with "Name" and "Search by name..." placeholder text. A "Create Project" button is located in the top right corner of the content area. The main table displays a single row for a project named "your-project". The columns in the table are "Name" (with a link to "your-project"), "Display Name" (showing "No display name"), "Status" (showing "Active" with a checkmark), and "Requester" (showing "youruser").

**Figure 7.1: 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.

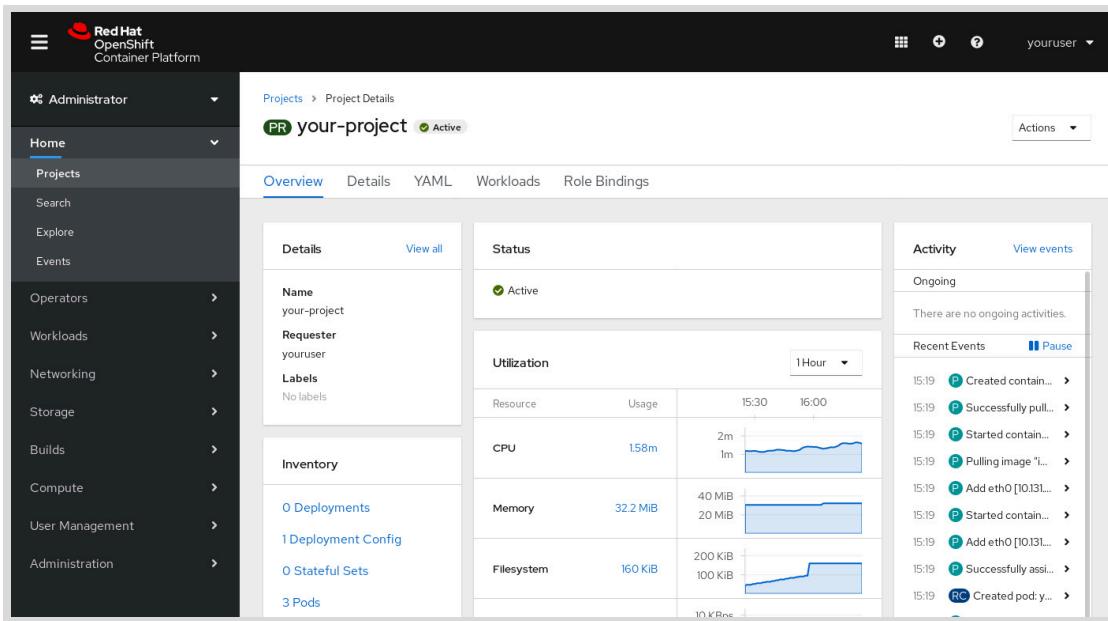


Figure 7.2: Web console project overview

## 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.

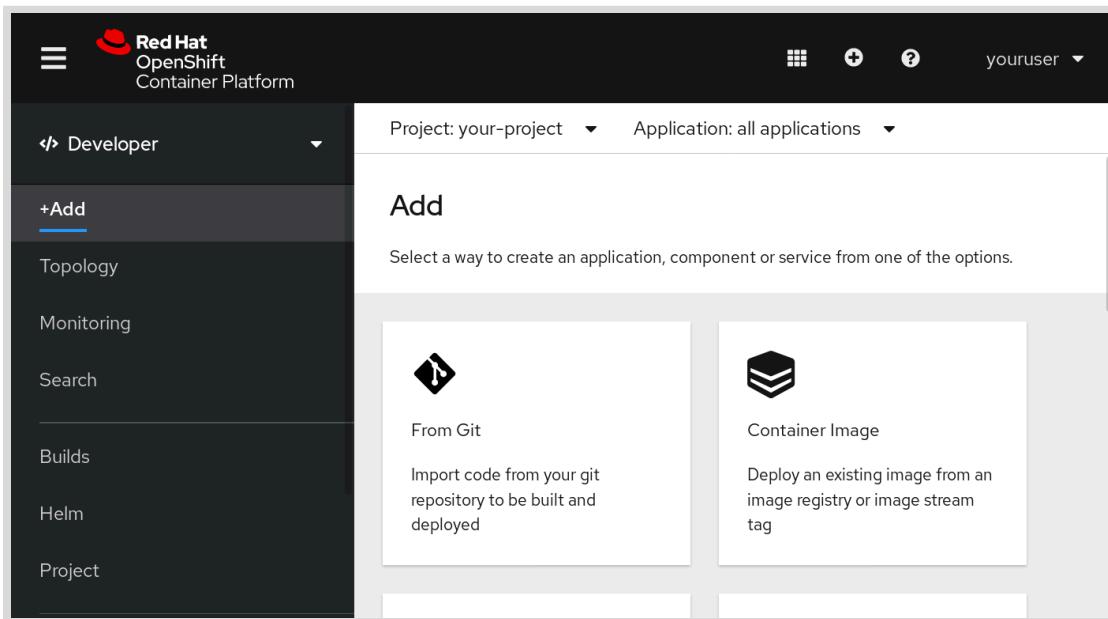


Figure 7.3: Actions available from the +Add section

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.

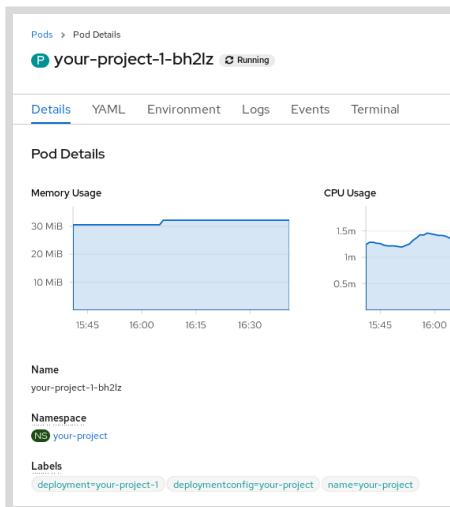
## Resource Detail Pages

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 7.4: 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, it says 'Log stream ended.' and shows a log stream with 56 lines. The log content includes build steps like 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, there are 'Download' and 'Expand' buttons.

```
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 sha256:02cb2f884027e51a3c232637b7496e15a9a5671885a64f0d5e4a9dc2c1674fd
Copying blob sha256:9e7a6dc796f0a75c560158a9f9e30fb8b5a90cb53edce9ffbd57784064de39
Copying blob sha256:fcc5b206e9329a1674dd9e8cfbee45c9be28dd0dcbabb3c6bb67a2f22cfcf2a
Copying blob sha256:e70210e0589e97471d999c4265b7c8e64da328e48f116b5f260353b2e0a2ad373
Copying blob sha256:a0a629e11c996fb5c92c3ede13fa50843042f978963a2a9e31288d1a7d5489
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 7.5: 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](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
```

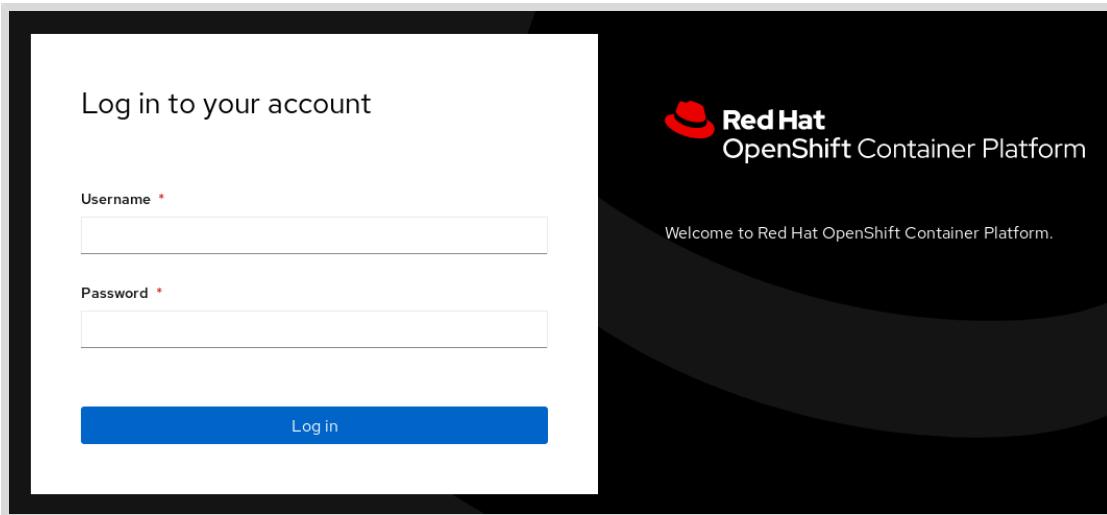
Another way to find the host name of your OpenShift web console is inspecting the routes on the `openshift-console` project. You must be logged into OpenShift before.

```
[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 ...
```

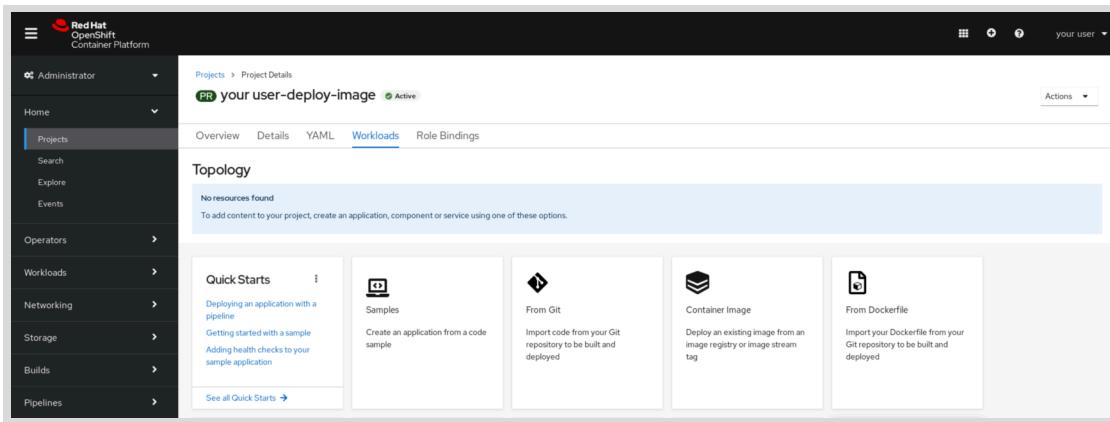
**Note**

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.

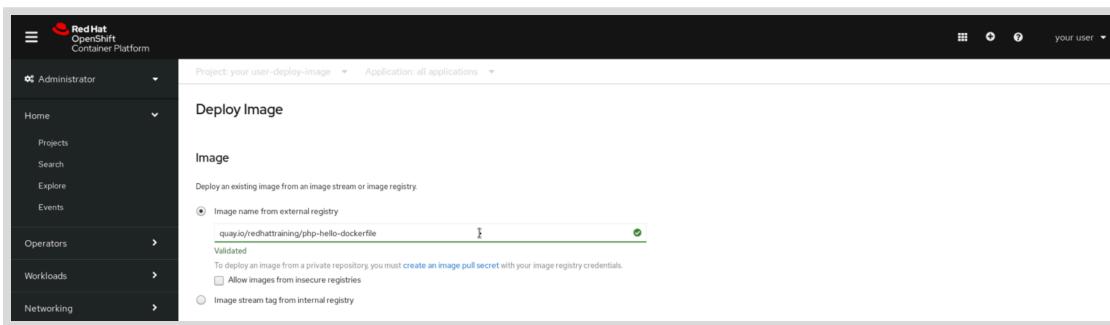
- 1.2. Open a web browser and navigate to `https://console-openshift-console.apps.cluster.domain.example.com` to access the OpenShift web console. Replace `apps.cluster.domain.example.com` with the value you got from the previous step. You should see the login page for the web console.
- 1.3. Log in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



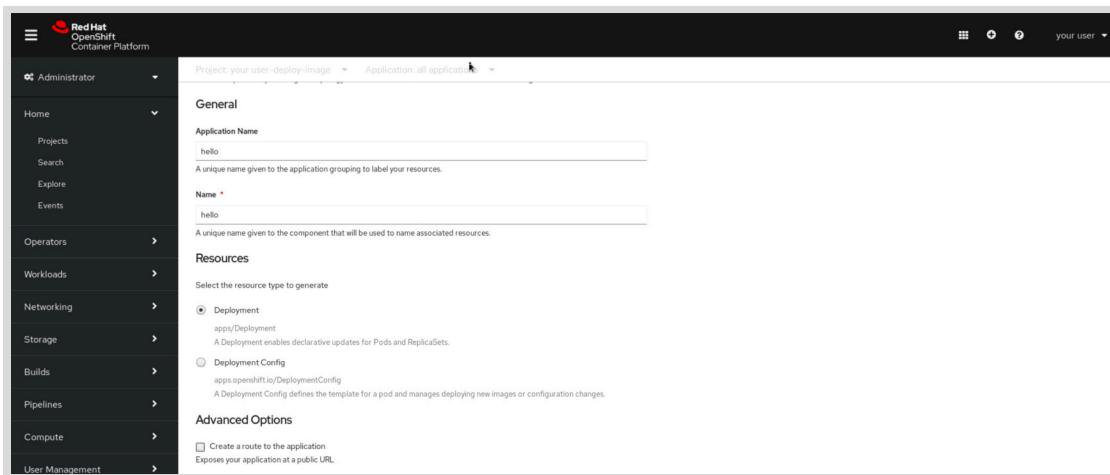
- 1.4. Navigate to the Administrator perspective, then select **Home** → **Projects** from the left side menu. First time users are placed in that page initially, Click **Create Project**. In the **Create Project** dialog box, enter `youruser-deploy-image` in the **Name** field. Replace `youruser` with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

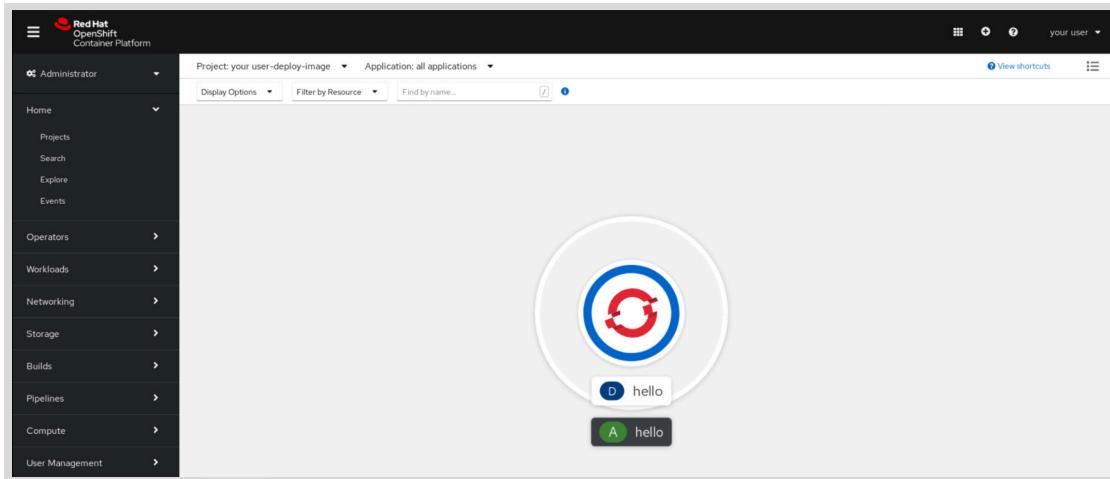


- 2.3. Scroll down to see information about the image and replace `php-hello-dockerfile` with `hello` in the **Name** and **Application Name** fields.

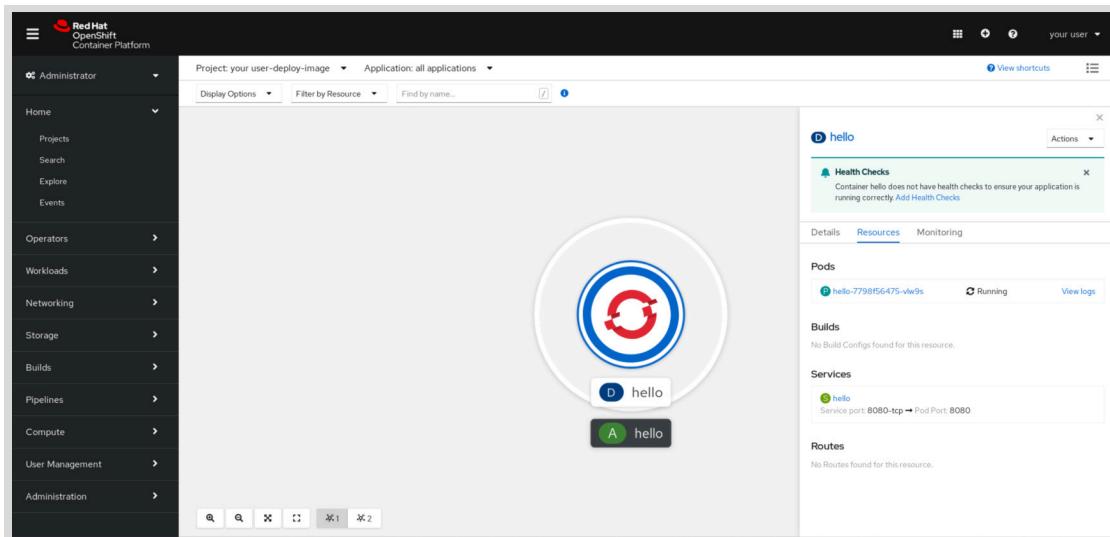


Uncheck the **Create a route to the application** option in the **Advanced Options** section at the bottom of the page.

- 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.

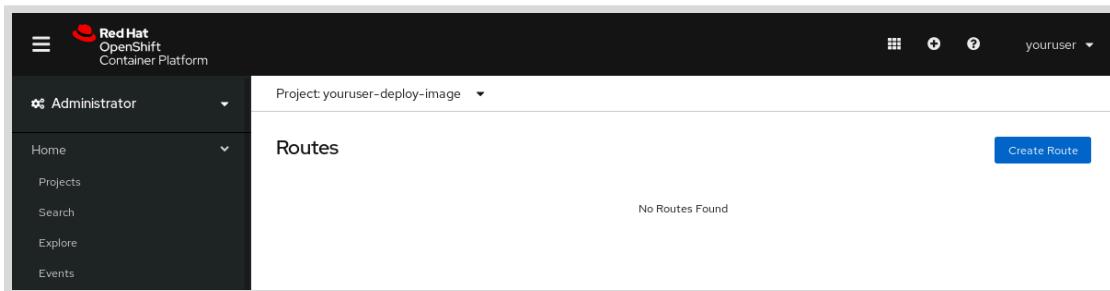


- 2.5. Click the **i** icon above the deployment name to expand the deployment details and to view the number of running pods. Wait until the overview page displays a successful deployment with one pod:



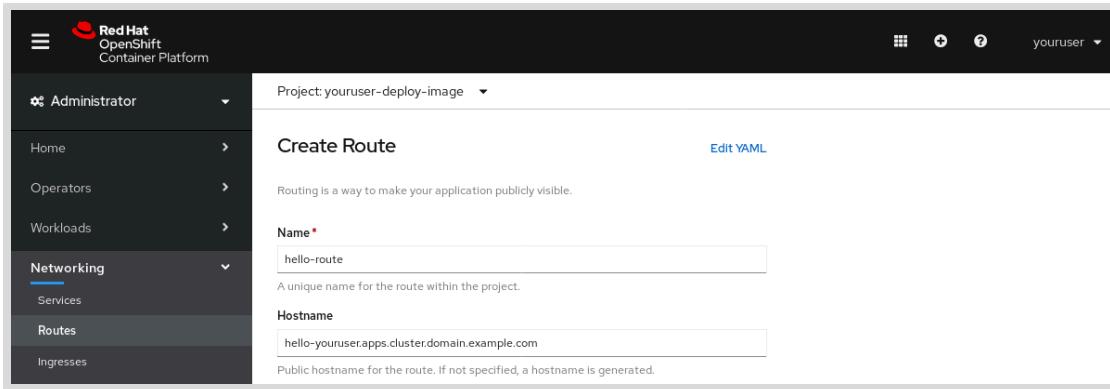
- 2.6. In the navigation bar, click **Networking → Routes**.

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

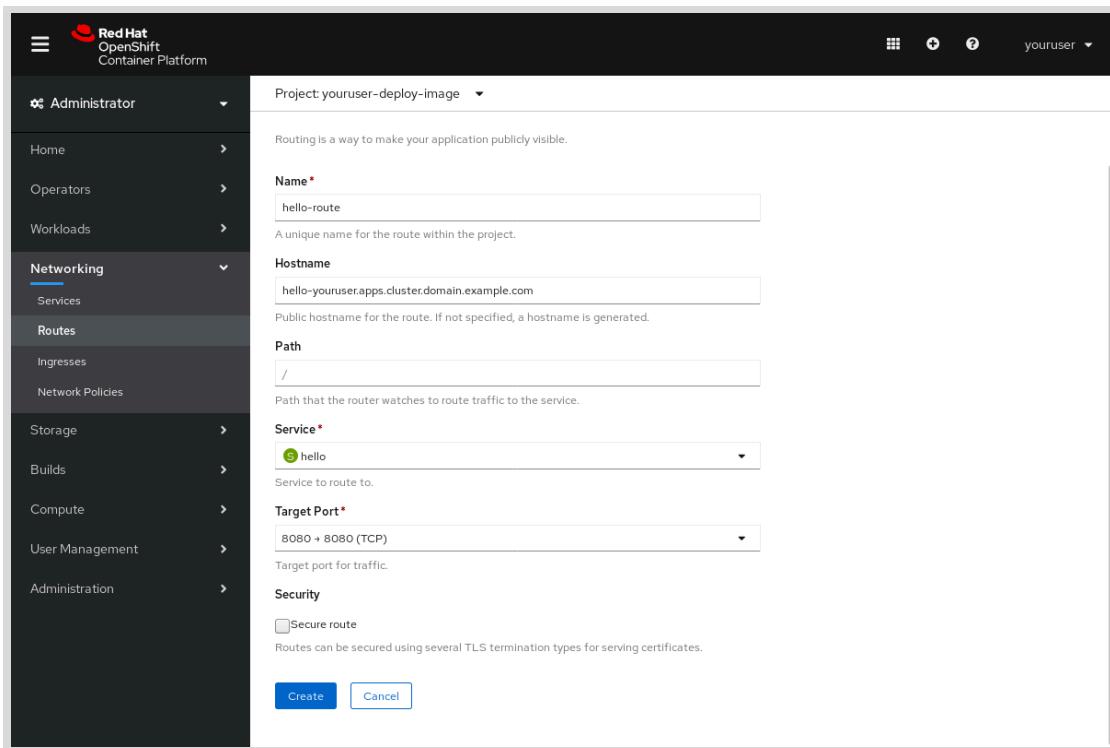


Enter **hello-route** in the **Name** field. Enter **hello-youruser.apps.cluster.domain.example.com** in the **Hostname** field.

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 (with Networking, Services, Routes, Ingresses, Network Policies), Storage, and Builds. The 'Routes' section under 'Networking' is currently selected. The main content area is titled 'Project: youruser-deploy-image' and shows 'Route Details' for 'hello-route'. The route is listed as 'RT hello-route Accepted'. The 'Details' tab is active. Route details include: Name: hello-route, Namespace: youruser-deploy-image, Status: Accepted, Host: hello-youruser.apps.cluster.domain.example.com, Path: -, Location: http://hello-youruser.apps.cluster.domain.example.com. Labels listed are app=hello, app.kubernetes.io/component=hello, app.kubernetes.io/instance=hello, app.kubernetes.io/part-of=hello, and app.openshift.io/runtime-version=latest.

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 (with Pods), and Builds. The 'Pods' section under 'Workloads' is currently selected. The main content area is titled 'Project: your user-deploy-image' and shows the 'Pods' page. A single pod is listed: Name: hello-7798f56475-vlw9s, Status: Running, Ready: 1/1, Restarts: 0, Owner: hello-7798f56475, Memory: 61.3 MB, CPU: 0.001 cores, Created: Jun 8, 2:07 pm. A 'Create Pod' button is visible in the top right.

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 (with Pods), and Builds. The 'Pods' section under 'Workloads' is currently selected. The main content area is titled 'Project: your user-deploy-image' and shows the 'Pod Details' page for 'hello-7798f56475-vlw9s'. The pod status is 'Running'. The 'Logs' tab is selected. The logs output is as follows:

```
[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
AH00558: 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.

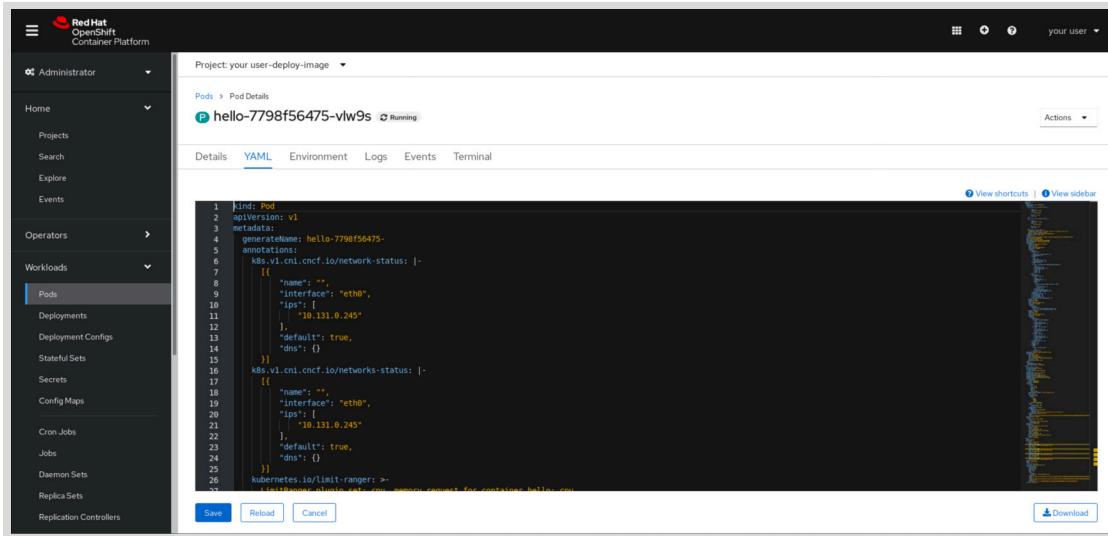
Still on the Pod Details page, click the Terminal tab to open a remote shell to the single container in the application pod. If the terminal window is too small, click Expand to hide the web console navigation panels. The terminal can only execute commands that exist inside the application's container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

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

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

### 3.3. View a resource definition.

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



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

► 4. Delete resources from the project.

- 4.1. On the **Pod Details** page, click **Actions** → **Delete Pod**. In the confirmation dialog box, click **Delete** to delete the running pod. The web console displays the **Pods** page.  
Wait until the **Pods** page shows that a new pod that was created by the deployment configuration to replace the deleted pod.
  - 4.2. On the navigation bar, click **Workloads** → **Deployment** 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](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
 ...output omitted...
 "objects": [
 {
 "apiVersion": "v1",
 "kind": "Secret", ②
 ...output omitted...
 },
 "stringData": {
 "database-password": "${DATABASE_PASSWORD}",
 "database-user": "${DATABASE_USER}"
 ...output omitted...
 {
 "apiVersion": "route.openshift.io/v1",
 "kind": "Route", ③
 ...output omitted...
 "spec": {
 "host": "${APPLICATION_DOMAIN}",
 "to": {
 "kind": "Service",
 "name": "${NAME}"
 ...output omitted...
```

- ① The template is a copy of the standard cakephp-mysql-example template, with resources and parameters specific to that framework deleted. The

custom template is suitable for any simple PHP application that uses a MySQL database.

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

```
...output omitted...
{
 "apiVersion": "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": [
...output omitted...
```

```

{
 "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 quotesdb-6b7ffcc649-ds1pq \
| 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\n
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.

# 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 8

# 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](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:

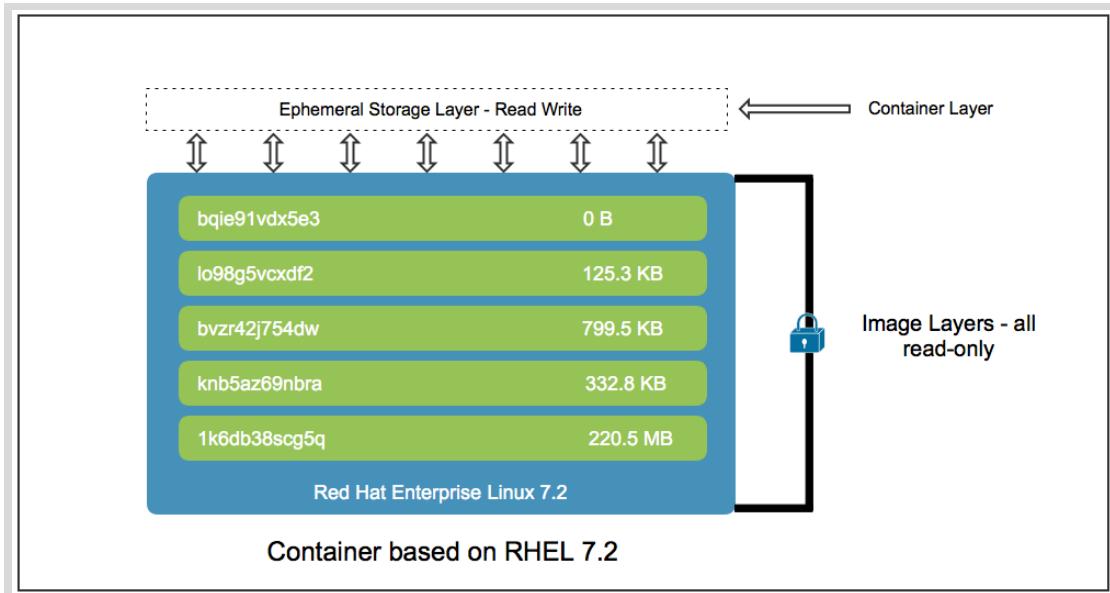
- **Minimal size:** The Universal Base Image is a relatively minimal (approximately 90-200 MB) base container image with fast startup times.
- **Security:** Provenance is a huge concern when using container base images. You must use a trusted image, from a trusted source. Language runtimes, web servers, 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 8.1: 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.<br><br>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](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](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 ③

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 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...
```

- 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         | Actions |
|-------------------------|--------------|-------------------------------------|------|-----------------|---------|
| 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 8.2: 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 8.3: 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](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 D0288-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.

```
[student@workstation D0288-apps]$ oc get pods
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 9

# 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

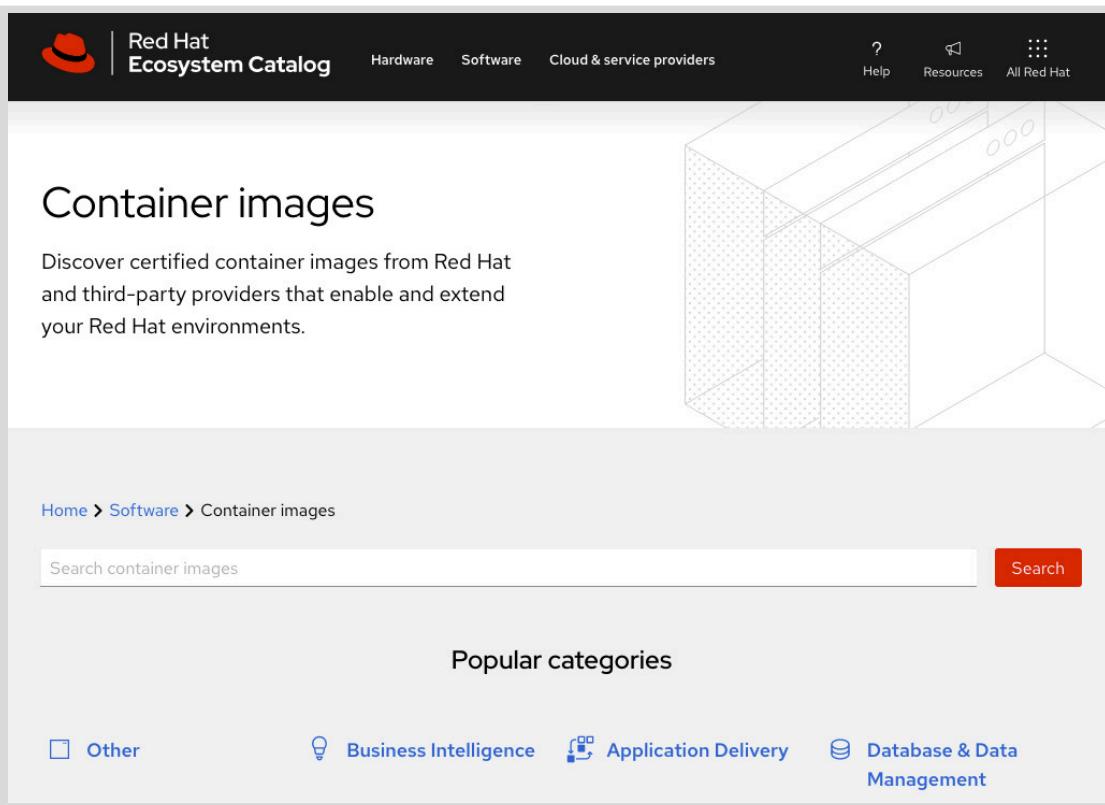
Red Hat manages a set of public and private container registries to serve different kinds of container images to distinct audiences. Container images that are supported with production-

## Chapter 9 | 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 9.1: 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](https://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.

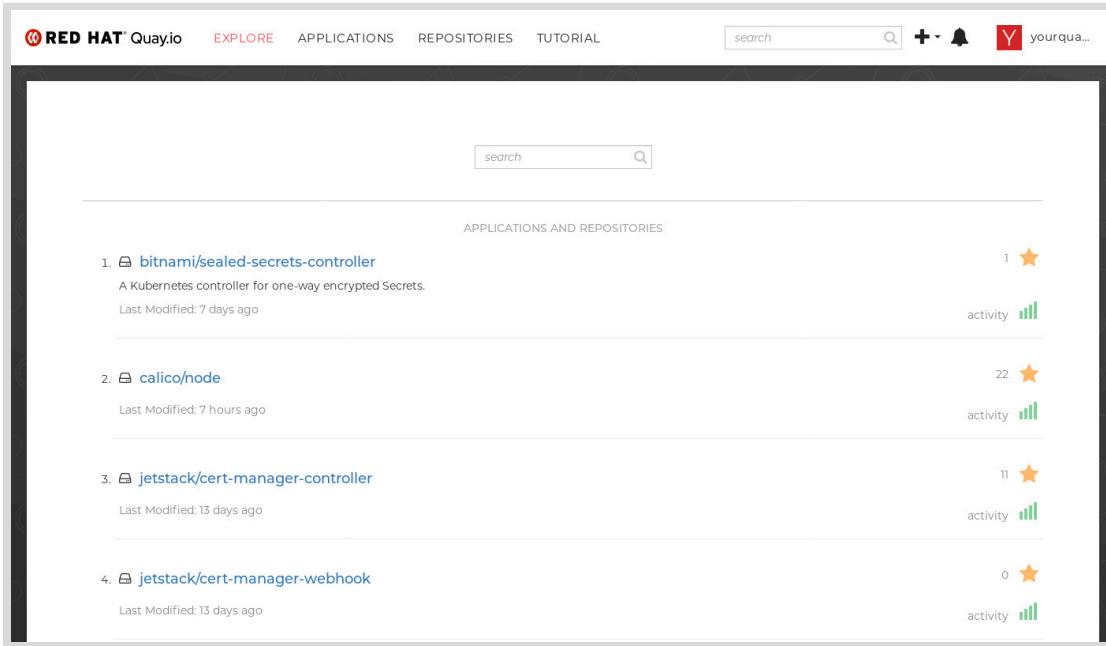


Figure 9.2: Quay.io web page

## 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](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 ~/D0288/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:

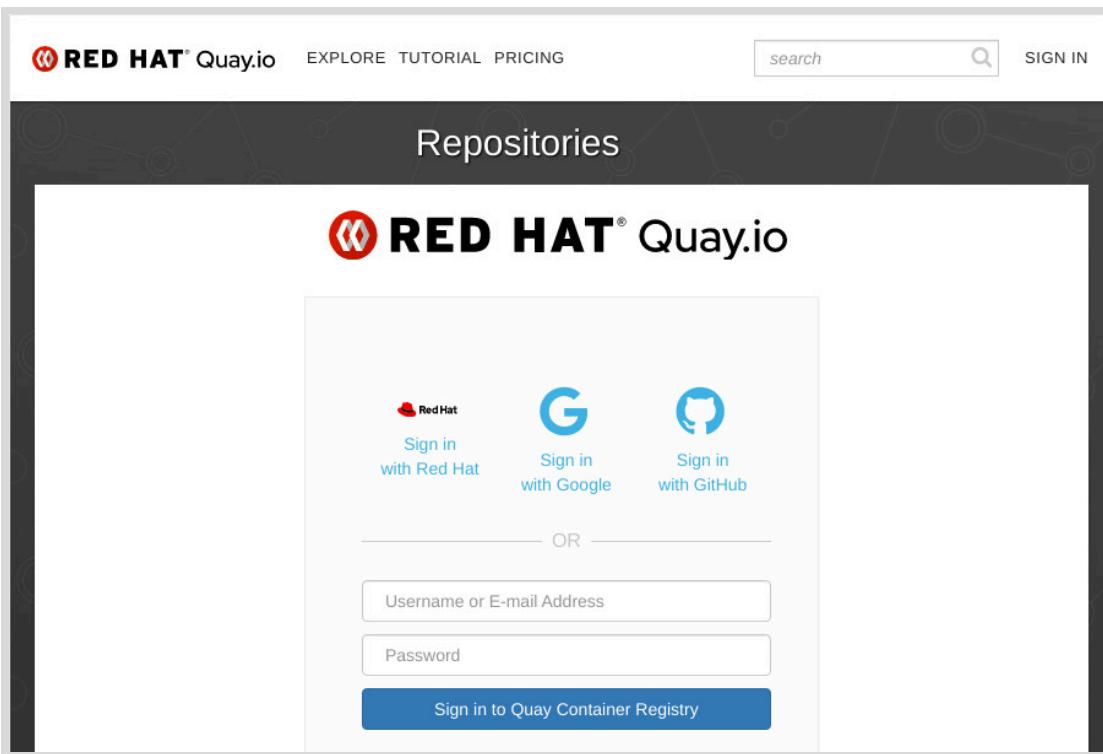
```
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-external-registry
project.project.openshift.io "external-registry" deleted
```

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

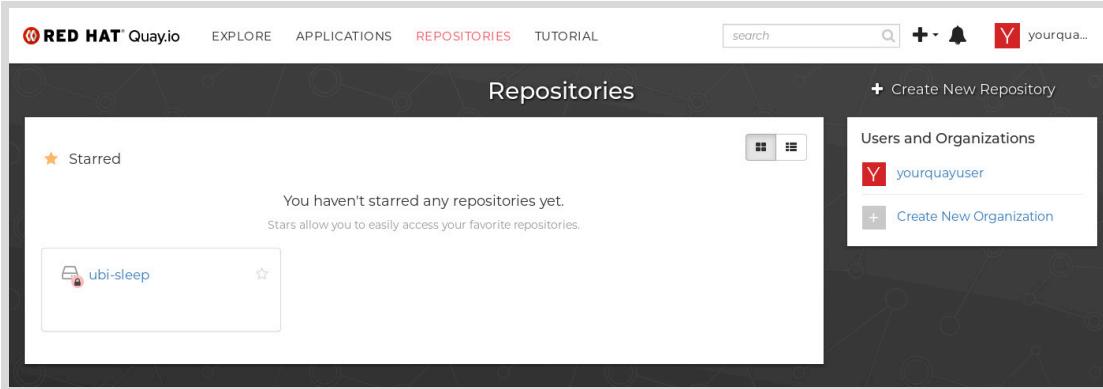
```
[student@workstation ~]$ skopeo delete \
docker://quay.io/${RHT_OCP4_QUAY_USER}/ubi-sleep:1.0
```

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

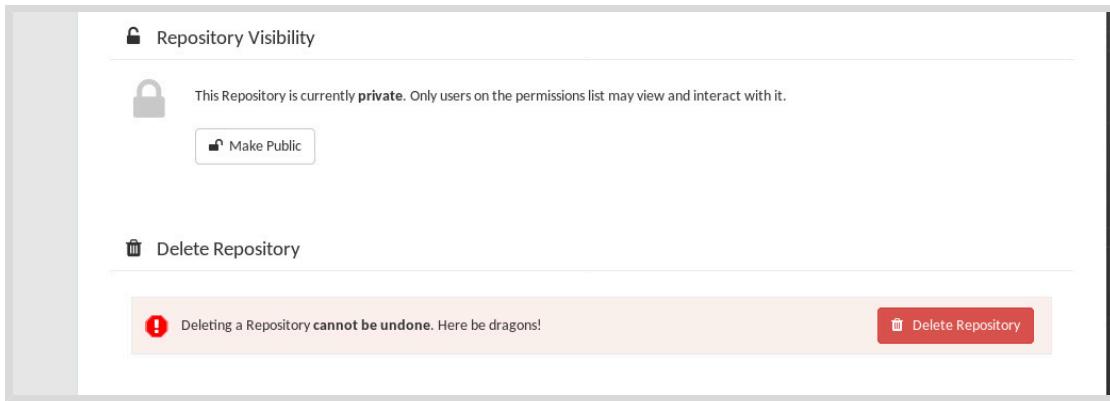
Navigate to <http://quay.io> and click **Sign In** to provide your user credentials. Click **Sign in to Quay Container Registry** to log in to Quay.io.



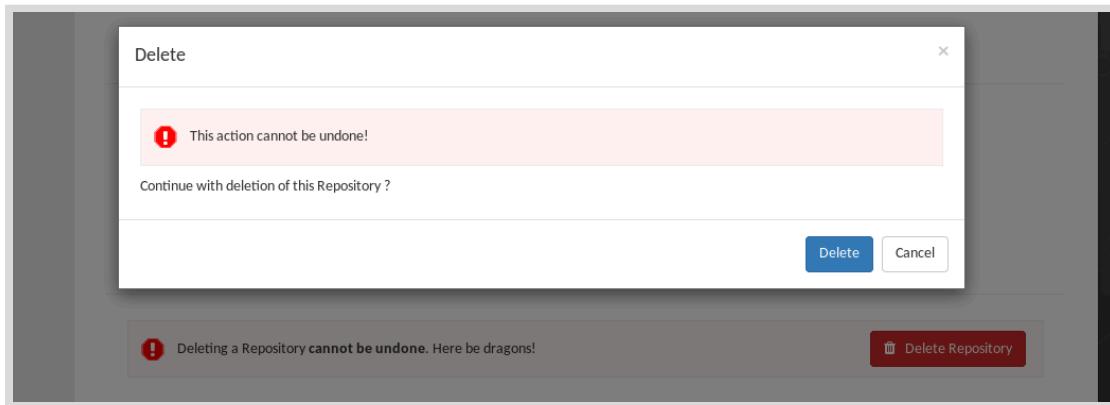
- 4.4. On the Quay.io main menu, click **Repositories** and look for **ubi-sleep**. The lock icon indicates that it is a private repository that requires authentication for both pulls and pushes. Click **ubi-sleep** to display the **Repository Activity** page.



- 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](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](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.

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](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 9 |** 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:

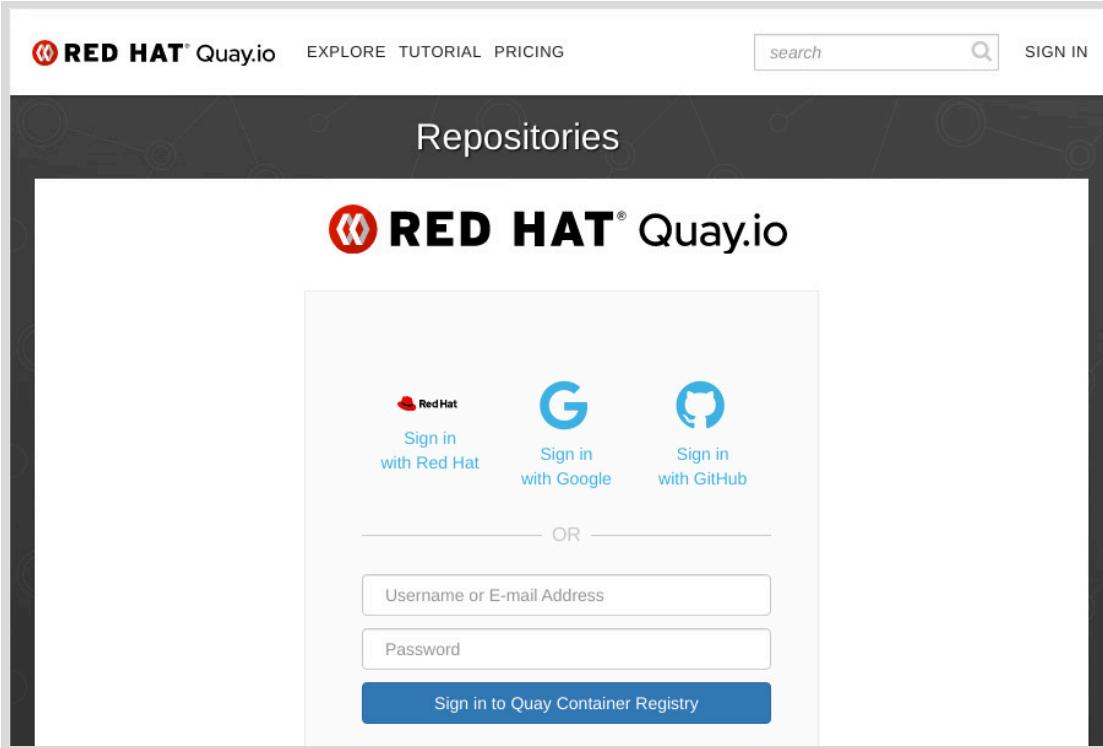
```
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-common
project.project.openshift.io "youruser-common" deleted
```

6.3. Delete the container image from the external registry:

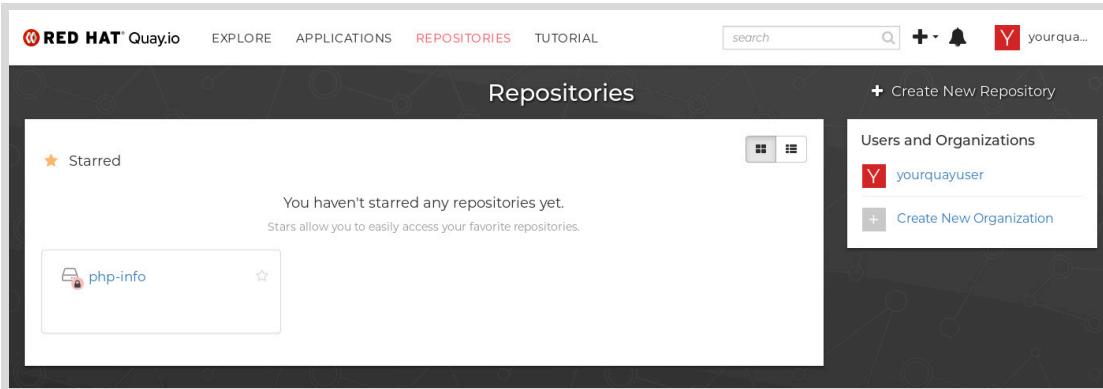
```
[student@workstation ~]$ skopeo delete \
docker://quay.io/${RHT_OCP4_QUAY_USER}/php-info:latest
```

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

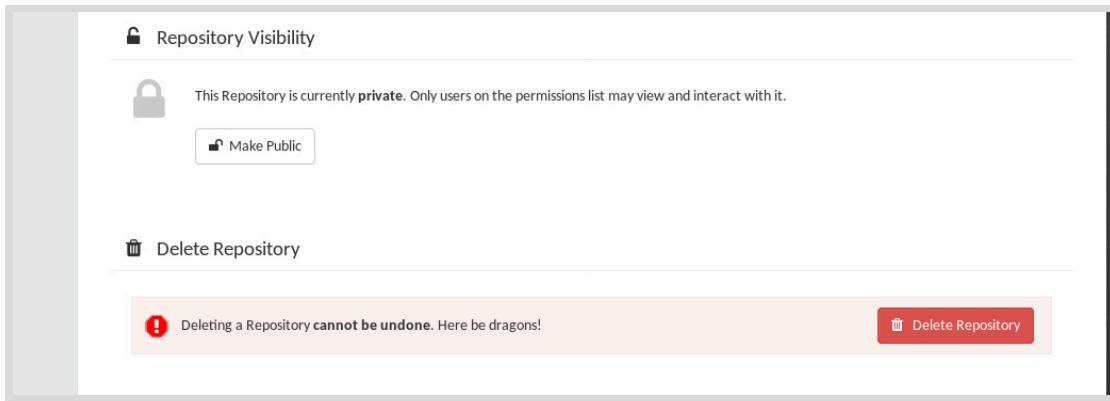
Navigate to <http://quay.io> and click **Sign In** to provide your user credentials. Click **Sign in to Quay Container Registry** to log in on Quay.io.



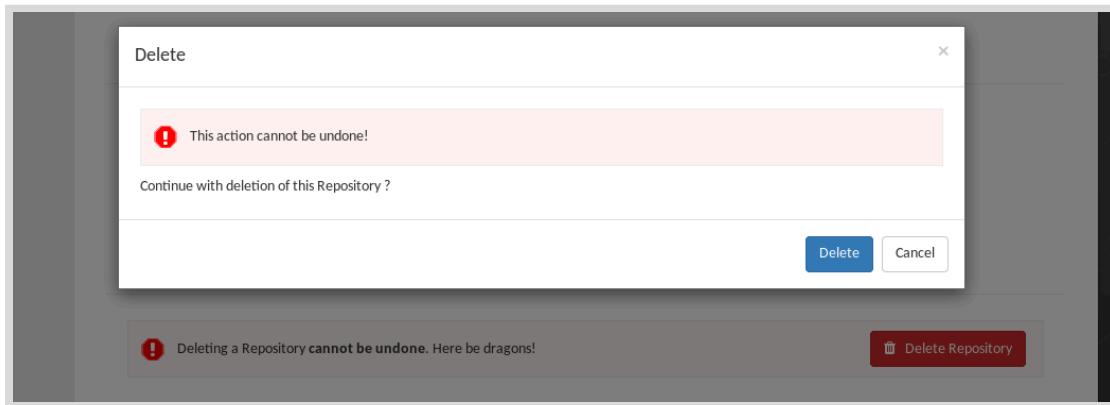
- 6.5. On the Quay.io main menu, click **Repositories** and look for **php-info**. Click **php-info** to open the **Repository Activity** page.



- 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 10

# 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](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"
 }
 },
 ...
 },
}
```

- Multiple builds can run simultaneously.
- After successfully built, the `php-example:latest` container image is available for deployment.
- The build uses a Git input source to create the final container image.
- The BuildConfig uses the Docker strategy.
- 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"
 }
 },
 ...
 },
}
```

- Multiple builds can run simultaneously.
- After successfully built, the `php-example:latest` container image is available for deployment.
- The build uses a Git input source to create the final container image.
- The BuildConfig uses the Docker strategy.
- 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](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]$ 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](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/D0288/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.

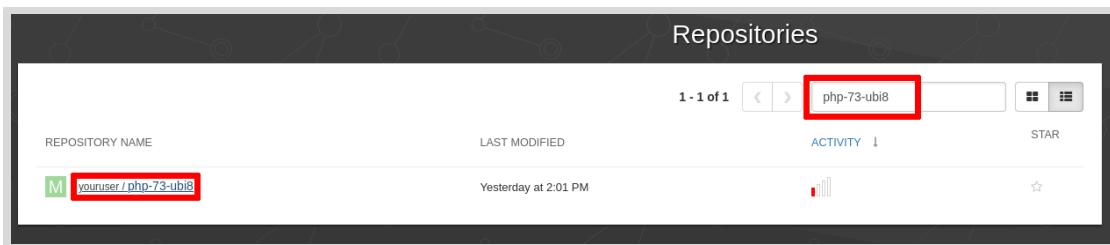
6.1. Change to the home directory.

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

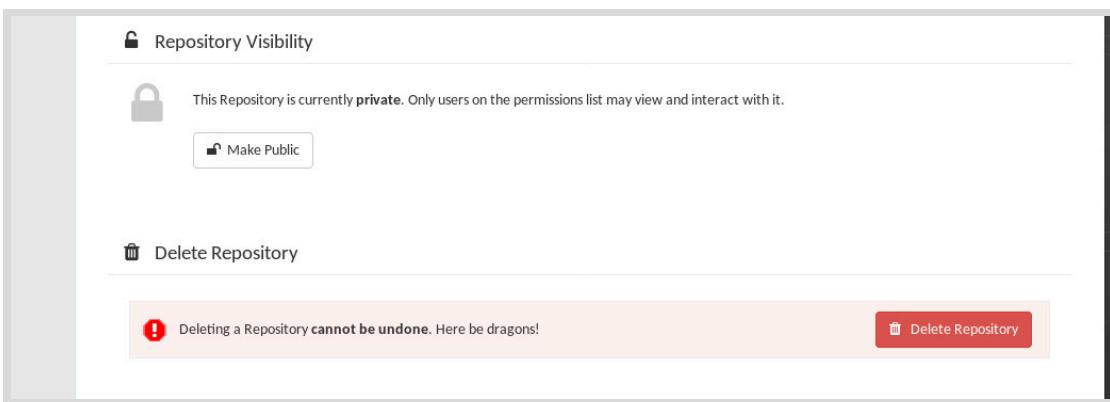
6.2. Delete the container image from the external registry:

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

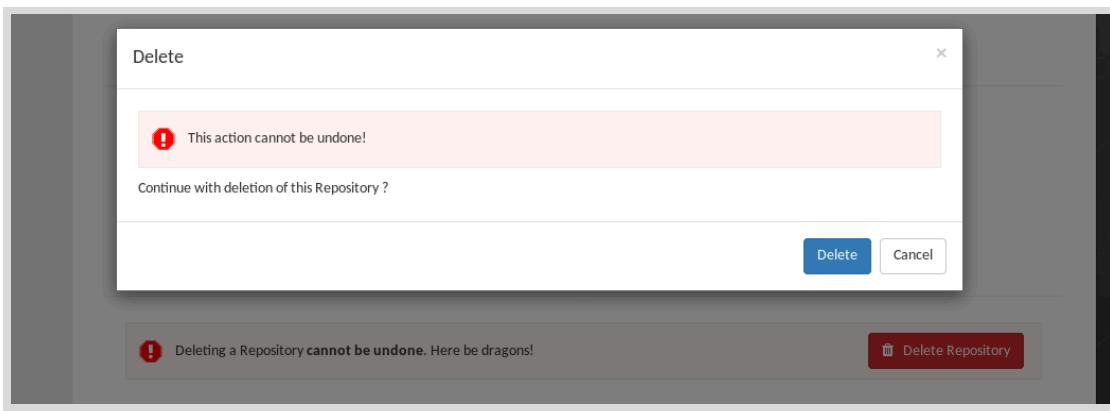
- 6.3. Log in on Quay.io using your personal free account.
- 6.4. On Quay.io's top-level menu, click **Repositories** and filter repositories with the **php-73-ubi8** name. Then, click the **php-73-ubi8** repository.



- 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](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:

```
[student@workstation ~]$ oc get pods
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 | <a href="http://github.com/youruser/DO288-apps">http://github.com/youruser/DO288-apps</a> |

- 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 11

# 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.

### Sections

- Customizing an Existing S2I Base Image
- Customizing an Existing S2I Base Image
- Creating an S2I Base Image
- Creating an S2I Base Image

### Lab

- Lab: Customizing Source-to-Image Builds

# 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](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 RHOCP 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:
'httpd -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 RHOCP 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 RHOCP 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 RHOCP.

## 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-nginx \
docker://quay.io/${RHT_OCP4_QUAY_USER}/s2i-do288-nginx
```

- 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 9, 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](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="httdp" \
 Name="s2i-do288-httdp" \
 Version="1.0" \
 Release="1"

Labels consumed by RHOC
LABEL io.k8s.description="A basic Apache HTTP Server S2I builder image" \
 io.k8s.display-name="Apache HTTP Server S2I builder image for D0288" \
 io.openshift.expose-services="8080:http" \
 io.openshift.s2i.scripts-url="image:///usr/libexec/s2i" ②

This label is used to categorize this image as a builder image in the
RHOC web console.
LABEL io.openshift.tags="builder, httdp, httdp24"

Apache HTTP Server DocRoot
ENV DOCROOT /var/www/html

RUN yum install -y --nodocs --disableplugin=subscription-manager httdp && \ ③
```

```
yum clean all --disableplugin=subscription-manager -y && \
echo "This is the default index page from the s2i-do288-httd 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-httd` directory and overwrite the generated template Containerfile:

```
[student@workstation ~]$ rm ~/s2i-do288-httd/Dockerfile
[student@workstation ~]$ cp ~/D0288/labs/apache-s2i/Containerfile
~/s2i-do288-httd/
```

- 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-httd/s2i/bin` directory and overwrite the generated scripts:

```
[student@workstation ~]$ cp -Rv ~/D0288/labs/apache-s2i/s2i ~/s2i-do288-httd/
'D0288/labs/apache-s2i/s2i/bin/assemble' -> 's2i-do288-httd/s2i/bin/assemble'
'D0288/labs/apache-s2i/s2i/bin/usage' -> 's2i-do288-httd/s2i/bin/usage'
'D0288/labs/apache-s2i/s2i/bin/run' -> 's2i-do288-httd/s2i/bin/run'
```

- 3.3. This exercise does not implement the `save-artifacts` script. Remove it from the `~/s2i-do288-httd/s2i/bin` directory:

```
[student@workstation ~]$ rm -f ~/s2i-do288-httd/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...
...output omitted...
```

- 4.10. Use the `curl` command to test the application:

```
[student@workstation s2i-sample-app]$ curl http://localhost:8080
This is the index page from the app
```

- 4.11. Stop the test application container:

```
[student@workstation s2i-sample-app]$ 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
```

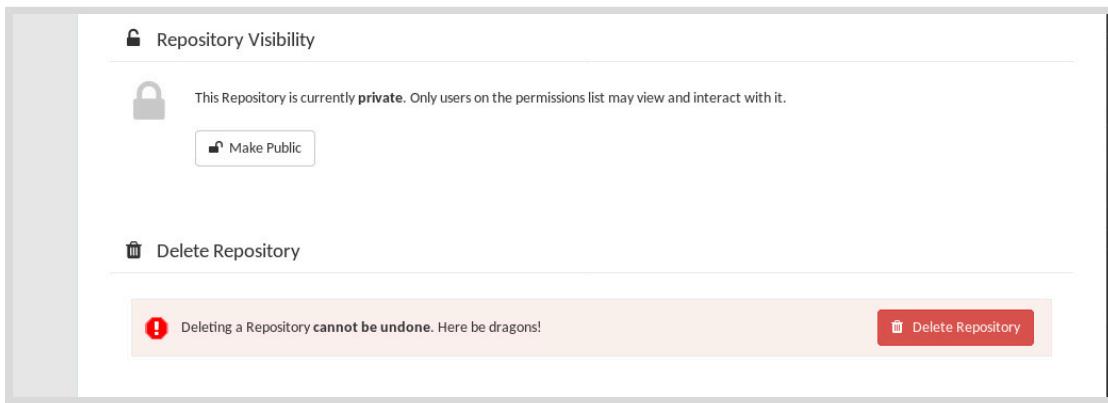
- 8.4. Delete the `s2i-do288-httdp` image from the external registry:

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

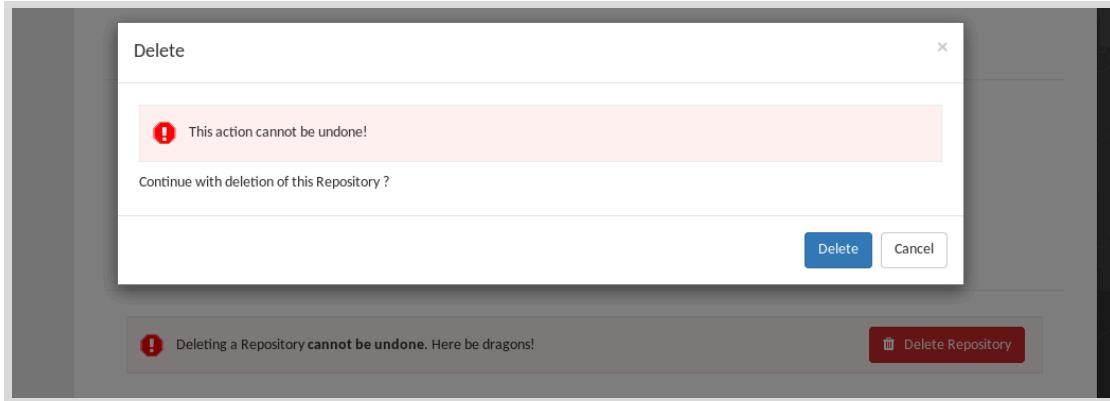
- 8.5. Log in to Quay.io using your personal free account.

    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:

```
[student@workstation ~]$ oc get pods
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 12

# 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.

### Sections

- Creating a Helm Chart
- Creating a Helm Chart
- Customizing Deployments with Kustomize
- Customizing Deployments with Kustomize

### Lab

- Lab: Deploying Applications with Helm Charts and Kustomize

# 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://exemplerepo.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 `~/D0288/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 `~/D0288/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 `~/D0288/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:

```
[student@workstation famouschart]$ oc get deployments
NAME READY UP-TO-DATE AVAILABLE AGE
famousapp-famouschart 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 ~/D0288/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 ~/D0288/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 `~/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
```

- 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 13

# 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 |
|---------------------|-----------|---------------------------------------------------------------------------------------------------------------------------------|---------------|
| initialDelaySeconds | Yes       | Determines how long to wait after the container starts before beginning the probe.                                              | 0             |
| timeoutSeconds      | 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             |
| periodSeconds       | No        | Specifies the frequency of the checks.                                                                                          | 1             |
| successThreshold    | No        | Specifies the minimum consecutive successes for the probe to be considered successful after it has failed.                      | 1             |
| failureThreshold    | No        | Specifies the minimum consecutive failures for the probe to be considered failed after it has succeeded.                        | 3             |

## Methods of Checking Application Health

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 13.1: Adding probes by using the web console**

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

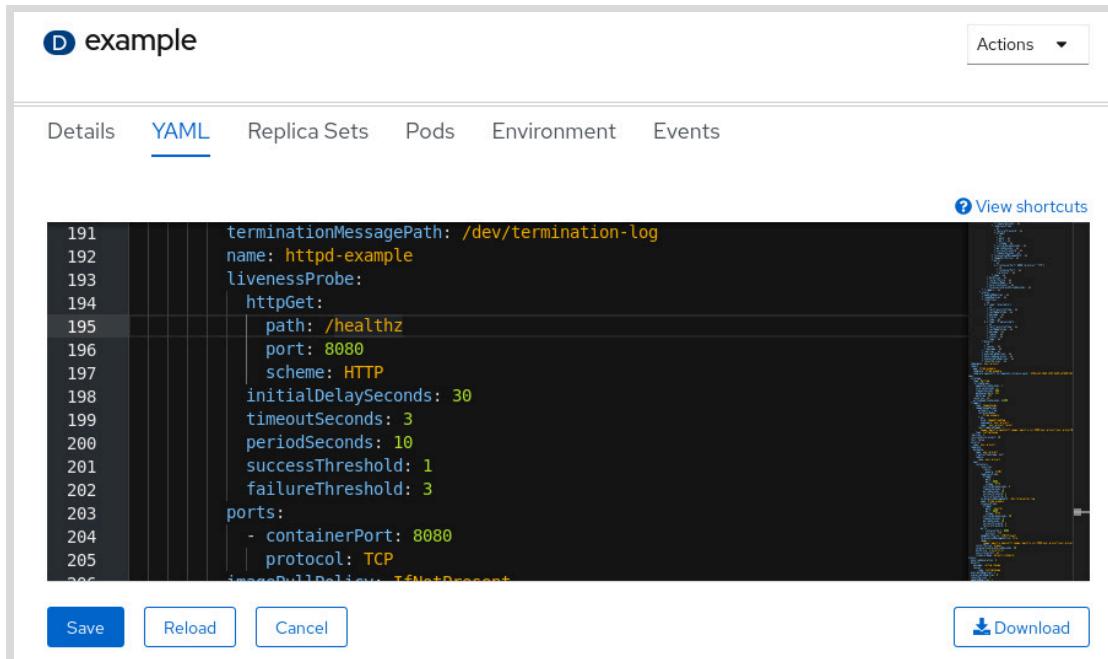


Figure 13.2: Adding probes by using the web console YAML editor

## 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](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 D0288-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-fsfx8 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-fsfx8 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 13.3: 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](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 ~]$ ~/D0288/labs/strategy/post-hook.sh
deploymentconfig.apps.openshift.io/mysql patched
```

- ▶ 4. Verify the patched deployment configuration and roll out the new deployment configuration.
- 4.1. Verify that the deployment strategy is now Recreate, and that there is a post life-cycle hook that executes the `import.sh` script:

```
[student@workstation ~]$ oc describe dc/mysql | grep -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**:

```
[student@workstation ~]$ watch -n 2 oc get pods
NAME READY STATUS RESTARTS ...
mysql-1-deploy 0/1 Completed 0 5m26s
mysql-2-deploy 0/1 Error 0 3m30s
mysql-2-hook-post 0/1 Error 0 3m8s
mysql-3-29jwt 1/1 Running 0 57s
mysql-3-deploy 0/1 Completed 0 79s
mysql-3-hook-post 0/1 Completed 0 48s
```

- 5.5. Open a new terminal to display the logs from the running post life-cycle hook pod. They show that the script can connect to the database a few times, and then returns a success status:

```
[student@workstation ~]$ oc logs -f mysql-3-hook-post
Downloading SQL script that initializes the database...
Trying 5 times, sleeping 2 sec between tries:
Checking if MySQL is up...Database is up
mysql: [Warning] Using a password on the command line interface can be insecure.
Database initialized successfully
```

The number of tries depends on your hardware and the nodes each pod is scheduled to run by 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   |
| <b>mysql-3-29jwt</b> | <b>1/1</b> | <b>Running</b> | 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. The `latest` option indicates that the newest version of the template must be used:

```
[user@host ~]$ oc rollout latest dc/name
```

This option is often used to start a new deployment or upgrade an application to the latest version.

- To view the history of deployments for a specific deployment configuration, use the `oc rollout history 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](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 imageregistry.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 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.
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 DO288-apps]$ oc get route console -n openshift-console \
-o jsonpath='{.spec.host}{ "\n"}'
console.openshift-console.apps.cluster.domain.example.com
```

- 5.2. Log in as the developer user.

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 interface with the 'Workloads' tab selected. The top navigation bar includes tabs for Overview, Details, YAML, Workloads (which is highlighted with a red box), and Role Bindings. Below the tabs are three input fields: 'Display Options' with a dropdown arrow, 'Filter by Resource' with a dropdown arrow, and a search bar labeled 'Find by name...' with a magnifying glass icon. The main content area is titled 'Deployment Config' and lists one item: 'DC scale'. The 'scale' entry is preceded by a blue circle containing 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.

The screenshot shows the Red Hat OpenShift Project Details interface. The top navigation bar includes 'Projects' > 'Project Details' and a 'PR youruser-manage-deploy' section with an 'Active' status and an 'Actions' dropdown. Below this is a navigation bar with tabs: 'Overview', 'Details', 'YAML', 'Workloads' (which is underlined), and 'Role Bindings'. A search bar with 'Group by: Application' and a filter input 'Filter by name...' is present. The main content area is titled 'scale' and shows a single pod named 'DC scale, #1'. A callout bubble indicates '1 of 1 pods'. Below the pod list is a note: '1 and 2 selects items, and 3 filters items.' To the right, a 'Health Checks' section is shown with a warning message: 'Container scale does not have health checks to ensure your application is running correctly. Add Health Checks'. A detailed view of the pod 'scale' is open, showing 'Details', 'Resources', and 'Monitoring' tabs. The pod details table has one row: Name 'scale' and Latest Version '1'. A red box highlights the 'scale' name and the '1 pod' count.

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...
```

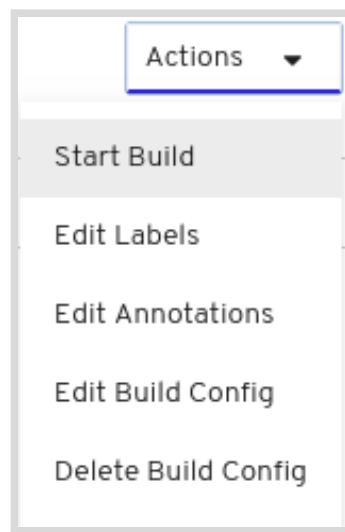
7. Start a new build of the application.

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:

```
[student@workstation D0288-apps]$ oc get pods
NAME READY STATUS RESTARTS AGE
scale-1-build 0/1 Completed 0 40m
scale-1-deploy 0/1 Completed 0 36m
scale-2-build 0/1 Completed 0 10m
scale-2-deploy 0/1 Completed 0 6m59s
scale-3-bcxpg 1/1 Running 0 58s
scale-3-deploy 0/1 Completed 0 77s
scale-3-lnp46 1/1 Running 0 68s
```

### 8.3. 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 14

# 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.

### Sections

- Integrating External Services
- Integrating External Services
- Deploying Cloud-Native Applications with JKube
- Deploying Cloud-Native Applications with JKube

### Lab

- Lab: Building 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](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](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:

```
[student@workstation ~]$ oc get pod
NAME READY STATUS RESTARTS AGE
todoapp-1-6z6qg 1/1 Running 0 1m
todoapp-1-build 0/1 Completed 0 4m
todoapp-1-deploy 0/1 Completed 0 1m
```

- 1.6. Inspect the environment variables inside the application pod to get the database connection parameters:

```
[student@workstation ~]$ oc rsh todoapp-1-6z6qg env | grep DATABASE
DATABASE_PASSWORD=redhat123
DATABASE_SVC=tododb
DATABASE_USER=todoapp
DATABASE_INIT=false
DATABASE_NAME=todo
```

- 2. Verify that the To Do List application cannot reach the database server specified by the `DATABASE_SVC` environment variable, which is `tododb`.
- 2.1. Get the host name where the application is exposed. Because the host name is very long, save it into a shell variable.

```
[student@workstation ~]$ HOSTNAME=$(oc get route todoapp \
-o jsonpath='{.spec.host}')
[student@workstation ~]$ echo ${HOSTNAME}
todoapp-youruser-external-service.apps.cluster.domain.example.com
```

- 2.2. Use the `curl` command, the host name from the previous step, and the API resource path `/todo/api/items/6` to fetch an item from the database. The error message from application indicates that it cannot resolve the `tododb` service host name.

```
[student@workstation ~]$ curl -si http://${HOSTNAME}/todo/api/items/6
HTTP/1.1 500 Internal Server Error
...output omitted...
{"message":"getaddrinfo ENOTFOUND tododb tododb:3306"}
```

► 3. Inspect the external database.

- 3.1. Find the host name of the external MariaDB server. This host name is the same as your OpenShift cluster's wildcard domain, replacing 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.

# 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.

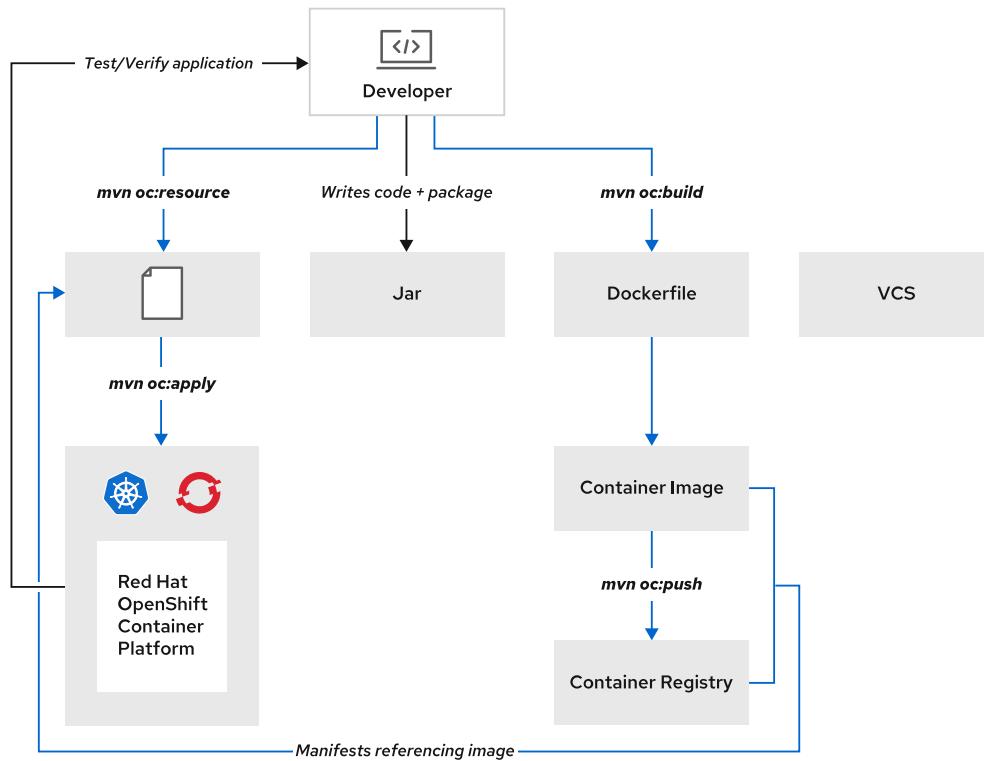
### Micronaut

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 14.1: 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, service</code>         |
| Route            | <code>route</code>                |
| Deployment       | <code>deployment</code>           |
| DeploymentConfig | <code>dc, deploymentconfig</code> |
| ConfigMap        | <code>cm, 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.

- 1.1. Load your classroom environment configuration. Run the following command to load the environment variables created in the first guided exercise:

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

- 1.2. Log in to OpenShift using your developer username.

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

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

```
[student@workstation micro-java]$ oc new-project ${RHT_OCP4_DEV_USER}-micro-java
Now using project "youruser-micro-java" on server ...
...output omitted...
```

- 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 `D0288-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/D0288/solutions/todo-migrate` directory. You can copy it to the `src/main/jkube` subdirectory:

```
[student@workstation todo-api]$ cp \
~/D0288/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 15

# Comprehensive Review: Containers, Kubernetes, and Red Hat OpenShift Development II

### Goal

Review tasks from *Containers, Kubernetes, and Red Hat OpenShift Development II*

### Objectives

Review tasks from *Containers, Kubernetes, and Red Hat OpenShift Development II*

### 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 *Containers, Kubernetes, and Red Hat OpenShift Development II*.

## Reviewing Containers, Kubernetes, and Red Hat OpenShift Development II

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 7, 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 8, 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 9, 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 10, 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 11, 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.

## **Chapter 12, 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.

## **Chapter 13, 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 14, 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.

## ► 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 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
```

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-bqvz 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:

| 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:

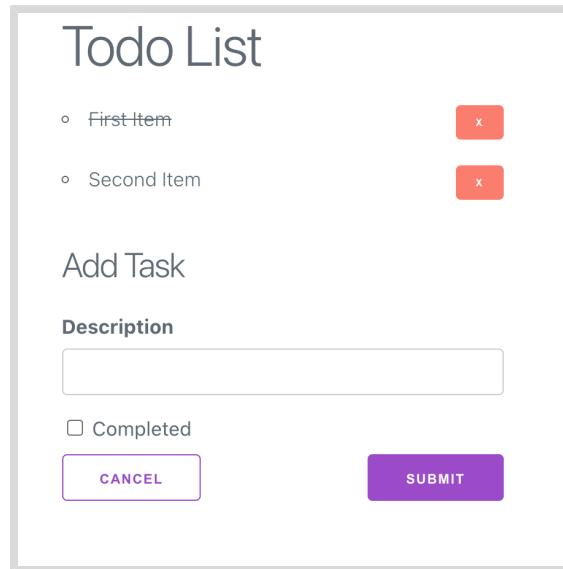
```
[student@workstation ~]$ oc expose svc/todo-frontend
route.route.openshift.io/todo-frontend exposed
```

- 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 for the labs in the course.

### Creating a GitHub Account

You need a GitHub account to create one or more *public* Git repositories for the labs in this course. If you already have a GitHub account, you can skip the steps listed in this appendix.

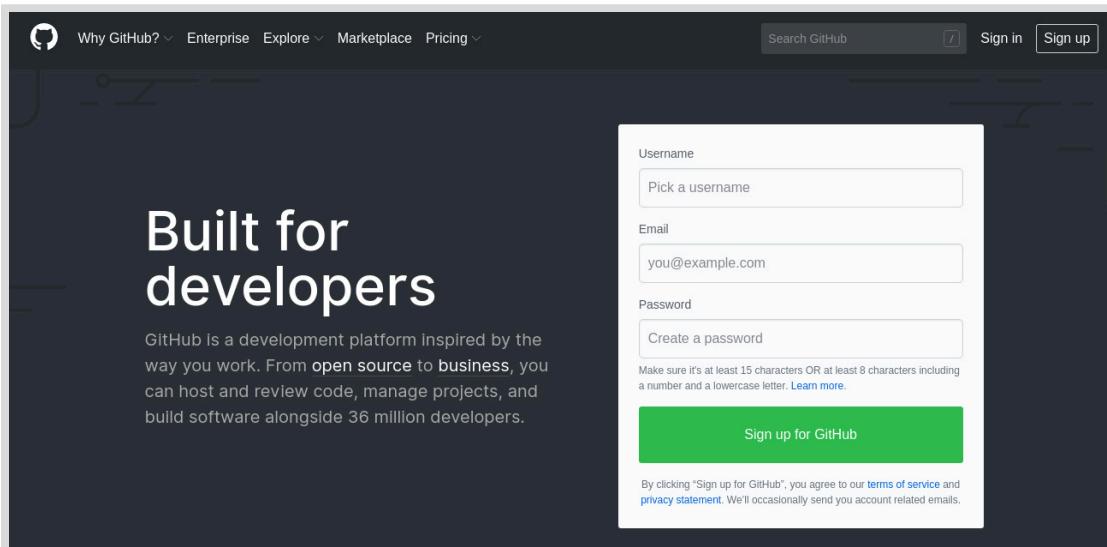


#### Important

If you already have a GitHub account, ensure that you only create *public* Git repositories for the labs in this course. The lab grading scripts and instructions require unauthenticated access to clone the repository. The repositories must be accessible without providing passwords, SSH keys, or GPG keys.

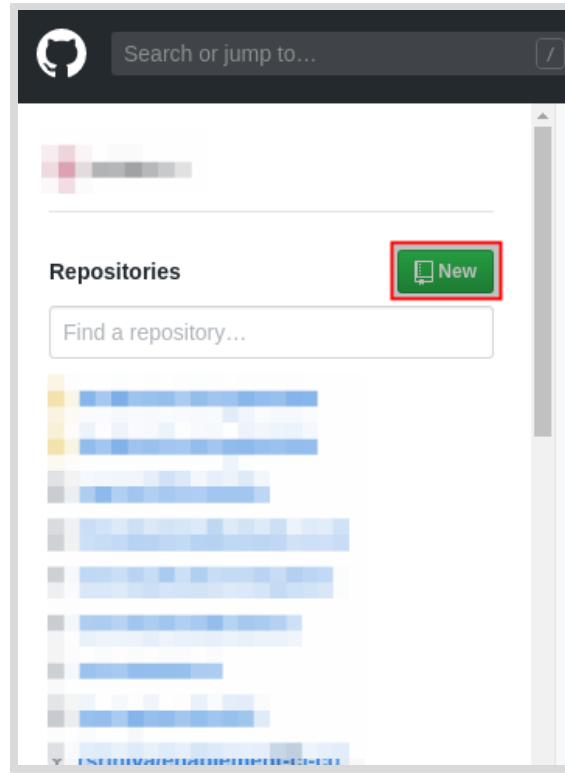
To create a new GitHub account, perform the following steps:

1. Navigate to <https://github.com> using a web browser.
2. Enter the required details and then click **Sign up for GitHub**.



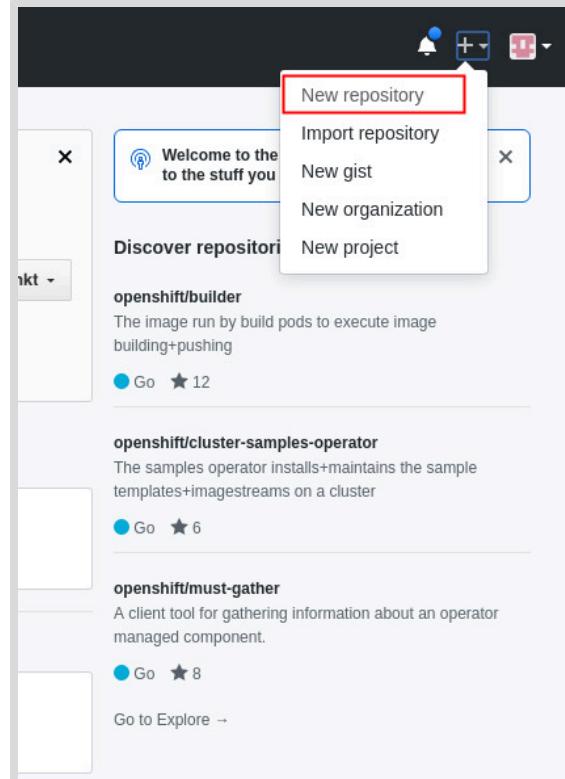
**Figure A.1: Creating a GitHub account**

3. You will receive an email with instructions on how to activate your GitHub account. Verify your email address and then sign in to the GitHub website using the username and password you provided during account creation.
4. After you have logged in to GitHub, you can create new Git repositories by clicking **New** in the **Repositories** pane on the left of the GitHub home page.



**Figure A.2: Creating a new Git repository**

Alternatively, click the plus icon (+) in the upper-right corner (to the right of the bell icon) and then click **New repository**.



**Figure A.3: Creating new Git repository**



## References

### **Signing up for a new GitHub account**

<https://help.github.com/en/articles/signing-up-for-a-new-github-account>

## Appendix B

# Creating a Quay Account

### Goal

Describe how to create a Quay account for labs in the course.

# Creating a Quay Account

## Objectives

After completing this section, you should be able to create a Quay account and create public container image repositories for the labs in the course.

### Creating a Quay Account

You need a Quay account to create one or more *public* container image repositories for the labs in this course. If you already have a Quay account, you can skip the steps to create a new account listed in this appendix.

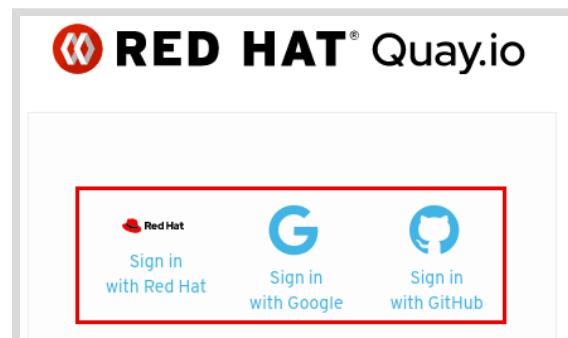


#### Important

If you already have a Quay account, ensure that you only create *public* container image repositories for the labs in this course. The lab grading scripts and instructions require unauthenticated access to pull container images from the repository.

To create a new Quay account, perform the following steps:

1. Navigate to <https://quay.io> using a web browser.
2. Click **Sign in** in the upper-right corner (next to the search bar).
3. On the **Sign in** page, you can log in using your Red Hat (created in Appendix D), Google or GitHub credentials (created in Appendix A).



**Figure B.1: Sign in using Google or GitHub credentials.**

Alternatively, click **Create Account** to create a new account.

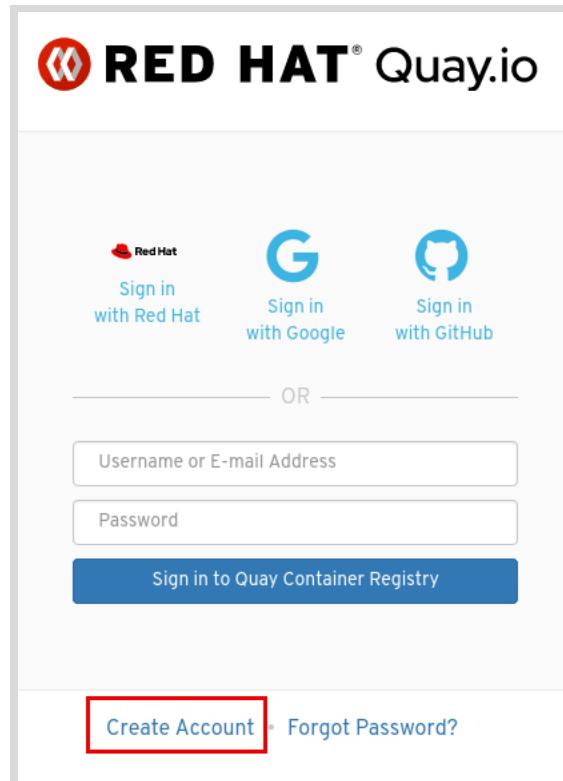


Figure B.2: Creating a new account

1. If you chose to skip the Red Hat, Google or GitHub login method and instead opted to create a new account, you will receive an email with instructions on how to activate your Quay account. Verify your email address and then sign in to the Quay website with the username and password you provided during account creation.
2. After you have logged in to Quay you can create new image repositories by clicking **Create New Repository** on the **Repositories** page.

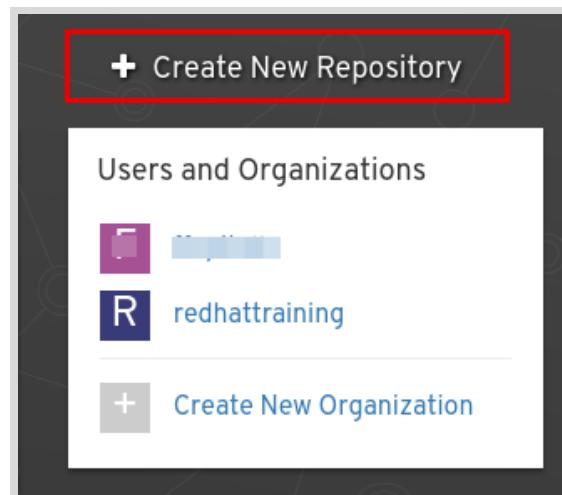


Figure B.3: Creating a new image repository

Alternatively, click the plus icon (+) in the upper-right corner (to the left of the bell icon), and then click **New Repository**.

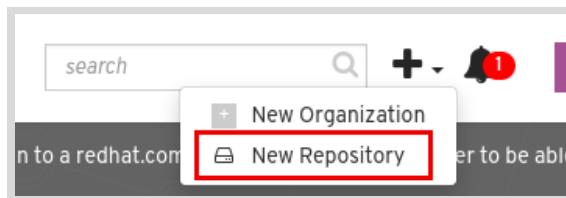


Figure B.4: Creating a new image repository

## Working with CLI tools

If you created your account with Red Hat, Google or Github, you need to set an account password to use CLI tools like Podman or Docker.

1. Click <YOUR\_USERNAME> in the upper-right corner.
2. Click **Account Settings**.
3. Click the *Change password* link.



### References

#### Getting Started with Quay.io

<https://docs.quay.io/solution/getting-started.html>

# Repository Visibility

---

## Objectives

After completing this section, you should be able to control repository visibility on Quay.io.

## Quay.io Repository Visibility

Quay.io offers the possibility of creating public and private repositories. Public repositories can be read by anyone without restrictions, despite write permissions must be explicitly granted. Private repositories have both read and write permissions restricted. Nevertheless, the number of private repositories in quay.io is limited, depending on the namespace plan.

## Default Repository Visibility

Repositories created by pushing images to quay.io are private by default. In order OpenShift (or any other tool) to fetch those images you can either configure a private key in both OpenShift and Quay, or make the repository public, so no authentication is required. Setting up private keys is out of scope of this document.

The screenshot shows the Red Hat Quay.io interface. At the top, there is a navigation bar with icons for back, forward, and refresh, followed by a lock icon and the URL <https://quay.io/repository/>. Below the navigation bar, the Red Hat Quay.io logo is displayed, along with menu options: EXPLORE, APPLICATIONS, REPOSITORIES (which is highlighted in red), and TUTORIAL. The main content area has a dark background with a network-like graphic. The title "Repositories" is centered at the top of the content area. On the left, there is a section titled "Starred" with an orange star icon. Below it, a message says "You haven't starred any repositories yet." with a smaller subtitle "Stars allow you to easily access your favorite repositories." To the right of the message are two small square icons: one with a grid of squares and another with a grid of three squares. Below this section, there is a heading "RHT\_OCP4\_QUAY\_USER" with a small brown square icon. Underneath this heading are four repository cards arranged in a 2x2 grid. Each card contains a small icon, the repository name, and a star icon. The repository names are: "do180-mysql-57-rhel7", "do180-todonodejs", "do180-quote-php", and "nexus". The "nexus" card has a red lock icon next to its name.

## Updating Repository Visibility

In order to set repository visibility to public select the appropriate repository in <https://quay.io/repository/> (log in to your account if needed) and open the **Settings** page by clicking on the gear icon on the bottom left. Scroll down to the **Repository Visibility** section and click the **Make Public** button.

The screenshot shows the Quay.io repository settings interface. At the top, a banner indicates "Trust Disabled" with the message "Signing is disabled on this repository." and a "Enable Trust" button. Below this is the "Events and Notifications" section, which states "No notifications have been setup for this repository." and includes a "Create Notification" button. The "Repository Visibility" section shows the repository is "private" and has a "Make Public" button. The "Delete Repository" section contains a warning message: "Deleting a Repository cannot be undone. Here be dragons!" and a red "Delete Repository" button.

Get back the the list of repositories. The lock icon besides the repository name have disappeared, indicating the repository became public.



## Appendix C

# Creating a Red Hat Account

### Goal

Describe how to create a Red Hat account for labs in the course.

# Creating a Red Hat Account

## Objectives

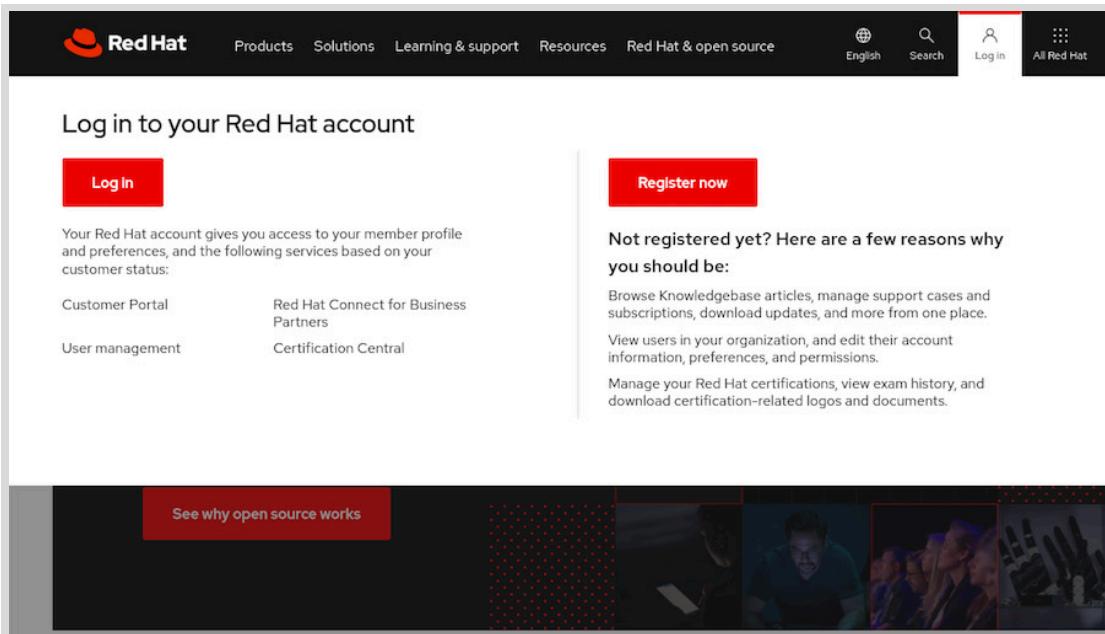
After completing this section, you should be able to create a Red Hat account to access the Red Hat Container Catalog images for the labs in the course.

### Creating a Red Hat Account

You need a Red Hat account to access the Red Hat Container Catalog images. If you already have a Red Hat account, you can skip this steps.

To create a new Red Hat account, perform the following steps:

1. Navigate to <https://redhat.com> using a web browser.
2. Click **Log in** in the upper-right corner.
3. Click **Register now** on the right side pane.



**Figure C.1: Register a new account**

4. Set Account type as Personal.

The screenshot shows the 'Create a Red Hat account' page. At the top is the Red Hat logo. Below it is the heading 'Create a Red Hat account'. A sub-instruction says 'Sign up and use this Red Hat account to access all of Red Hat's applications, communities, support, and more.' A note below states: 'Red Hat will collect your contact and account information to create your Red Hat account. We use your personal data to identify you and to provide you with information, support, and customer service. For more information, please see Red Hat's [privacy statement](#)'. A section titled 'Join an existing account' provides instructions on how to access subscriptions through an existing account or contact customer service. A note at the bottom right indicates '\* Required fields'. The main form area is titled 'Account type' with the instruction 'Choose account type \*'. It contains two radio button options: 'Corporate' (unchecked) and 'Personal' (checked). A descriptive text for 'Corporate' reads: 'A corporate Red Hat account allows a set of users within your organization to centrally make purchases or administer systems.' A descriptive text for 'Personal' reads: 'A personal Red Hat account is for purchasing or administering your own personal systems.'

**Figure C.2: Set account type**

5. Fill in the remainder of the form with your personal information. In this form you also select the username and password for this account.
6. Click **CREATE MY ACCOUNT**.

The screenshot shows the 'Fill in personal information' step of the account creation process. It features two input fields: 'City \*' with a placeholder 'Enter city' and 'State/Province \*' with a placeholder 'Enter state/province'. Below these fields is a large red button labeled 'CREATE MY ACCOUNT'. At the bottom of the form is a link '« Back to login page'.

**Figure C.3: Fill in personal information form**



## References

### Red Hat Customer Portal

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



## Appendix D

# Useful Git Commands

### Goal

Describe useful Git commands that are used for the labs in this course.

# Git Commands

## Objectives

After completing this section, you should be able to restart and redo exercises in this course. You should also be able to switch from one incomplete exercise to perform another, and later continue the previous exercise where you left off.

## Working with Git Branches

This course uses a Git repository hosted on GitHub to store the application course code source code. At the beginning of the course, you create your own fork of this repository, which is also hosted on GitHub.

During this course, you work with a local copy of your fork, which you clone to the **workstation** VM. The term **origin** refers to the remote repository from which a local repository is cloned.

As you work through the exercises in the course, you use separate Git branches for each exercise. All changes you make to the source code happen in a new branch that you create only for that exercise. Never make any changes on the **master** branch.

A list of scenarios and the corresponding Git commands that you can use to work with branches, and to recover to a known good state are listed below.

## Redoing an Exercise from Scratch

To redo an exercise from scratch after you have completed it, perform the following steps:

1. You commit and push all the changes in your local branch as part of performing the exercise. You finished the exercise by running its **finish** subcommand to clean up all resources:

```
[student@workstation ~]$ lab your-exercise finish
```

2. Change to your local clone of the DO180-apps repository and switch to the **master** branch:

```
[student@workstation ~]$ cd ~/DO180-apps
[student@workstation DO180-apps]$ git checkout master
```

3. Delete your local branch:

```
[student@workstation DO180-apps]$ git branch -d your-branch
```

4. Delete the remote branch on your personal GitHub account:

```
[student@workstation DO180-apps]$ git push origin --delete your-branch
```

5. Use the **start** subcommand to restart the exercise:

```
[student@workstation D0180-apps]$ cd ~
[student@workstation ~]$ lab your-exercise start
```

## Abandoning a Partially Completed Exercise and Restarting it from Scratch

You may run into a scenario where you have partially completed a few steps in the exercise, and you want to abandon the current attempt, and restart it from scratch. Perform the following steps:

1. Run the exercise's `finish` subcommand to clean up all resources.

```
[student@workstation ~]$ lab your-exercise finish
```

2. Enter your local clone of the D0180-apps repository and discard any pending changes on the current branch using `git stash`:

```
[student@workstation ~]$ cd ~/D0180-apps
[student@workstation D0180-apps]$ git stash
```

3. Switch to the `master` branch of your local repository:

```
[student@workstation D0180-apps]$ git checkout master
```

4. Delete your local branch:

```
[student@workstation D0180-apps]$ git branch -d your-branch
```

5. Delete the remote branch on your personal GitHub account:

```
[student@workstation D0180-apps]$ git push origin --delete your-branch
```

6. You can now restart the exercise by running its `start` subcommand:

```
[student@workstation D0180-apps]$ cd ~
[student@workstation ~]$ lab your-exercise start
```

## Switching to a Different Exercise from an Incomplete Exercise

You may run into a scenario where you have partially completed a few steps in an exercise, but you want to switch to a different exercise, and revisit the current exercise at a later time.

Avoid leaving too many exercises uncompleted to revisit later. These exercises tie up cloud resources and you may use up your allotted quota on the cloud provider and on the OpenShift cluster you share with other students. If you think it may be a while until you can go back to the current exercise, consider abandoning it and later restarting from scratch.

If you prefer to pause the current exercise and work on the next one, perform the following steps:

1. Commit any pending changes in your local repository and push them to your personal GitHub account. You may want to record the step where you stopped the exercise:

**Appendix D |** Useful Git Commands

```
[student@workstation ~]$ cd ~/D0180-apps
[student@workstation D0180-apps]$ git commit -a -m "Paused at step X.Y"
[student@workstation D0180-apps]$ git push
```

2. Do not run the `finish` command of the original exercise. This is important to leave your existing OpenShift projects unchanged, so you can resume later.
3. Start the next exercise by running its `start` subcommand:

```
[student@workstation ~]$ lab your-exercise start
```

4. The next exercise switches to the `master` branch and optionally creates a new branch for its changes. This means the changes made to the original exercise in the original branch are left untouched.
5. Later, after you have completed the next exercise, and you want to go back to the original exercise, switch back to its branch:

```
[student@workstation ~]$ git checkout original-branch
```

Then you can continue with the original exercise at the step where you left off.

**Note**

Git branch man page [<https://git-scm.com/docs/git-branch>]

What is a Git branch? [<https://git-scm.com/book/en/v2/Git-Branching-Banches-in-a-Nutshell>]

Git Tools - Stashing [<https://git-scm.com/book/en/v2/Git-Tools-Stashing-and-Cleaning>]