

Chapter 1. Introducing Container Technology

[Overview of Container Technology](#)

[Quiz: Overview of Container Technology](#)

[Overview of Container Architecture](#)

[Quiz: Overview of Container Architecture](#)

[Overview of Kubernetes and OpenShift](#)

[Quiz: Describing Kubernetes and OpenShift](#)

[Guided Exercise: Configuring the Classroom Environment](#)

[Summary](#)

Abstract

Goal	Describe how applications run in containers orchestrated by Red Hat OpenShift Container Platform.
Objectives	<ul style="list-style-type: none">• 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	<ul style="list-style-type: none">• 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 containerized 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 closely related to 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: Container versus operating system differences](#) 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: Container versus operating system differences](#).

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 (control groups). 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 might be unreliable due to container limitations.

Quiz: Overview of Container Technology

Choose the correct answers to the following questions:

1.

2.

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.

3. CheckResetShow Solution

4.

5.

- 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.

6. CheckResetShow Solution

7.

8.

- 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.

9. CheckResetShow Solution

10.

11.

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

12. CheckResetShow Solution

[Previous](#) [Next](#)

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 that lists 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](#)
- [Red Hat Quay](#)
- [Docker Hub](#)
- [Google Container Registry](#)
- [Amazon Elastic Container Registry](#)

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, container images and interacting with image registries. It offers the following key features:

- It uses image format specified by the [Open Container Initiative](#) (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.

Quiz: Overview of Container Architecture

Choose the correct answers to the following questions:

1.

2.

- 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

3. CheckResetShow Solution

4.

5.

- 2.** Which of the following best describes a container image?

- A A virtual machine image from which a container is created.
- B A file-system bundle that contains all dependencies required to execute the process inside the container.
- C A runtime environment where an application will run.
- D The container's index file used by a registry.

6. CheckResetShow Solution

7.

8.

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

9. CheckResetShow Solution

10.

11.

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.

12.CheckResetShow Solution

13.

14.

5. Which of
the
following
are
Podman
features
(Choose
two.)

- A Manage operating system configuration and permissions to execute virtual machines.
- B Manage containers, container images and interact with registries.
- C Execute a daemon on the local machine to run containers.
- D Podman uses the same command patterns as Docker.

15.CheckResetShow Solution

[Previous](#) [Next](#)

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 (RHOC).

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 their 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 pods 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 lifecycle 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 (RHOC) is a set of modular components and services built on top of a Kubernetes container infrastructure. RHOC 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 RHOC 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

RHOC 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](#)

[OpenShift: Container Application Platform by Red Hat, Built on Containers and Kubernetes](#)

[Previous](#) [Next](#)

Quiz: Describing Kubernetes and OpenShift

Choose the correct answers to the following questions:

1.

2.

- 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 internal application failures.

D Containers are not load-balanced.

E Containers are heavily-isolated, packaged applications.

3. CheckResetShow Solution

4.

5.

- 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.

6. CheckResetShow Solution

7.

8.

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.

9. CheckResetShow Solution

10.

11.

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.

12. CheckResetShow Solution

[Previous](#) [Next](#)

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.

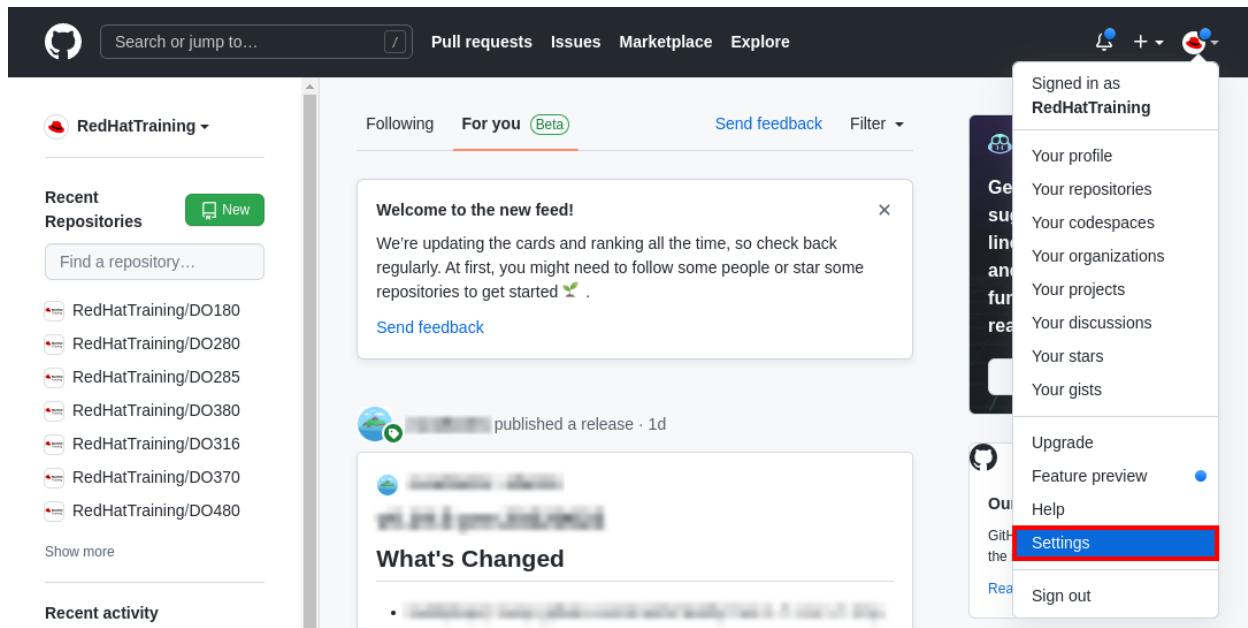
To perform this exercise, ensure you have:

- Access to the DO180 course in the Red Hat Training's Online Learning Environment.
- The connection parameters and a developer user account to access the OpenShift cluster provided in your classroom environment.
- A personal, free GitHub account. If you need to register to GitHub, see the instructions in [Appendix B, Creating a GitHub Account](#).
- A personal, free Quay.io account. If you need to register to Quay.io, see the instructions in [Appendix C, Creating a Quay Account](#).
- A personal, GitHub access token.

Procedure 1.1. Instructions

Before starting any exercise, ensure you have:

1. Prepare your GitHub access token.
 1. Navigate to <https://github.com> using a web browser and authenticate.
 2. On the top of the page, click your profile icon, select the **Settings** menu



3. Figure 1.2: User menu
4. Select **Developer settings** in the left pane of the page.

The screenshot shows the GitHub developer settings page. On the left, there's a sidebar with various settings categories: Profile, Account, Appearance (which has a 'New' badge), Account security, Billing & plans, Security log, Security & analysis, Emails, Notifications, SSH and GPG keys, Repositories, Packages, Organizations, Saved replies, Applications, and Developer settings. The 'Developer settings' link is highlighted with a red box. The main content area contains sections for Name, Public email, Bio, URL, Twitter username, Company, and a note about linking to a company organization.

5. Figure 1.3: Developer settings

6. 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.

The screenshot shows the GitHub developer settings page. The left sidebar has three options: GitHub Apps, OAuth Apps, and Personal access tokens, with Personal access tokens selected and highlighted by a red box. The main content area is titled "Personal access tokens". It contains a note: "Need an API token for scripts or testing? [Generate a personal access token](#) for quick access to the GitHub API." Below this, a paragraph explains that personal access tokens function like ordinary OAuth access tokens and can be used instead of a password for Git over HTTPS or for API authentication. A "Generate new token" button is located in the top right corner of this section, also highlighted by a red box.

7. Figure 1.4: Personal access token pane

7. Figure 1.4: Personal access token pane
8. Write a short description of your new access token on the Note field.
9. 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](#).

The screenshot shows the "New personal access token" configuration page. It has two main sections: "Note" and "Select scopes".
The "Note" section contains a text input field with the placeholder "Course DO180" and a question "What's this token for?" Both the input field and the question text are highlighted by a red box.
The "Select scopes" section lists several OAuth scopes with checkboxes:

- repo: Full control of private repositories
- repo:status: Access commit status
- repo_deployment: Access deployment status
- public_repo: Access public repositories (this option is highlighted by a red box)
- repo:invite: Access repository invitations
- security_events: Read and write security events

10. Figure 1.5: Personal access token configuration

10. Figure 1.5: Personal access token configuration
11. 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

[Generate new token](#)

[Revoke all](#)

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

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

✓ ghp_kgYGzWcGE1crdovkzuzeLTWvYY6eBX2l0vCK [Copy](#)

[Delete](#)

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

12. Figure 1.6: Generated access token

13. 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
```

Note

The default timeout is 900 seconds, you can specify a different timeout if needed.

```
[student@workstation ~]$ git config --global credential.helper \
'cache --timeout=3600'
```

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.

1. Configure a password for your Quay.io account. On the *Account Settings* page, click the *Change password* link. See [Appendix C, Creating a Quay Account](#) for further details.
3. Configure the workstation machine.

Open a terminal on the workstation machine and execute the following command.

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

Answer the interactive prompts before starting any other exercise in this course.

Note

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

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

1. The `lab-configure` command starts by displaying a series of interactive prompts and uses sensible defaults when they are available.
2. [student@workstation ~]\$ `lab-configure`
- 3.
4. · Enter the GitHub account name: `yourgituser`
5. Verifying GitHub account name: `yourgituser`
- 6.
7. · Enter the Quay.io account name: `yourquayuser`
8. Verifying Quay.io account name: `yourquayuser`
- 9.

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

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 B, Creating a GitHub Account](#) and [Appendix C, Creating a Quay Account](#) for instructions about how to register.

10. If `lab-configure` finds any issues, it displays an error message and exits.

You must verify your information and run the `lab-configure -d` command again. The following listing shows an example of a verification error.

```
11. · Enter the Quay.io account name: notexists  
12. Verifying Quay.io account name: notexists  
13. ERROR: Cannot find Quay.io account user 'notexists'  
14.
```

To reconfigure, run: `lab-configure -d`

15. Finally, the `lab-configure` command verifies that the developer user can log in to your OpenShift cluster.

```
16. ....output omitted...  
17.  
18. . Ensuring user 'developer' can log in to the OpenShift cluster.  
19.
```

....output omitted...

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

21. If there are 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:

1. Open a web browser and navigate to the DO180-apps GitHub repository.
 - <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 GitHub interface for the repository 'RedHatTraining / DO180-apps'. At the top, there's a navigation bar with links for 'Why GitHub?', 'Enterprise', 'Explore', 'Marketplace', 'Pricing', a search bar, and buttons for 'Sign in' and 'Sign up'. Below the navigation is the repository header with the name 'RedHatTraining / DO180-apps' and icons for 'Watch' (4), 'Star' (0), and 'Fork' (0). A navigation bar below the header includes 'Code', 'Issues 0', 'Pull requests 0', 'Projects 0', 'Security', and 'Insights'. The main content area displays the repository's files and structure.

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

The screenshot shows the GitHub sign-in page. It features a large GitHub logo at the top, followed by the text 'Sign in to GitHub'. There are two input fields: 'Username or email address' containing 'yourgituser' and 'Password' with a redacted value. To the right of the password field is a 'Forgot password?' link. A green 'Sign in' button is at the bottom of the form.

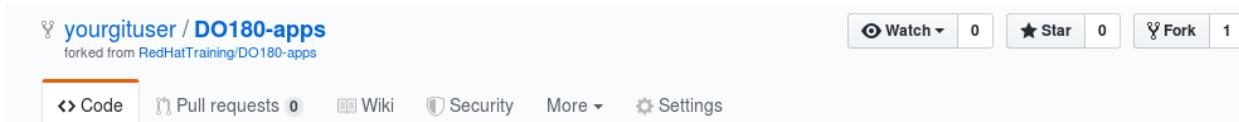
3. Navigate to the RedHatTraining/D0180-apps repository and click **Fork** in the top right corner.

The screenshot shows the GitHub repository page for 'RedHatTraining / DO180-apps'. The top navigation bar includes 'Code', 'Issues 0', 'Pull requests 0', 'Actions', 'Wiki', 'Security', and 'More'. In the top right corner, there are buttons for 'Watch' (4), 'Star' (0), and 'Fork' (0).

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

The screenshot shows a modal dialog box titled 'Fork D0180-apps'. The question 'Where should we fork DO180-apps?' is displayed. Below it, a dropdown menu is open, showing the option 'yourgituser' selected. On the left side of the dialog, there's a preview of the repository 'DO180 Repository for Sample Applications'.

5. Important
6. 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.
7. After a few minutes, the GitHub web interface displays your new repository `yourgituser/DO180-apps`.

A screenshot of a GitHub repository page for 'yourgituser/DO180-apps'. The page shows basic statistics: 0 watch, 0 stars, and 1 fork. Navigation links include 'Code', 'Pull requests 0', 'Wiki', 'Security', 'More ▾', and 'Settings'.

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

1. Run the following command to clone this course's sample applications repository. Replace *yourgituser* with the name of your personal GitHub account.
2. [student@workstation ~]\$ git clone https://github.com/yourgituser/DO180-apps
3. Cloning into 'DO180-apps'...

...output omitted...

4. Verify that /home/student/DO180-apps is a Git repository.

5. [student@workstation ~]\$ cd DO180-apps
6. [student@workstation DO180-apps]\$ git status
7. # On branch master

nothing to commit, working directory clean

8. Create a new branch to test your new personal access token.

9. [student@workstation DO180-apps]\$ git checkout -b testbranch

Switched to a new branch **testbranch**

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

11. [student@workstation DO180-apps]\$ echo "DO180" > TEST
12. [student@workstation DO180-apps]\$ git add .
13. [student@workstation DO180-apps]\$ git commit -m "DO180"

...output omitted...

14. Push the changes to your recently created testing branch.

15. [student@workstation DO180-apps]\$ git push --set-upstream origin testbranch

```
16. Username for https://github.com:
```

```
17. Password for https://yourgituser@github.com:
```

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

Enter your GitHub username,

Enter your personal access token.

18. 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.

```
19. [student@workstation D0180-apps]$ echo "OCP4" > TEST
```

```
20. [student@workstation D0180-apps]$ git add .
```

```
21. [student@workstation D0180-apps]$ git commit -m "OCP4"
```

```
22. [student@workstation D0180-apps]$ git push
```

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

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

```
24. [student@workstation D0180-apps]$ head README.md
```

```
25. # D0180-apps
```

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

```
27.
```

```
28. [student@workstation D0180-apps]$ cd ~
```

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

Finish

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 E, Useful Git Commands](#).

This concludes the guided exercise.

[Previous](#) [Next](#)

Summary

In this chapter, you learned:

- Containers are isolated application runtimes, 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.

[Previous](#) [Next](#)

Chapter 2. Creating Containerized Services

Provisioning Containerized Services

Guided Exercise: Creating a MySQL Database Instance

Using Rootless Containers

Guided Exercise: Exploring Root and Rootless Containers

Lab: Creating Containerized Services

Summary

Abstract

Goal	Provision a service using container technology.
Objectives	<ul style="list-style-type: none">Create a database server from a container image.
Sections	<ul style="list-style-type: none">Provisioning a Containerized Database Server (and Guided Exercise)Using Rootless Containers (and Guided Exercise)
Lab	<ul style="list-style-type: none">Creating Containerized Services

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
ff4ec6d74e9b2a7b55c49f138e56f8bc46fe2a09c23093664fea7febcd3dfa1b2
[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 `-1` flag (`1` 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:

Container images

Container images offer lightweight and self-contained software to enable deployment at scale.

The screenshot shows the Red Hat Container Catalog search interface. At the top, there is a search bar containing "Apache httpd" and a red "Search" button. Below the search bar, there are three filter categories: "Provider" (IBM, Red Hat, Inc.), "Category" (Database & Data Management, Programming Languages & Runtimes, Web Services), and "Product". The main content area displays three container image cards:

- Red Hat** Apache httpd 2.4 (rhscl/httpd-24-rhel7) by Red Hat, Inc. Platform for running Apache httpd 2.4 or building httpd-based application. Updated 11 hours ago.
- Red Hat** Apache httpd 2.4 (rhel8/httpd-24) by Red Hat, Inc. Platform for running Apache httpd 2.4 or building httpd-based application. Updated 11 hours ago.
- IBM** Apache CouchDB (ibm/couchdb3) by IBM Apache CouchDB is a database that uses JSON for documents, an HTTP API, & JavaScript/... Updated 4 days ago.

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

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 `rhe18/httpd-24` from the suggested list.

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

Standalone Image

Apache httpd 2.4

rhel8/httpd-24

Provided by



Architecture: amd64 Tag: latest

A screenshot of a web-based container image management interface. At the top, there's a navigation bar with links for Home, Software, Container images, and Apache httpd 2.4. Below this is a section titled "Standalone Image" with the heading "Apache httpd 2.4". Underneath the heading is the text "rhel8/httpd-24". On the right side, there's a "Provided by" section with the Red Hat logo. Below the heading are two dropdown menus: "Architecture" set to "amd64" and "Tag" set to "latest". The main content area has a header "latest" with a badge showing "1" and a timestamp "1-123.1614609239". Below this are several tabs: "Overview" (which is underlined), "Security", "Packages", "Dockerfile", and "Get this image". The "Overview" tab contains sections for "Description", "Published", "Release category", "Usage in OpenShift", and "Health index". The "Description" section contains a detailed paragraph about Apache HTTP Server 2.4. The "Published" section shows "28 days ago". The "Release category" section shows "Generally Available". The "Usage in OpenShift" section contains a note about using the image through the "httpd:24" imagestream tag. The "Health index" section shows a green box with "B" and "0" and a small info icon. On the far right, there's 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.

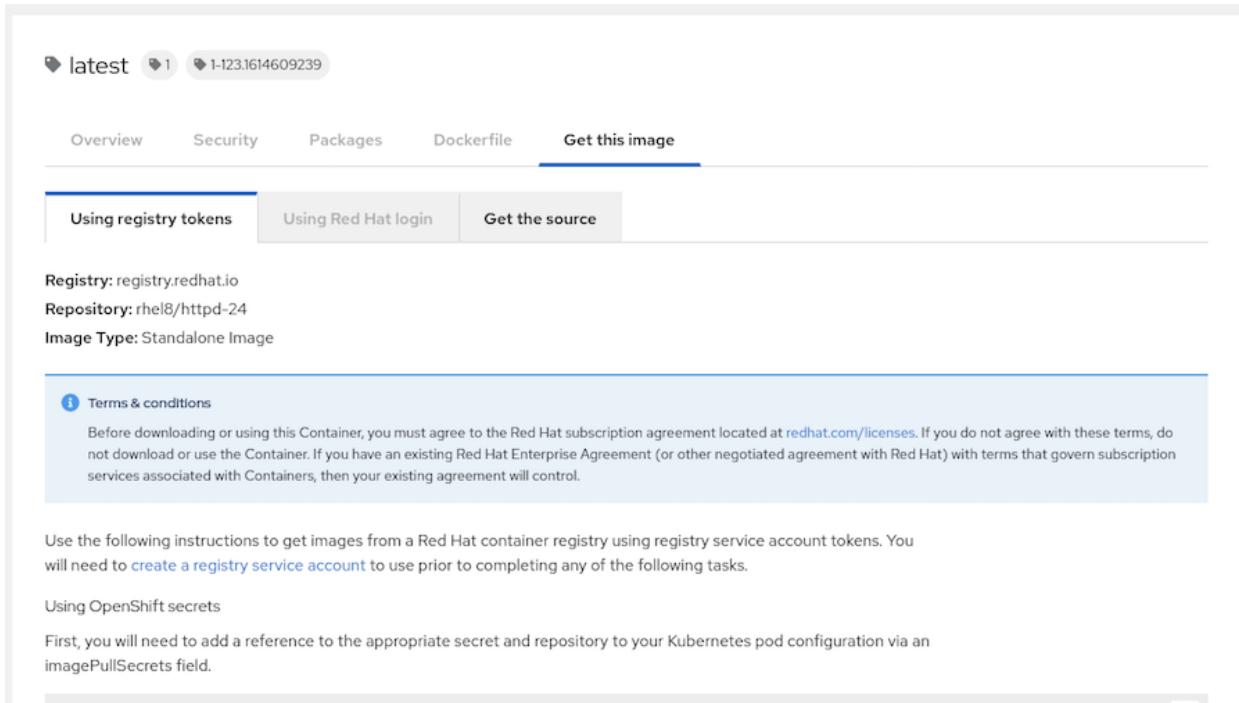


Figure 2.3: Apache httpd 2.4 (rhel8/httpd-24) latest image page

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](#)

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.

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

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

Procedure 2.1. Instructions

1. Create a MySQL container instance.
 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 D, Creating a Red Hat Account](#).
 2. [student@workstation ~]\$ podman login registry.redhat.io
 3. Username: `your_username`
 4. Password: `your_password`

Login Succeeded!

5. Start a container from the Red Hat Container Catalog MySQL image.
 6. [student@workstation ~]\$ podman run --name mysql-basic \
7. > -e MYSQL_USER=user1 -e MYSQL_PASSWORD=mypa55 \
8. > -e MYSQL_DATABASE=items -e MYSQL_ROOT_PASSWORD=r00tpa55 \
9. > -d registry.redhat.io/rhel8/mysql-80:1
10. Trying to pull ...*output omitted*...
11. Copying blob ...*output omitted*...
12. Writing manifest to image destination
13. Storing signatures

```
2d37682eb33a70330259d6798bdfdc37921367f56b9c2a97339d84faa3446a03
```

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.

14. Verify that the container started without errors.

```
15. [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:

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

```
bash-4.4$
```

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

4. Add data to the database.

1. Connect to MySQL as the database administrator user (`root`).

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

```
bash-4.4$ 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)
```

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

```
3. mysql> USE items;
```

```
Database changed
```

4. Create a table called `Projects` in the `items` database.

```
5. mysql> CREATE TABLE Projects (id int NOT NULL,
6.   -> name varchar(255) DEFAULT NULL,
7.   -> code varchar(255) DEFAULT NULL,
8.   -> 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.

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

```
10. mysql> SHOW TABLES;
11. +-----+
12. | Tables_in_items      |
13. +-----+
14. | Projects             |
15. +-----+
```

```
1 row in set (0.00 sec)
```

16. Use the `insert` command to insert a row into the table.

```
17. mysql> INSERT INTO Projects (id, name, code) VALUES (1, 'DevOps', 'D0180');
```

```
Query OK, 1 row affected (0.02 sec)
```

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

```
19. mysql> SELECT * FROM Projects;  
20. +----+-----+-----+  
21. | id | name      | code   |  
22. +----+-----+-----+  
23. | 1  | DevOps    | D0180  |  
24. +----+-----+-----+
```

```
1 row in set (0.00 sec)
```

25. Exit from the MySQL prompt and the MySQL container.

```
26. mysql> exit  
27. Bye  
28. bash-4.4$ 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.

[Previous](#) [Next](#)

Using Rootless Containers

Objectives

After completing this section, you should be able to:

- Explain the differences between running root and rootless containers.
- Describe the advantages and disadvantages of each case.
- Run as root and rootless containers with Podman.

Evolution of Container Usage

If you have been running containers for some time, chances are that you are running them as a privileged user. Historically, tools for container creation required that runtime engines run as root, and privileged access was necessary to create resources, such as network interfaces.

From a security perspective, providing this level of access is a bad practice. You should always run software with privileges that are as limited as possible. When a security bug is exploited, either on the runtime engine or the application itself, the impact is minimized.

A better practice is to containerize applications in such a way that they do not require a privileged user to run. These applications should use a well-known user instead.

Many community container images, such as those available at docker.io, still require root to run.

Some tools, such as Podman and Red Hat OpenShift, run rootless containers by default. Docker announced that rootless mode is generally available in version 20.10.

Advantages of Rootless Containers

Rootless containers are a new concept of containers that do not require root privileges to run. Rootless containers are advantageous from a security perspective for several reasons, including:

- Allows code to run inside a rootless container with root privileges, without having to run as the host's root user.
- Adds a new security layer; if the container engine is compromised, then the attacker will not gain root privileges on the host.
- Allows multiple unprivileged users to run containers on the same machine.
- Allows isolation inside nested containers.

Despite all of these advantages, running rootless containers has several limitations, including:

- Dropped capabilities
- Binding to ports less than 1024
- Volume mounting other content

There is a detailed list of rootless shortcomings in <https://github.com/containers/podman/blob/master/rootless.md>.

Understanding Rootless Containers

To understand how rootless containers work, consider the following concepts.

User Namespaces

Containers use Linux namespaces to isolate themselves from the host on which they run. In particular, the User namespace is used to make containers rootless. This namespace maps user and group IDs so that a process inside the namespace might appear to be running under a different ID.

Rootless containers use the User namespace to make application code appear to be running as root. However, from the host's perspective, permissions are limited to those of a regular user. If an attacker manages to escape the user namespace onto the host, then it will have only the capabilities of a regular, unprivileged user.

Networking

To allow proper networking inside a container, a Virtual Ethernet device is created. This poses a problem for rootless containers because only a real root user has the privileges to create this and similar devices.

On a rootless container, networking is usually managed by Slirp. It works by forking into the container's user and network namespaces and creating a tap device that becomes the default route. Then, it passes the device's file descriptor to the parent,

who runs in the default network namespace and can now communicate with both the container and the internet.

Storage

By default, container engines use a special driver called Overlay2 (or Overlay) to create a layered file system that is efficient in both capacity and performance. This cannot be done with rootless containers, because most Linux distributions do not allow mounting overlay file systems in user namespaces.

For rootless containers, the solution is to create a new storage driver. The FUSE-OverlayFS is a user-space implementation of Overlay, which is more efficient than the VFS storage driver used before and can run inside user namespaces.

References

[user namespaces\(7\) man page](#)

[Use of Podman in a Rootless environment](#)

[Previous](#) [Next](#)

Guided Exercise: Exploring Root and Rootless Containers

In this exercise, you will understand the differences between running containers as root and running them rootless.

Outcomes

You should be able to see the UIDs for processes running inside containers.

As the student user on the workstation machine, use the `lab` command to prepare your system for this exercise.

This command ensures that you have available the tools needed to perform this exercise.

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

Procedure 2.2. Instructions

1. Run a container as the root user and check the UID of a running process inside it.

1. Start a container with sudo from the Red Hat UBI 8 image.

2. [student@workstation ~]\$ sudo podman run --rm --name asroot -it \
3. > registry.access.redhat.com/ubi8:latest /bin/bash
4. Trying to pull registry.access.redhat.com/ubi8:latest...
5. Getting image source signatures
6. ...output omitted...
7. [root@f95d16108991 /]# whoami
8. root
9. [root@f95d16108991 /]# id

```
uid=0(root) gid=0(root) groups=0(root)
```

10. Start a sleep process inside the container.

```
[root@f95d16108991 /]# sleep 1000
```

11. On a new terminal, run ps to search for the process.

```
12. [student@workstation ~]$ sudo ps -ef | grep "sleep 1000"
```

```
root      3137      3117  0 10:18 pts/0    00:00:00 /usr/bin/coreutils --c  
oreutils-prog-shebang=sleep /usr/bin/sleep 1000
```

13. Exit the container.

```
14. [root@f95d16108991 /]# sleep 1000
```

```
15. ^C
```

```
16. [root@f95d16108991 /]# exit
```

```
17. exit
```

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

2. Run a container as a regular user and check the UID of a running process inside it.

1. Start another container from the Red Hat UBI 8 as a regular user.

```
2. [student@workstation ~]$ podman run --rm --name asuser -it \
3. > registry.access.redhat.com/ubi8:latest /bin/bash
4. Trying to pull registry.access.redhat.com/ubi8:latest...
5. Getting image source signatures
6. ...output omitted...
7. [root@d289dccd5285 /]# whoami
8. root
9. [root@d289dccd5285 /]# id
```

```
uid=0(root) gid=0(root) groups=0(root)
```

10. Start a sleep process inside the container.

```
[root@d289dccd5285 /]# sleep 2000
```

11. On a new terminal, run ps to search for the process.

```
12. [student@workstation ~]$ sudo ps -ef | grep "sleep 2000" | grep -v grep
```

```
student      3345      3325  0 10:24 pts/0    00:00:00 /usr/bin/coreutils --coreutils-prog-shebang=sleep /usr/bin/sleep 2000
```

13. Exit the container.

```
14. [root@d289dccd5285 /]# sleep 2000
```

```
15. ^C
```

```
16. [root@d289dccd5285 /]# exit
```

```
17. exit
```

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

Finish

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 container-rootless finish
```

This concludes the section.

[Previous](#) [Next](#)

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.

[Previous](#) [Next](#)

Chapter 3. Managing Containers

[Managing the Lifecycle of Containers](#)

[Guided Exercise: Managing a MySQL Container](#)

[Attaching Persistent Storage to Containers](#)

[Guided Exercise: Create MySQL Container with Persistent Database](#)

[Accessing Containers](#)

[Guided Exercise: Loading the Database](#)

[Lab: Managing Containers](#)

[Summary](#)

[Abstract](#)

Goal	Make use of prebuilt container images to create and manage containerized services.
Objectives	<ul style="list-style-type: none"> • Manage a container's lifecycle from creation to deletion. • Save container application data with persistent storage. • Describe how to use port forwarding to access a container.
Sections	<ul style="list-style-type: none"> • Managing the Lifecycle of Containers (and Guided Exercise) • Attaching Persistent Storage to Containers (and Guided Exercise) • Accessing Containers (and Guided Exercise)
Lab	<ul style="list-style-type: none"> • Managing Containers

Managing the Lifecycle of Containers

Objectives

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

Container Lifecycle 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 lifecycle. 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 lifecycle. 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
47c9aad6049        registry.redhat.io/rhel8/httpd-24   "/usr/bin/run-http..."   ... focused_fermat
```

The container ID is unique and generated automatically.

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-httdp-container \
> registry.redhat.io/rhel8/httdp-24
...output omitted... AH00094: Command line: 'httdp -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-httdp-container \
> -d registry.redhat.io/rhel8/httdp-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/httdp-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 `httdp` image. Hence, the `httdp` 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 `-1` (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 lifecycle. This lifecycle 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-httpd...  ...ago  
Up...  my-htt...
```

Each row describes information about the container.

CONTAINER ID

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

IMAGE

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

COMMAND

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

CREATED

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

STATUS

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

PORTS

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

NAMES

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

The default action for each signal is specified as follows:

Default Action	Description
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.

podman rm

This 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](#)

[Next](#)

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.

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
```

Procedure 3.1. 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. 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 D, Creating a Red Hat Account](#).

```
2. [student@workstation ~]$ podman login registry.redhat.io  
3. Username: your-username  
4. Password: your-password
```

Login Succeeded!

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

```
6. [student@workstation ~]$ podman run --name mysql-db \  
7. > registry.redhat.io/rhel8/mysql-80:1  
8. Trying to pull ...output omitted...  
9. ...output omitted...  
10. Writing manifest to image destination  
11. Storing signatures  
12. You must either specify the following environment variables:  
13. MYSQL_USER      (regex: ...)  
14. MYSQL_PASSWORD (regex: ...)  
15. MYSQL_DATABASE (regex: ...)  
16. Or the following environment variable:  
17. MYSQL_ROOT_PASSWORD (regex: ...)  
18. Or both.  
19. Optional Settings:
```

...output omitted...

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.

```
a8a6090a0a632b5876001725b581fdd331c9ab8b9eda21cc2a2899c23f078509
```

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

4. [student@workstation ~]\$ podman ps
5. CONTAINER ID ... STATUS ... NAMES

```
a8a6090a0a63 ... Up 13 seconds ago ... mysql
```

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

6. Populate the `items` database with the `Projects` table:

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

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

2. Populate the `items` database with the `Projects` table.

3. [student@workstation ~]\$ podman exec mysql /bin/bash -c \
4. > 'mysql -uuser1 -pmypa55 items < /db.sql'

```
mysql: [Warning] Using a password on the command line interface can be insecure.
```

7. 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.
8. [student@workstation ~]\$ podman run --name mysql-2 -it \
9. > registry.redhat.io/rhel8/mysql-80:1 /bin/bash

```
bash-4.4$
```

10. 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.

11. Exit the container.

```
bash-4.4$ exit
```

12. Verify that the mysql-2 container is not running.

```
13. [student@workstation ~]$ podman ps -a  
14. CONTAINER ID  STATUS          NAMES  
15. 27b4b00b7f5c  Exited (1) 4 minutes ago  mysql-db  
16. a8a6090a0a63  Up 4 minutes ago  mysql
```

```
bd517765c217  ... Exited (0) 8 seconds ago  mysql-2
```

17. 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.

```
18. [student@workstation ~]$ podman exec mysql /bin/bash -c \  
19. > 'mysql -uuser1 -pmypa55 -e "select * from items.Projects;"'
```

```
20.mysql: [Warning] Using a password on the command-line interface can be insecure.
```

id	name	code
----	------	------

1	DevOps	D0180
---	--------	-------

Finish

On workstation, run the following script to complete this exercise.

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

This concludes the exercise.

[Previous](#) [Next](#)

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.

4. [user@host ~]\$ sudo semanage fcontext -a -t container_file_t \

```
'/home/student/dbfiles(/.*)?'
```

5. 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.

References

[Dealing with user namespaces and SELinux on rootless containers](#)

[Understanding SELinux labels for container runtimes](#)

[Previous](#) [Next](#)

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.

The workstation should not have any container images running.

Run the following command on workstation:

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

Procedure 3.2. Instructions

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

1. Create the /home/student/local/mysql directory.

```
2. [student@workstation ~]$ mkdir -vp /home/student/local/mysql
```

```
3. mkdir: created directory /home/student/local
```

```
mkdir: created directory /home/student/local/mysql
```

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

5. [student@workstation ~]\$ sudo semanage fcontext -a \

```
> -t container_file_t '/home/student/local/mysql(/.*)?'
```

6. Apply the SELinux policy to the newly created directory.

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

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

8. [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
```

9. 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.
 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 D, Creating a Red Hat Account](#).

```
2. [student@workstation ~]$ podman login registry.redhat.io
3. Username: your-username
4. Password: your-password
```

Login Succeeded!

5. Pull the MySQL container image.

```
6. [student@workstation ~]$ podman pull registry.redhat.io/rhel8/mysql-80:1
7. Trying to pull registry.redhat.io/rhel8/mysql-80:1...
8. Getting image source signatures
9. Checking if image destination supports signatures
10. Copying blob 028bdc977650 done
11....output omitted...
12. Writing manifest to image destination
13. Storing signatures
```

4ae3a3f4f409a8912cab9fbf71d3564d011ed2e68f926d50f88f2a3a72c809c5

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

```
15. [student@workstation ~]$ podman run --name persist-db \
16. > -d -v /home/student/local/mysql:/var/lib/mysql/data \
17. > -e MYSQL_USER=user1 -e MYSQL_PASSWORD=mypa55 \
18. > -e MYSQL_DATABASE=items -e MYSQL_ROOT_PASSWORD=r00tpa55 \
19. > 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.

20. Verify that the container started correctly.

```
21. [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.

```
4. [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.

[Previous](#) [Next](#)

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 8080:8080 \
> registry.redhat.io/rhel8/httpd-24
```

This example creates an externally accessible container. The value 8080 colon 8080 specifies that any requests to port 8080 on the host are forwarded to port 8080 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 127.0.0.1:8081:8080 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 8080 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::8080 \
> 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
8080/tcp -> 127.0.0.1:35134

[user@host ~]$ curl -s 127.0.0.1:35134 | egrep '</?title>'<br/>
<title>Test Page for the HTTP Server on Red Hat Enterprise Linux</title>
```

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 8080 registry.redhat.io/rhel8/httpd-24

[user@host ~]$ podman port apache4
8080/tcp -> 0.0.0.0:37068
```

In this example, any routable request to host port 37068 is forwarded to port 8080 in the container.

References

[Container Network Interface - networking for Linux containers](#)

[Cloud Native Computing Foundation](#)

[Previous](#) [Next](#)

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.

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.

Procedure 3.3. Instructions

1. Create a MySQL container instance with persistent storage and port forwarding.
 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 D, Creating a Red Hat Account](#).
 2. [student@workstation ~]\$ podman login registry.redhat.io
 3. Username: `your-username`
 4. Password: `your-password`

Login Succeeded!

5. Download the MySQL database container image and then create a MySQL container instance with persistent storage and port forwarding.
 6. [student@workstation ~]\$ podman run --name mysqladb-port \
 7. > -d -v /home/student/local/mysql:/var/lib/mysql/data -p 13306:3306 \
 8. > -e MYSQL_USER=user1 -e MYSQL_PASSWORD=mypa55 \
 9. > -e MYSQL_DATABASE=items -e MYSQL_ROOT_PASSWORD=r00tpa55 \
 10. > registry.redhat.io/rhel8/mysql-80:1

```
11. Trying to pull registry.redhat.io/rhel8/mysql-80:1...
12. Getting image source signatures
13. Checking if image destination supports signatures
14. Copying blob 0c673eb68f88 done
15....output omitted...
16. Writing manifest to image destination
17. 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.

- Verify that the mysql-80 container started successfully and enables port forwarding.

```
3. [student@workstation ~]$ podman ps --format="{{.ID}} {{.Names}} {{.Ports}}"
```

```
9941da2936a5    mysql-80      0.0.0.0:13306->3306/tcp
```

- 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.

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

```
8. [student@workstation ~]$ podman exec -it mysql ldb-port \
9. > mysql -uroot items -e "SELECT * FROM Item"
10. +-----+-----+
11. | id | description | done |
12. +-----+-----+
13. | 1 | Pick up newspaper | 0 |
14. | 2 | Buy groceries | 1 |
```

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

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

```
16. [student@workstation ~]$ mysql -uuser1 -h 127.0.0.1 -pmypa55 -P13306 \
17. > items -e "SELECT * FROM Item"
18. +-----+-----+
19. | id | description | done |
20. +-----+-----+
21. | 1 | Pick up newspaper | 0 |
22. | 2 | Buy groceries | 1 |
```

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

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

1. Open a Bash shell inside the container.

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

```
bash-4.4$
```

3. Verify that the database contains data:

```
4. bash-4.4$ mysql -uroot items -e "SELECT * FROM Item"
5. +-----+-----+
6. | id | description | done |
7. +-----+-----+
```

8.		1		Pick up newspaper		0	
9.		2		Buy groceries		1	

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

10. Exit from the container:

```
11. bash-4.4$ exit
```

```
12. exit
```

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

Finish

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

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

This concludes the exercise.

[Previous](#) [Next](#)

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.

[Previous](#) [Next](#)

Chapter 4. Managing Container Images

[Accessing Registries](#)

[Quiz: Working with Registries](#)

[Manipulating Container Images](#)

[Guided Exercise: Creating a Custom Apache Container Image](#)

[Lab: Managing Images](#)

[Summary](#)

Abstract

Goal	Manage the lifecycle of a container image from creation to deletion.
Objectives	<ul style="list-style-type: none"> • Search for and pull images from remote registries. • Export, import, and manage container images locally and in a registry.
Sections	<ul style="list-style-type: none"> • Accessing Registries (and Quiz) • Manipulating Container Images (and Guided Exercise)
Lab	<ul style="list-style-type: none"> • 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

Note

The following content uses the version 1 format for registry configuration. The version 1 format is deprecated in recent versions of Podman and does not support using registry mirrors, longest-prefix matches, or location rewriting.

Refer to the `containers-registries.conf(5)` man page for further details on the supported registry configuration format and capabilities.

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 <number></code>	Limits the number of listed images per registry.
<code>--filter <filter=value></code>	Filter output based on conditions provided. Supported filters are: <code>stars=<number></code> : Show only images with at least this number of stars. <code>is-automated=<true false></code> : Show only images automatically built. <code>is-official=<true false></code> : Show only images flagged as official.
<code>--tls-verify <true false></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...o5G8",  
"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 searches for images in registries in the same order they appear in the configuration file.

Red Hat recommends always including the registry name to prevent accidental use of unexpected images.

Malicious actors can upload images to registries. If these registries appear first in your configuration, then you use the image uploaded by a malicious actor instead of the image you intend to use.

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](#)

[Quay.io](#)

[Docker Registry HTTP API V2](#)

[Next](#)

Quiz: Working with Registries

Note

The following quiz uses the version 1 format for registry configuration. The version 1 format is deprecated in recent versions of Podman and does not support using registry mirrors, longest-prefix matches, or location rewriting.

Refer to the `containers-registries.conf(5)` man page for further details on the supported registry configuration format and capabilities.

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:

Table 4.1. Image names/tags per registry

Registry	Image
registry.redhat.io	<ul style="list-style-type: none">nginx:1.1.6mysql:8.0httpd:2.4
quay.io	<ul style="list-style-type: none">mysql:5.7httpd:2.4

No images are locally available.

1.

2.

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`

3. CheckResetShow Solution

4.

5.

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.

6. CheckResetShow Solution

7.

8.

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

9. CheckResetShow Solution

10.

11.

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

```
[user@host ~]$ podman pull registry.redhat.io/httpd:2.4  
[user@host ~]$ podman
```

```
pull quay.i  
o/mysql:8.0
```

- A quay.io/httpd:2.4, and
`registry.redhat.io/mysql:8.0
- B registry.redhat.io/httpd:2.4,
and 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.

12.CheckResetShow Solution

[Previous](#) [Next](#)

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 `Containerfile`s.

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.

Option	Description
--format	Selects the format of the image. Valid options are oci and docker.

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/mysqld
A /run/mysqld/mysqld.pid
A /run/mysqld/mysqld.sock
A /run/mysqld/mysqld.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](#)

[Previous](#) [Next](#)

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.

Make sure you have completed [the section called “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
```

Procedure 4.1. 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.

2. [student@workstation ~]\$ podman login quay.io
3. Username: *your_quay_username*
4. Password: *your_quay_password*
5. Login Succeeded!
- 6.
7. [student@workstation ~]\$ podman run -d --name official-httdp \
8. > -p 8180:80 quay.io/redhattraining/httpd-parent
9. Trying to pull quay.io/redhattraining/httpd-parent:latest...
10. Getting image source signatures
11. Copying blob a3ed95caeb02 done
- 12....*output omitted*...
13. Writing manifest to image destination
14. Storing signatures

```
3a6baecaff2b4e8c53b026e04847dda5976b773ade1a3a712b1431d60ac5915d
```

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

15. Create an HTML page on the official-httdp container.

1. Access the shell of the container by using the podman exec command and create an HTML page.

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

```
bash-4.4# echo "DO180 Page" > /var/www/html/do180.html
```

3. Exit from the container.

```
4. bash-4.4# exit
```

```
exit
```

5. 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
```

The expected output is similar to the following:

```
D0180 Page
```

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

```
17. [student@workstation ~]$ podman diff official-httdp
```

```
18. C /etc
```

```
19. C /root
```

```
20. A /root/.bash_history
```

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

```
22. C /tmp
```

```
23. C /var
```

```
24. C /var/log
```

```
25. C /var/log/httdp
```

```
26. A /var/log/httdp/access_log
```

```
27. A /var/log/httdp/error_log
```

```
28. C /var/www
```

```
29. 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.

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

1. Stop the official-`httpd` container.

```
2. [student@workstation ~]$ podman stop official-httpd
```

```
official-httpd
```

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

```
4. [student@workstation ~]$ podman commit -a 'Your Name' \
5. > official-httpd do180-custom-httpd
6. Getting image source signatures
7. ...output omitted...
8. Copying blob 7f9108fde4a1 skipped: already exists
9. Copying blob 4c0f28c7efd0 done
10. Copying config 5c2bb39b76 done
11. Writing manifest to image destination
12. Storing signatures
```

```
31c3ac78e9d4137c928da23762e7d32b00c428eb1036cab1caeeb399bef2a23
```

13. List the available container images.

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

The expected output is similar to the following:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ZE				
localhost/do180-custom- <code>httpd</code>	latest	31c3ac78e9d4
.				
quay.io/redhattraining/httpd-parent	latest	2cc07fbb5000
.				

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

31. Publish the saved container image to the container registry.

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

```
3. [student@workstation ~]$ podman tag do180-custom-httd \
```

```
> quay.io/${RHT_OCP4_QUAY_USER}/do180-custom-httd:v1.0
```

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

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

REPOSITORY	TAG	IMAGE ID
localhost/do180-custom-httd	latest	31c3ac78e9d4
quay.io/\${RHT_OCP4_QUAY_USER}/do180-custom-httd	v1.0	31c3ac78e9d4

quay.io/redhattraining/httpd-parent	latest	2cc07fbb5000
...		

9. Publish the image to your Quay.io registry.

```
10. [student@workstation ~]$ podman push \
```

```
11. > quay.io/${RHT_OCP4_QUAY_USER}/do180-custom-httd:v1.0
```

```
12. Getting image source signatures
```

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

```
14. Copying blob 24d85c895b6b skipped: already exists
```

```
15. Copying config 5c2bb39b76 done
```

```
16. Writing manifest to image destination
```

```
Storing signatures
```

Note

Pushing the `do180-custom-httdp` 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.

17. 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.

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

```
31c3ac78e9d4137c928da23762e7d32b00c428eb1036cab1caeeb3
```

32. Create a container from the newly published image.

Use the `podman run` command to start a new container.

Use `your_quay_username/do180-custom-httdp:v1.0` as the base image.

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

33. 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.

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

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

Procedure 4.2. 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.

Show Solution

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

Property	Description
Name	<code>official-nginx</code>
Run as daemon	<code>yes</code>
Container image	<code>quay.io/redhattraining/nginx:1.17</code>
Port forward	<code>yes</code> , from host port 8080 to container port 80.

3. Show Solution
4. Log in to the container using the `podman exec` command. Replace the contents of the `index.html` file with `DO180`. 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.

Show Solution

5. 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:

Property	Description
Image name	<code>do180/mynginx</code>
Image tag	<code>v1.0-SNAPSHOT</code>
Author name	<i>your name</i>

6. Show Solution
7. Start a new container using the updated Nginx image, according to the specifications listed in the following list.

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

8. Show Solution
9. 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.

Show Solution

10. 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:

Property	Description
Image name	do180/mynginx
Image tag	v1.0
Author name	<i>your name</i>

11. Show Solution

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

Show Solution

13. Use the image tagged `do180/mynginx:v1.0` to create a new container with the following specifications:

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

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

15. Show Solution

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.

[Previous](#) [Next](#)

Summary

In this chapter, you learned:

- The Red Hat Container Catalog provides tested and certified images at `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.

[Previous](#) [Next](#)

Chapter 5. Creating Custom Container Images

[**Designing Custom Container Images**](#)

[**Quiz: Approaches to Container Image Design**](#)

[**Building Custom Container Images with Containerfiles**](#)

[**Guided Exercise: Creating a Basic Apache Container Image**](#)

[**Lab: Creating Custom Container Images**](#)

[**Summary**](#)

[**Abstract**](#)

Goal	Design and code a Containerfile to build a custom container image.
Objectives	<ul style="list-style-type: none">• Describe the approaches for creating custom container images.• Create a container image using common Containerfile commands.
Sections	<ul style="list-style-type: none">• 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](#). 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 `rhsc1-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.

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 \(RHSCl\)](#)

[Red Hat Container Catalog \(RHCC\)](#)

[RHSCl Dockerfiles on GitHub](#)

[Using Red Hat Software Collections Container Images](#)

[Quay.io](#)

[Docker Hub](#)

[Docker Library GitHub project](#)

[The S2I GitHub project](#)

[Next](#)

Quiz: Approaches to Container Image Design

Choose the correct answers to the following questions:

1.

2.

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.

3. CheckResetShow Solution

4.

5.

2. Which of the following options is an advantage of using the stand-alone S2I process as an alternative to Containerfiles?

- 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.

6. CheckResetShow Solution

7.

8.

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.

9. CheckResetShow Solution

[Previous](#) [Next](#)

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

FROM      ubi8/ubi:8.5

LABEL     description="This is a custom httpd container image"

MAINTAINER John Doe <jdoe@xyz.com>

RUN      yum install -y httpd

EXPOSE   80

ENV      LogLevel "info"

ADD      http://someserver.com/filename.pdf /var/www/html

COPY     ./src/    /var/www/html/

USER     apache

ENTRYPOINT [ "/usr/sbin/httpd" ]

CMD      [ "-D", "FOREGROUND" ]
```

Lines that begin with a hash, or pound, sign (#) are comments.

The `FROM` instruction declares that the new container image extends `ubi8/ubi:8.5` 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.

The `LABEL` is responsible for adding generic metadata to an image. A `LABEL` is a simple key-value pair.

`MAINTAINER` indicates the `Author` field of the generated container image's metadata. You can use the `podman inspect` command to view image metadata.

`RUN` executes commands in a new layer on top of the current image. The shell that is used to execute commands is `/bin/sh`.

`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.

`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.

`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.

`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.

`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.

`ENTRYPOINT` specifies the default command to execute when the image runs in a container. If omitted, the default `ENTRYPOINT` is `/bin/sh -c`.

`CMD` provides the default arguments for the `ENTRYPOINT` instruction. If the default `ENTRYPOINT` applies (`/bin/sh -c`), then `CMD` forms an executable command and parameters that run at container start.

CMD and ENTRYPOINT

`ENTRYPOINT` and `CMD` instructions have two formats:

Exec form

This form uses a JSON array:

```
ENTRYPOINT ["command", "param1", "param2"]
CMD      ["param1", "param2"]
```

Shell form

```
ENTRYPOINT 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 `ENTRYPOINT` 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 `ENTRYPOINT` 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 `ENTRYPOINT`. In this case, the base image's `ENTRYPOINT` applies, or the default `ENTRYPOINT` 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.

```
ENTRYPOINT ["/bin/date", "+%H:%M"]
```

The `ENTRYPOINT` 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.

```
ENTRYPOINT ["/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 `ENTRYPOINT` 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:

Shell form

```
ADD source ... destination
```

```
COPY source ... destination
```

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 local tar archive in a recognized compression format (identity, gzip, bzip2 or xz), then the `ADD` instruction unpacks the archive as a directory to the `destination` folder. The `COPY` instruction does not have this functionality.

Important

Both the `ADD` and `COPY` instructions copy normal files, with `root` as the owner. The `ADD` instruction retrieves the contents of the URL and the file is copied with `root` as the owner. The `ADD` instruction extracts the tar archive and preserve the owner and permissions.

You can use the `--chown=<user>:<group>` option with the `COPY` or `ADD` command to set the desired owner or use a `RUN` instruction after the copy to set the owner and permissions.

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
```

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 ins  
tall -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:

```
[user@host ~]$ 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](#).

References

[Dockerfile Reference Guide](#)

[Creating base images](#)

[Previous](#) [Next](#)

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.

Run the following command to download the relevant lab files and to verify that Docker is running:

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

Procedure 5.1. Instructions

1. Create the Apache Containerfile.
 1. Open a terminal on workstation. Using your preferred editor, create a new Containerfile.

```
[student@workstation ~]$ vim ~/D0180/labs/dockerfile-create/Containerfile
```

2. Use UBI 8.5 as a base image by adding the following FROM instruction at the top of the new Containerfile:

```
FROM ubi8/ubi:8.5
```

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>
```

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"
```

5. Add a RUN instruction with a yum install command to install Apache on the new container.

```
6. RUN yum install -y httpd && \
```

```
yum clean all
```

7. Add a RUN instruction to replace contents of the default HTTPD home page.

```
RUN echo "Hello from Containerfile" > /var/www/html/index.html
```

8. 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.

9. At the end of the file, use the following CMD instruction to set httpd as the default entry point:

```
CMD ["httpd", "-D", "FOREGROUND"]
```

10. Verify that your Containerfile matches the following before saving and proceeding with the next steps:

```
11. FROM ubi8/ubi:8.5
12.
13. MAINTAINER Your Name <youremail>
14.
15. LABEL description="A custom Apache container based on UBI 8"
16.
17. RUN yum install -y httpd && \
18.     yum clean all
19.
20. RUN echo "Hello from Containerfile" > /var/www/html/index.html
21.
22. EXPOSE 80
23.
```

```
CMD ["httpd", "-D", "FOREGROUND"]
```

2. Build and verify the Apache container image.
 1. Use the following commands to create a basic Apache container image using the newly created Containerfile:
 2. [student@workstation ~]\$ cd ~/D0180/labs/dockerfile-create
 - 3.
 4. [student@workstation dockerfile-create]\$ podman build --layers=false \
5. > -t do180/apache .
 6. STEP 1/7: FROM ubi8/ubi:8.5
 7. ...output omitted...
 8. Trying to pull registry.access.redhat.com/ubi8/ubi:8.5...
 9. ...output omitted...
 10. STEP 2/7: MAINTAINER Your Name <youremail>
 11. STEP 3/7: LABEL description="A custom Apache container based on UBI 8"
 12. STEP 4/7: RUN yum install -y httpd && yum clean all
 13. Updating Subscription Management repositories.
 14. Unable to read consumer identity
 15. Subscription Manager is operating in container mode.
 - 16....output omitted...
 17. STEP 5/7: RUN echo "Hello from Containerfile" > /var/www/html/index.html
 18. STEP 6/7: EXPOSE 80
 19. STEP 7/7: CMD ["httpd", "-D", "FOREGROUND"]
 20. COMMIT do180/apache
 21. Getting image source signatures
 - 22....output omitted...
 23. Writing manifest to image destination
 24. Storing signatures
 25. --> 16c8493a19a
 26. Successfully tagged localhost/do180/apache:latest

```
16c8493a19ad5162031cf17c6b2e338171f4619b6358cb367608c80791aa7718
```

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.

27. After the build process has finished, run `podman images` to see the new image in the image repository.

```
28. [student@workstation dockerfile-create]$ podman images
29. REPOSITORY           TAG      IMAGE ID      CREATED
   SIZE
30. localhost/do180/apache    latest   16c8493a19ad  45 seconds ago
   257 MB
```

```
registry.access.redhat.com/ubi8/ubi  8.5      202c1768b1f7  2 months ago
235 MB
```

3. Run the Apache container.

1. Use the following command to run a container using the Apache image:
2. [student@workstation dockerfile-create]\$ podman run --name lab-apache \
- 3. > -d -p 10080:80 do180/apache

```
fa1d1c450e8892ae085dd8bbf763edac92c41e6ffaa7ad6ec6388466809bb391
```

4. Run the `podman ps` command to see the running container.

```
5. [student@workstation dockerfile-create]$ podman ps
6. CONTAINER ID  IMAGE           COMMAND      CREATED
7. STATUS        PORTS          NAMES
8. fa1d1c450e88  localhost/do180/apache:latest  httpd -D ...  10 seconds ago
```

```
Up 9 seconds ago  0.0.0.0:10080->80/tcp  lab-apache
```

9. Use the `curl` command to verify that the server is running.

```
10. [student@workstation dockerfile-create]$ curl -s 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.

[Previous](#) [Next](#)

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.

Run the following command to download the relevant lab files and to verify that Docker is running:

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

Procedure 5.1. Instructions

1. Create the Apache Containerfile.
 1. Open a terminal on `workstation`. Using your preferred editor, create a new Containerfile.

```
[student@workstation ~]$ vim ~/D0180/labs/dockerfile-create/Containerfile
```

2. Use UBI 8.5 as a base image by adding the following `FROM` instruction at the top of the new Containerfile:

```
FROM ubi8/ubi:8.5
```

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>
```

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"
```

5. Add a `RUN` instruction with a `yum install` command to install Apache on the new container.

```
6. RUN yum install -y httpd && \
```

```
yum clean all
```

7. Add a `RUN` instruction to replace contents of the default HTTPD home page.

```
RUN echo "Hello from Containerfile" > /var/www/html/index.html
```

8. 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.

9. At the end of the file, use the following `CMD` instruction to set `httpd` as the default entry point:

```
CMD ["httpd", "-D", "FOREGROUND"]
```

10. Verify that your Containerfile matches the following before saving and proceeding with the next steps:

```
11. FROM ubi8/ubi:8.5
12.
13. MAINTAINER Your Name <youremail>
14.
15. LABEL description="A custom Apache container based on UBI 8"
16.
17. RUN yum install -y httpd && \
18.     yum clean all
19.
20. RUN echo "Hello from Containerfile" > /var/www/html/index.html
21.
22. EXPOSE 80
23.
```

```
CMD ["httpd", "-D", "FOREGROUND"]
```

2. Build and verify the Apache container image.

1. Use the following commands to create a basic Apache container image using the newly created Containerfile:

```
2. [student@workstation ~]$ cd ~/D0180/labs/dockerfile-create
3.
4. [student@workstation dockerfile-create]$ podman build --layers=false \
5. > -t do180/apache .
6. STEP 1/7: FROM ubi8/ubi:8.5
7. ...output omitted...
8.
8. Trying to pull registry.access.redhat.com/ubi8/ubi:8.5...
9. ...output omitted...
10. STEP 2/7: MAINTAINER Your Name <youremail>
11. STEP 3/7: LABEL description="A custom Apache container based on UBI 8"
12. STEP 4/7: RUN yum install -y httpd &&     yum clean all
```

```
13. Updating Subscription Management repositories.  
14. Unable to read consumer identity  
15. Subscription Manager is operating in container mode.  
16. ....output omitted...  
17. STEP 5/7: RUN echo "Hello from Containerfile" > /var/www/html/index.html  
18. STEP 6/7: EXPOSE 80  
19. STEP 7/7: CMD ["httpd", "-D", "FOREGROUND"]  
20. COMMIT do180/apache  
21. Getting image source signatures  
22. ....output omitted...  
23. Writing manifest to image destination  
24. Storing signatures  
25. --> 16c8493a19a  
26. Successfully tagged localhost/do180/apache:latest
```

```
16c8493a19ad5162031cf17c6b2e338171f4619b6358cb367608c80791aa7718
```

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.

27. After the build process has finished, run `podman images` to see the new image in the image repository.

```
28. [student@workstation dockerfile-create]$ podman images  
29. REPOSITORY          TAG      IMAGE ID      CREATED  
     SIZE  
30. localhost/do180/apache    latest   16c8493a19ad  45 seconds ago  
                                257 MB
```

```
registry.access.redhat.com/ubi8/ubi  8.5      202c1768b1f7  2 months ago  
235 MB
```

3. Run the Apache container.

1. Use the following command to run a container using the Apache image:

```
2. [student@workstation dockerfile-create]$ podman run --name lab-apache \
3. > -d -p 10080:80 do180/apache
```

```
fa1d1c450e8892ae085dd8bbf763edac92c41e6ffaa7ad6ec6388466809bb391
```

4. Run the `podman ps` command to see the running container.

```
5. [student@workstation dockerfile-create]$ podman ps
6. CONTAINER ID IMAGE COMMAND CREATED
7. STATUS PORTS NAMES
8. fa1d1c450e88 localhost/do180/apache:latest httpd -D ... 10 seconds ago
```

```
Up 9 seconds ago 0.0.0.0:10080->80/tcp lab-apache
```

9. Use the `curl` command to verify that the server is running.

```
10. [student@workstation dockerfile-create]$ curl -s 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.

[Previous](#) [Next](#)

Chapter 6. Deploying Containerized Applications on OpenShift

[Describing Kubernetes and OpenShift Architecture](#)

[Quiz: Describing Kubernetes and OpenShift](#)

[Creating Kubernetes Resources](#)

[Guided Exercise: Deploying a Database Server on OpenShift](#)

[Creating Routes](#)

[Guided Exercise: Exposing a Service as a Route](#)

[Creating Applications with Source-to-Image](#)

[Guided Exercise: Creating a Containerized Application with Source-to-Image](#)

[Creating Applications with the OpenShift Web Console](#)

[Guided Exercise: Creating an Application with the Web Console](#)

[Lab: Deploying Containerized Applications on OpenShift](#)

[Summary](#)

Abstract

Goal	Deploy single container applications on OpenShift Container Platform.
Objectives	<ul style="list-style-type: none">• Describe the architecture of Kubernetes and Red Hat OpenShift Container Platform.• Create standard Kubernetes resources.• Create a route to a service.• Build an application using the Source-to-Image facility of OpenShift Container Platform.• Create an application using the OpenShift web console.
Sections	<ul style="list-style-type: none">• Describing Kubernetes and OpenShift Architecture (and Quiz)• Creating Kubernetes Resources (and Guided Exercise)• Creating Routes (and Guided Exercise)• Creating Applications with the Source-to-Image Facility (and Guided Exercise)• Creating Applications with the OpenShift Web Console (and Guided Exercise)
Lab	<ul style="list-style-type: none">• Deploying Containerized Applications on OpenShift

Describing Kubernetes and OpenShift Architecture

Objectives

After completing this section, students should be able to:

- Describe the architecture of a Kubernetes cluster running on the Red Hat OpenShift Container Platform (RHOCP).
- List the main resource types provided by Kubernetes and RHOCP.
- Identify the network characteristics of containers, Kubernetes, and RHOCP.
- List mechanisms to make a pod externally available.

Kubernetes and OpenShift

In previous chapters we saw that Kubernetes is an orchestration service that simplifies the deployment, management, and scaling of containerized applications. One of the main advantages of using Kubernetes is that it uses several nodes to ensure the resiliency and scalability of its managed applications. Kubernetes forms a cluster of node servers that run containers and are centrally managed by a set of control plane servers. A server can act as both a control plane node and a compute node, but those roles are usually separated for increased stability.

Table 6.1. Kubernetes Terminology

Term	Definition
Node	A server that hosts applications in a Kubernetes cluster.
Control Plane	Provides basic cluster services such as APIs or controllers.
Compute Node	This node executes workloads for the cluster. Application pods are scheduled onto compute nodes.
Resource	Resources are any kind of component definition managed by Kubernetes. Resources contain the configuration of the managed component (for example, the role assigned to a node), and the current state of the component (for example, if the node is available).
Controller	A controller is a Kubernetes process that watches resources and makes changes attempting to move the current state towards the desired state.
Label	A key-value pair that can be assigned to any Kubernetes resource. Selectors use labels to filter eligible resources for scheduling and other operations.
Namespace	A scope for Kubernetes resources and processes, so that resources with the same name can be used in different boundaries.

Note

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

Red Hat OpenShift Container Platform is a set of modular components and services built on top of Red Hat CoreOS and Kubernetes. RHOCOP adds PaaS capabilities such as remote management, increased security, monitoring and auditing, application lifecycle management, and self-service interfaces for developers.

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

Table 6.2. OpenShift Terminology

Term	Definition
Infra Node	A node server containing infrastructure services like monitoring, logging, or external routing.
Console	A web UI provided by the RHOCOP cluster that allows developers and administrators to interact with cluster resources.
Project	OpenShift extension of Kubernetes' namespaces. Allows the definition of user access control (UAC) to resources.

The following schema illustrates the OpenShift Container Platform stack:

Figure 6.1: OpenShift component stack

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

- The base OS is Red Hat CoreOS. Red Hat CoreOS is a Linux distribution focused on providing an immutable operating system for container execution.
- CRI-O is an implementation of the Kubernetes Container Runtime Interface (CRI) to enable using Open Container Initiative (OCI) compatible runtimes. CRI-O can use any container runtime that satisfies CRI: runc (used by the Docker service), libpod (used by Podman) or rkt (from CoreOS).
- Kubernetes manages a cluster of hosts, physical or virtual, that run containers. It uses resources that describe multicontainer applications composed of multiple resources, and how they interconnect.
- Etcd is a distributed key-value store, used by Kubernetes to store configuration and state information about the containers and other resources inside the Kubernetes cluster.
- Custom Resource Definitions (CRDs) are resource types stored in Etcd and managed by Kubernetes. These resource types form the state and configuration of all resources managed by OpenShift.
- Containerized services fulfill many PaaS infrastructure functions, such as networking and authorization. RHOCP uses the basic container infrastructure from Kubernetes and the underlying container runtime for most internal functions. That is, most RHOCP internal services run as containers orchestrated by Kubernetes.
- Runtimes and xPaaS are base container images ready for use by developers, each preconfigured with a particular runtime language or database. The xPaaS offering is a set of base images for Red Hat middleware products such as JBoss EAP and ActiveMQ. Red Hat OpenShift Application Runtimes (RHOAR) are a set runtimes optimized for cloud native applications in OpenShift. The application runtimes available are Red Hat JBoss EAP, OpenJDK, Thorntail, Eclipse Vert.x, Spring Boot, and Node.js.
- RHOCP provides web UI and CLI management tools for managing user applications and RHOCP services. The OpenShift web UI and CLI tools are built from REST APIs which can be used by external tools such as IDEs and CI platforms.

This OpenShift and Kubernetes architecture illustration gives further insight into how the infrastructure components work together.

Figure 6.2: OpenShift and Kubernetes architecture

New Features in RHOCP 4

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

- CoreOS as the mandatory operating system for all nodes, offering an immutable infrastructure optimized for containers.
- A brand new cluster installer which guides the process of installation and update.
- A self-managing platform, able to automatically apply cluster updates and recoveries without disruption.
- A redesigned application lifecycle management.
- An Operator SDK to build, test, and package Operators.

Describing Kubernetes Resource Types

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

Pods (pod)

Represent a collection of containers that share resources, such as IP addresses and persistent storage volumes. It is the basic unit of work for Kubernetes.

Services (svc)

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

Replication Controllers (rc)

A Kubernetes resource that defines how pods are replicated (horizontally scaled) into different nodes. Replication controllers are a basic Kubernetes service to provide high availability for pods and containers.

Persistent Volumes (pv)

Define storage areas to be used by Kubernetes pods.

Persistent Volume Claims (pvc)

Represent a request for storage by a pod. PVCs links a PV to a pod so its containers can make use of it, usually by mounting the storage into the container's file system.

ConfigMaps (cm) and Secrets

Contains a set of keys and values that can be used by other resources. ConfigMaps and Secrets are usually used to centralize configuration values used by several resources. Secrets differ from ConfigMaps maps in that Secrets' values are always encoded (not encrypted) and their access is restricted to fewer authorized users.

Note

Although Kubernetes pods can be created standalone, they are usually created by high-level resources such as replication controllers.

OpenShift Resource Types

The main resource types added by OpenShift Container Platform to Kubernetes are as follows:

Deployment and Deployment config (dc)

OpenShift 4.5 introduced the Deployment resource concept to replace the DeploymentConfig as the default configuration for pods. Both are the representation of a set of containers included in a pod, and the deployment strategies to be used. It contains the configuration to be applied to all containers of each pod replica, such as the base image, tags, storage definitions and the commands to be executed when the containers start.

The Deployment object serves as the improved version of the DeploymentConfig object. Some replacements of functionalities between both objects are the following:

- Automatic rollback is no longer supported by deployment objects.
- Every change in the pod template used by deployment objects triggers a new rollout automatically.
- Lifecycle hooks are no longer supported by deployment objects.
- The deployment process of a deployment object can be paused at any time without affecting the deployer process.

- A deployment object can have as many active replica sets as the user wants, scaling down old replicas after a while. In contrast, the deploymentconfig object can only have two replication sets active at the same time.

Even if Deployment objects are meant to act as the default replacement of DeploymentConfig objects, in OpenShift 4.10 users can still make use of them if they need a specific feature provided by these objects. In this case, it is necessary to specify the type of object when creating a new application by specifying the `--as-deployment-config` flag.

Build config (bc)

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

Routes

Represent a DNS host name recognized by the OpenShift router as an ingress point for applications and microservices.

Note

To obtain a list of all the resources available in a RHOC cluster and their abbreviations, use the `oc api-resources` or `kubectl api-resources` commands.

Although Kubernetes replication controllers can be created standalone in OpenShift, they are usually created by higher-level resources such as deployment controllers.

Networking

Each container deployed in a Kubernetes cluster has an IP address assigned from an internal network that is accessible only from the node running the container. Because of the container's ephemeral nature, IP addresses are constantly assigned and released.

Kubernetes provides a software-defined network (SDN) that spawns the internal container networks from multiple nodes and allows containers from any pod,

inside any host, to access pods from other hosts. Access to the SDN only works from inside the same Kubernetes cluster.

Containers inside Kubernetes pods should not connect to each other's dynamic IP address directly. Services resolves this problem by linking more stable IP addresses from the SDN to the pods. If pods are restarted, replicated, or rescheduled to different nodes, services are updated, providing scalability and fault tolerance.

External access to containers is more complicated. Kubernetes services can be set as a NodePort service type. With this type of service, Kubernetes allocates a network port from a predefined range in each of the cluster nodes and proxies it into your service. Unfortunately, this approach does not scale well.

OpenShift makes external access to containers both scalable and simpler by defining route resources. A route defines external-facing DNS names and ports for a service. A router (ingress controller) forwards HTTP and TLS requests to the service addresses inside the Kubernetes SDN. The only requirement is that the desired DNS names are mapped to the IP addresses of the RHOCP router nodes.

Quiz: Describing Kubernetes and OpenShift

Choose the correct answers to the following questions:

1.

2.

- 1.** Which two sentences are correct regarding Kubernetes architecture?
(Choose two.)

A Kubernetes nodes can be managed without a control plane.

B Kubernetes control planes manage pod scaling.

- C Kubernetes control planes schedule pods to specific nodes.
- D Kubernetes tools cannot be used to manage resources in an OpenShift cluster.
- E Containers created from Kubernetes pods cannot be managed using standalone tools such as Podman.

3. CheckResetShow Solution

4.

5.

2. Which two sentences are correct regarding Kubernetes and OpenShift resource types? (Choose two.)

- A A pod is responsible for provisioning its own persistent storage.
- B All pods generated from the same replication controller have to run in the same node.
- C A persistent volume define storage areas available to pods through a persistent volume claim.
- D A replication controller is responsible for monitoring and maintaining the number of pods for a particular application.

6. CheckResetShow Solution

7.

8.

3. Which two statements are true regarding Kubernetes and OpenShift networking?
(Choose two.)

A

A Kubernetes service can provide an IP address to access a set of pods.

B

Kubernetes is responsible for providing a fully qualified domain name for a pod.

C

A replication controller is responsible for routing external requests to the pods.

D

A route is responsible for providing DNS names for external access.

9. CheckResetShow Solution

10.

11.

4. Which statement is correct regarding persistent storage in OpenShift and Kubernetes?

A

A PVC represents a storage area that a pod can use to store data and is provisioned by the application developer.

- B A PVC represents a storage area that can be requested by a pod to store data but is provisioned by the cluster administrator.
- C A PVC represents the amount of memory that can be allocated to a node, so that a developer can state how much memory he requires for his application to run.
- D A PVC represents the number of CPU processing units that can be allocated to an application pod, subject to a limit managed by the cluster administrator.

12.CheckResetShow Solution

13.

14.

5. Which statement is correct regarding OpenShift additions to Kubernetes?

- A OpenShift adds features to simplify Kubernetes configuration of many real-world use cases.
- B Container images created for OpenShift cannot be used with plain Kubernetes.
- C Red Hat maintains forked versions of Kubernetes internal to the RHOCUP product.
- D Doing continuous integration and continuous deployment with RHOCUP requires external tools.

15.CheckResetShow Solution

[Previous](#) [Next](#)

Creating Kubernetes Resources

Objectives

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

The Red Hat OpenShift Container Platform (RHOCP) Command-line Tool

The main method of interacting with an RHOCP cluster is using the `oc` command. The basic usage of the command is through its subcommands in the following syntax:

```
[user@host ~]$ oc <command>
```

Before interacting with a cluster, most operations require the user to log in. The syntax to log in is shown below:

```
[user@host ~]$ oc login <cluster-url>
```

Describing Pod Resource Definition Syntax

RHOCP 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: Pod
metadata:
  name: wildfly
  labels:
    name: wildfly
```

```

spec:
  containers:
    - resources:
        limits:
          cpu: 0.5

      image: do276/todojee
      name: wildfly
      ports:
        - containerPort: 8080
          name: wildfly

    env:
      - name: MYSQL_ENV_MYSQL_DATABASE
        value: items
      - name: MYSQL_ENV_MYSQL_USER
        value: user1
      - name: MYSQL_ENV_MYSQL_PASSWORD
        value: mypa55

```

Declares a Kubernetes pod resource type.

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

Creates a label with a key named `name` that other resources in Kubernetes, usually as service, can use to find it.

Defines the container image name.

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

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 6.3: 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 6.4: 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",  
  "apiVersion": "v1",  
  "metadata": {  
  
    "name": "quotedb"  
  },  
  "spec": {  
  
    "ports": [  
      {  
        "port": 3306,  
        "targetPort": 3306  
      }  
    ],  
    "selector": {  
  
      "name": "mysqlDb"  
    }  
  }  
}
```

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

A unique name for the service.

`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`.

`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 6.5: Alternative method for external access to a Kubernetes service](#) illustrates how NodePort services allow external access to Kubernetes services. OpenShift routes are covered in more detail later in this course.

Figure 6.5: 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-g1qrp 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 6.6: Resources created for a new application

The following command creates an application based on an image from a private Docker image registry:

```
[user@host ~]$ oc new-app --image=myregistry.com/mycompany/myapp --name=myapp
```

The following command creates an application based on source code stored in a Git repository:

```
[user@host ~]$ 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:

```
[user@host ~]$ 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	...
<i>...output omitted...</i>					

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"
  volumes:
    - name: "pvol"
  persistentVolumeClaim:
    claimName: "myapp"
```

This section declares that the `pvol` volume mounts at `/var/www/html` in the container file system.

This section defines the `pvol` volume.

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-1-144m7	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	
NAME	REVISION	DESIRED	CURRENT	
dc/nginx	1	1	1	
NAME	DESIRED	CURRENT	READY	
rc/nginx-1	1	1	1	
NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
svc/nginx	172.30.72.75	<none>	80/TCP,443/TCP	1h
NAME	READY	STATUS	RESTARTS	AGE

```
po/nginx-1-ypp8t  1/1      Running    0          1h
```

oc describe

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:

```
[user@host ~]$ 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](#)

Additional information about creating images is available in the OpenShift Container Platform documentation: [Creating Images](#)

Labels and label selectors details are available in *Working with Kubernetes Objects* section for the Kubernetes documentation: [Labels and Selectors](#)

[Previous](#) [Next](#)

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.

Make sure you have completed [the section called “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
```

Procedure 6.1. Instructions

1. Prepare the lab environment.
 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. Log in to the OpenShift cluster.

```
3. [student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
4. > -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
5. Login successful
```

...output omitted...

6. Create a new project that contains your RHOCUP developer username for the resources you create during this exercise:

```
7. [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 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.

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:

```
2. [student@workstation ~]$ oc status
3. In project ${RHT_OCP4_DEV_USER}-mysql-openshift on server ...
4.
5. svc/mysql - 172.30.151.91:3306
6. ...output omitted...
```

```
deployment #1 deployed 6 minutes ago - 1 pod
```

7. List the pods in this project to verify that the MySQL pod is ready and running:

```
8. [student@workstation ~]$ oc get pods
9. NAME          READY   STATUS    RESTARTS   AGE
10. mysql-1-deploy   0/1     Completed   0          120s
```

```
mysql-1-5vfn4   1/1     Running   0          100s
```

Note

Notice the name of the running pod. You need this information to be able to log in to the MySQL database server later.

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

```
12. [student@workstation ~]$ oc describe pod mysql-1-5vfn4
13. Name:      mysql-1-5vfn4
14. Namespace: ${RHT_OCP4_DEV_USER}-mysql-openshift
```

```
15.Priority:      0
16.Node:          master01/192.168.50.10
17.Start Time:    Mon, 29 Mar 2021 16:42:13 -0400
18.Labels:        deployment=mysql-1
19....output omitted...
20.Status:        Running
21.IP:            10.10.0.34
```

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

The output from the `oc describe` command might contain errors related to the readiness probe. You can ignore these errors.

22. List the services in this project and verify that the service to access the MySQL pod was created:

```
23. [student@workstation ~]$ oc get svc
24. NAME           TYPE       CLUSTER-IP      EXTERNAL-IP     PORT(S)      AGE
   mysql          ClusterIP   172.30.151.91 <none>        3306/TCP    10m
```

25. Retrieve the details of the `mysql` service using the `oc describe` command and note that the service type is `ClusterIP` by default:

```
26. [student@workstation ~]$ oc describe service mysql
27. Name:          mysql
28. Namespace:    ${RHT_OCP4_DEV_USER}-mysql-openshift
29. Labels:        app=mysql-persistent
30.             app.kubernetes.io/component=mysql-persistent
31.             app.kubernetes.io/instance=mysql-persistent
32.             template=mysql-persistent-template
33. Annotations:  openshift.io/generated-by: OpenShiftNewApp
34....output omitted...
35. Selector:     name=mysql
36. Type:          ClusterIP
37. IP Family Policy: SingleStack
38. IP Families:  IPv4
39. IP:            172.30.151.91
```

```
40. IPs:          172.30.151.91
41. Port:         mysql  3306/TCP
42. TargetPort:   3306/TCP
43. Endpoints:    10.10.0.34:3306
44. Session Affinity: None
```

```
Events: <none>
```

45. List the persistent storage claims in this project:

```
46. [student@workstation ~]$ oc get pvc
47. NAME      STATUS      VOLUME                                     CAPACITY      ...
     STORAGECLASS
mysql      Bound      pvc-e9bf0b1f-47df-4500-afb6-77e826f76c15  10Gi          ...
     standard
```

48. Retrieve the details of the mysql pvc using the oc describe command:

```
49. [student@workstation ~]$ oc describe pvc/mysql
50. Name:          mysql
51. Namespace:    ${RHT_OCP4_DEV_USER}-mysql-openshift
52. StorageClass: standard
53. Status:        Bound
54. Volume:       pvc-e9bf0b1f-47df-4500-afb6-77e826f76c15
55. Labels:        app=mysql-persistent
56.               app.kubernetes.io/component=mysql-persistent
57.               app.kubernetes.io/instance=mysql-persistent
58.               template=mysql-persistent-template
59. Annotations:   openshift.io/generated-by: OpenShiftNewApp
60....output omitted...
61. Capacity:     10Gi
62. Access Modes: RWO
63. VolumeMode:   Filesystem
64. Used By:      mysql-1-5vfn4
```

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

4. Connect to the MySQL database server and verify that the database was created successfully.

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.
2. [student@workstation ~]\$ oc port-forward mysql-1-5vfn4 3306:3306
3. Forwarding from 127.0.0.1:3306 -> 3306

```
Forwarding from [::1]:3306 -> 3306
```

4. From the workstation machine open another terminal and connect to the MySQL server using the MySQL client.

5. [student@workstation ~]\$ mysql -uuser1 -pmypa55 --protocol tcp -h localhost
6. mysql: [Warning] Using a password on the command line interface can be insecure.
7. Welcome to the MySQL monitor. Commands end with ; or \g.
8. Your MySQL connection id is 13
9. Server version: 8.0.26 Source distribution
- 10.
11. Copyright (c) 2000, 2021, Oracle and/or its affiliates.
- 12.
13. Oracle is a registered trademark of Oracle Corporation and/or its
14. affiliates. Other names may be trademarks of their respective
- 15.
16. Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
- 17.

```
mysql>
```

18. Verify the creation of the testdb database.

19. mysql> SHOW DATABASES;
20. +-----+
21. | Database |
22. +-----+

```
23. | information_schema |
24. | testdb           |
25. +-----+
```

```
2 rows in set (0.00 sec)
```

26. Exit from the MySQL prompt:

```
27. 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.

[Previous](#) [Next](#)

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 6.7: 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, Containerfiles, 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:

```
[user@host ~]$ 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:

```
[user@host ~]$ 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:

```
[user@host ~]$ 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](#).

[Previous](#) [Next](#)

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.

Make sure you have completed [the section called “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
```

Procedure 6.2. Instructions

1. Prepare the lab environment.
 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. Log in to the OpenShift cluster.
 3. [student@workstation ~]\$ oc login -u \${RHT_OCP4_DEV_USER} \
 4. > -p \${RHT_OCP4_DEV_PASSWORD} \${RHT_OCP4_MASTER_API}
 - 5.
 6. Login successful
 - 7.

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

8. Create a new project that contains your RHOPC 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.

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 \
> --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. Wait until the application finishes deploying by monitoring the progress with the oc get pods -w command:

3. [student@workstation ~]\$ oc get pods -w

4. NAME	READY	STATUS	RESTARTS	AGE
5. 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.

6. Review the service for this application using the `oc describe` command:

```
7. [student@workstation ~]$ oc describe svc/php-helloworld
8. Name:          php-helloworld
9. Namespace:    user-route
10. Labels:       app=php-helloworld
11.             app.kubernetes.io/component=php-helloworld
12.             app.kubernetes.io/instance=php-helloworld
13. Annotations: openshift.io/generated-by: OpenShiftNewApp
14. Selector:     deployment=php-helloworld
15. Type:         ClusterIP
16. IP:           172.30.100.236
17. Port:         8080-tcp  8080/TCP
18. TargetPort:   8080/TCP
19. Endpoints:   10.129.5.124:8080
20. 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:

```
4. [student@workstation ~]$ oc expose svc/php-helloworld
5. route.route.openshift.io/php-helloworld exposed
6.
7. [student@workstation ~]$ oc describe route
8. Name:          php-helloworld
9. Namespace:    user-route
10. Created:     16 seconds ago
11. Labels:      app=php-helloworld
12.           app.kubernetes.io/component=php-helloworld
13.           app.kubernetes.io/instance=php-helloworld
14. Annotations: openshift.io/host.generated=true
15. Requested Host: php-helloworld-user-route.apps.na46-stage2.dev.nextcle.com
16.           exposed on router default
17.           (host apps.na46-stage2.dev.nextcle.com) 16 seconds ago
18. Path:        <none>
19. TLS Termination: <none>
20. Insecure Policy: <none>
21. Endpoint Port: 8080-tcp
22.
23. Service:    php-helloworld
24. Weight:     100 (100%)
```

```
Endpoints: 10.129.5.124:8080
```

25. Access the service from a host external to the cluster to verify that the service and route are working.

```
26. [student@workstation ~]$ curl \
27. > 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.

28. Replace this route with a route named xyz.

1. Delete the current route:

```
2. [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.

3. Create a route for the service with a name of \${RHT_OCP4_DEV_USER}-xyz.

```
4. [student@workstation ~]$ oc expose svc/php-helloworld \
5. > --name=${RHT_OCP4_DEV_USER}-xyz
6. route.route.openshift.io/${RHT_OCP4_DEV_USER}-xyz exposed
7.
8. [student@workstation ~]$ oc describe route
9. Name:           user-xyz
10. Namespace:    user-route
11. Created:      23 seconds ago
12. Labels:       app=php-helloworld
13.             app.kubernetes.io/component=php-helloworld
14.             app.kubernetes.io/instance=php-helloworld
15. Annotations:  openshift.io/host.generated=true
16. Requested Host: user-xyz-user-route.apps.na46-stage2.dev.nextcle.com
17.               exposed on router default
18.               (host apps.na46-stage2.dev.nextcle.com) 22 seconds ago
19. Path:          <none>
```

```
20. TLS Termination: <none>
21. Insecure Policy: <none>
22. Endpoint Port: 8080-tcp
23.
24. Service: php-helloworld
25. 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.

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

```
27. [student@workstation ~]$ curl \
28. > ${RHT_OCP4_DEV_USER}-xyz-${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.

[Previous](#) [Next](#)

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 6.8: 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:

[user@host ~]\$ 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:

```
[user@host ~]$ oc new-app php~http://my.git.server.com/my-app \
```

```
> --name=myapp
```

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

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

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.

```
[user@host ~]$ 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:

```
[user@host ~]$ 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:

```
[user@host ~]$ 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:

```
[user@host ~]$ 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</code> , <code>Gemfile</code> , <code>config.ru</code>
Java EE	<code>pom.xml</code>
Node.js	<code>app.json</code> , <code>package.json</code>
PHP	<code>index.php</code> , <code>composer.json</code>
Python	<code>requirements.txt</code> , <code>config.py</code>
Perl	<code>index.pl</code> , <code>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:

```
[user@host ~]$ 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",
  "apiVersion": "image.openshift.io/v1",
  "metadata": {
    "name": "myapp",
    "creationTimestamp": null
  }
}
```

```
        "app": "myapp"
    },
    "annotations": {
        "openshift.io/generated-by": "OpenShiftNewApp"
    }
},
"spec": {
    "lookupPolicy": {
        "local": false
    }
},
"status": {
    "dockerImageRepository": ""
}
},
...output omitted...
```

Define a resource type of image stream.

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",
    "apiVersion": "build.openshift.io/v1",
    "metadata": {
        "name": "myapp",
        "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"
        }
    }
}
```

```
 },
 "strategy": {

   "type": "Source",
   "sourceStrategy": {
     "from": {
       "kind": "ImageStreamTag",
       "namespace": "openshift",
       "name": "php:7.3"
     }
   }
 },
 "output": {
   "to": {
     "kind": "ImageStreamTag",
     "name": "myapp:Latest"
   }
 },
 "resources": {},
 "postCommit": {},
 "nodeSelector": null
},
 "status": {
   "lastVersion": 0
}
},
...output omitted...
```

Define a resource type of `BuildConfig`.

Name the `BuildConfig` `myapp`.

Define the address to the source code Git repository.

Define the strategy to use S2I.

Define the builder image as the php:7.3 image stream.

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",
    "apiVersion": "apps/v1",
    "metadata": {
        "name": "myapp",
        "creationTimestamp": null,
        "labels": {
            "app": "myapp",
            "app.kubernetes.io/component": "myapp",
            "app.kubernetes.io/instance": "myapp"
        },
        "annotations": {
            "image.openshift.io/triggers": "[{\\"from\\":{\\\"kind\\\":\\\"ImageStrea
mTag\\\",\\\"name\\\":\\\"myapp:la
test\\\"},\\\"fieldPath\\\":\\\"spec.template.spec.containers[?(@.name==\\\\\"myapp\\\\\")].imag
e\\\"}]",

            "openshift.io/generated-by": "OpenShiftNewApp"
        }
    }
}
```

```
},
"spec": {
    "replicas": 1,
    "selector": {
        "matchLabels": {
            "deployment": "myapp"
        }
    },
    "template": {
        "metadata": {
            "creationTimestamp": null,
            "labels": {
                "deployment": "myapp"
            },
            "annotations": {
                "openshift.io/generated-by": "OpenShiftNewApp"
            }
        },
        "spec": {
            "containers": [
                {
                    "name": "myapp",
                    "image": " ",
                    "ports": [
                        {
                            "containerPort": 8080,
                            "protocol": "TCP"
                        },
                        {
                            "containerPort": 8443,
                            "protocol": "TCP"
                        }
                    ]
                }
            ]
        }
    }
}
```

```
        }
      ],
      "resources": {}
    }
  ]
}

},
"strategy": {}

},
"status": {}

},
...output omitted...
```

Define a resource type of Deployment.

Name the Deployment `myapp`.

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.

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

Defines the container image to deploy: `myapp:latest`.

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:

```
[user@host ~]$ 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:

```
[user@host ~]$ 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:

```
[user@host ~]$ oc get buildconfig
```

NAME	TYPE	FROM	LATEST
myapp	Source	Git	1

```
[user@host ~]$ 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](#)

[S2I GitHub repository](#)

[Previous](#) [Next](#)

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.

Make sure you have completed [the section called “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
```

Procedure 6.3. 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. 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:

```
2. [student@workstation ~]$ cd ~/D0180-apps  
3. [student@workstation D0180-apps]$ git checkout master
```

...output omitted...

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

```
5. [student@workstation D0180-apps]$ git checkout -b s2i  
6. Switched to a new branch 's2i'  
7.  
8. [student@workstation D0180-apps]$ git push -u origin s2i  
9. ...output omitted...  
10. * [new branch] s2i -> s2i
```

Branch 's2i' set up to track remote branch 's2i' from 'origin'.

11. Review the PHP source code of the application, inside the the php-helloworld folder.

Open the index.php file in the ~/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.

1. Load your classroom environment configuration.

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

```
[student@workstation DO180-apps]$ source /usr/local/etc/ocp4.config
```

2. Log in to the OpenShift cluster.

```
3. [student@workstation DO180-apps]$ oc login -u ${RHT_OCP4_DEV_USER} \  
4. > -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}  
5. Login successful
```

...output omitted...

6. Create a new project that contains your RHOCUP developer username for the resources you create during this exercise:

```
[student@workstation DO180-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 DO180-apps Git repository.

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 DO180-apps]$ oc new-app php:7.3 --name=php-helloworld \  
> https://github.com/${RHT_OCP4_GITHUB_USER}/DO180-apps#s2i \  
> --context-dir php-helloworld
```

2. Wait for the build to complete and the application to deploy. Verify that the build process starts with the `oc get pods` command.

```
3. [student@workstation openshift-s2i]$ oc get pods
```

NAME	READY	STATUS	RESTARTS	AGE
------	-------	--------	----------	-----

php-helloworld-1-build	1/1	Running	0	5s
------------------------	-----	---------	---	----

5. Examine the logs for this build. Use the build pod name for this build, `php-helloworld-1-build`.

```
6. [student@workstation D0180-apps]$ oc logs --all-containers \
7. > -f php-helloworld-1-build
8. ...output omitted...
9.
10.Writing manifest to image destination
11.Storing signatures
12.Generating dockerfile with builder image image-registry.openshift-image-...
.
13.php@sha256:3206...37b4
14.Adding transient rw bind mount for /run/secrets/rhsm
15.STEP 1: FROM image-registry.openshift-image-registry.svc:5000...
16.
17....output omitted...
18.
19.STEP 8: RUN /usr/libexec/s2i/assemble
20.
21....output omitted...
22.
23.Pushing image .../php-helloworld:latest ...
24.Getting image source signatures
25.
26....output omitted...
27.
28.Writing manifest to image destination
29.Storing signatures
30.Successfully pushed .../php-helloworld@sha256:3f1cdb278548c7f24429e2469c51
ae35482d54e2616d596ab6fb59d6b432c454
31.Push successful
32.Cloning "https://github.com/${RHT_OCP4_GITHUB_USER}/D0180-apps" ...
33. Commit: 9a042f7e3650ef38ad07af83b74f57c7a7d1820c (Added start up s
cript)
34.
```

```
...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 \${RHT_OCP4_DEV_USER}-s2i/php-helloworld:latest. The last step in the build process is to push this image to the OpenShift private registry.

35. Review the Deployment for this application:

```
36. [student@workstation D0180-apps]$ oc describe deployment/php-helloworld
37. Name:          php-helloworld
38. Namespace:    ${RHT_OCP4_DEV_USER}-s2i
39. CreationTimestamp: Tue, 30 Mar 2021 12:54:59 -0400
40. Labels:        app=php-helloworld
41.             app.kubernetes.io/component=php-helloworld
42.             app.kubernetes.io/instance=php-helloworld
43. Annotations:   deployment.kubernetes.io/revision: 2
44.             image.openshift.io/triggers:
45.             [{"from": {"kind": "ImageStreamTag", "name": "php-he
lloworld:latest"}, "fieldPath": "spec.template.spec.containers[?(@.name==\"p
hp-helloworld\")]..."}
46.             openshift.io/generated-by: OpenShiftNewApp
47. Selector:      deployment=php-helloworld
48. Replicas:      1 desired | 1 updated | 1 total | 1 available | 0
        unavailable
49. StrategyType: RollingUpdate
50. MinReadySeconds: 0
51. RollingUpdateStrategy: 25% max unavailable, 25% max surge
52. Pod Template:
53.   Labels:        deployment=php-helloworld
54.   Annotations:  openshift.io/generated-by: OpenShiftNewApp
55.   Containers:
56.     php-helloworld:
57.       Ports:        8080/TCP, 8443/TCP
58.       Host Ports:  0/TCP, 0/TCP
59.       Environment: <none>
60.       Mounts:      <none>
```

```
61. Volumes: <none>
62. Conditions:
63. Type Status Reason
64. ---- -----
65. Available True MinimumReplicasAvailable
66. Progressing True NewReplicaSetAvailable
67. OldReplicaSets: <none>
68. NewReplicaSet: php-helloworld-6f5d4c47ff (1/1 replicas created)
```

...output omitted...

69. Add a route to test the application:

```
70. [student@workstation D0180-apps]$ oc expose service php-helloworld \
71. > --name ${RHT_OCP4_DEV_USER}-helloworld
```

route.route.openshift.io/\${RHT_OCP4_DEV_USER}-helloworld exposed

72. Find the URL associated with the new route:

```
73. [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}

Note

The URL is displayed in a single line.

74. Test the application by sending an HTTP GET request to the URL you obtained in the previous step. Type the URL in a single line.

```
75. [student@workstation D0180-apps]$ curl -s \
76. > ${RHT_OCP4_DEV_USER}-helloworld-${RHT_OCP4_DEV_USER}-s2i.\
77. > ${RHT_OCP4_WILDCARD_DOMAIN}
```

Hello, World! php version is 7.3.29

The php:7.3 image stream might contain a more recent version of PHP 7.3.

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.

1. Enter the source code directory.

```
[student@workstation D0180-apps]$ cd ~/D0180-apps/php-helloworld
```

2. Edit the `index.php` file as shown below:

```
3. <?php  
4. print "Hello, World! php version is " . PHP_VERSION . "\n";  
5. print "A change is a coming!\n";
```

```
?>
```

Save the file.

6. Commit the changes and push the code back to the remote Git repository:

```
7. [student@workstation php-helloworld]$ git add .  
8. [student@workstation php-helloworld]$ git commit -m 'Changed index page contents.'  
9. [s2i b1324aa] changed index page contents  
10. 1 file changed, 1 insertion(+)  
11. [student@workstation php-helloworld]$ git push origin s2i  
12....output omitted...  
13.Counting objects: 7, done.  
14.Delta compression using up to 2 threads.  
15.Compressing objects: 100% (3/3), done.  
16.Writing objects: 100% (4/4), 417 bytes | 0 bytes/s, done.  
17.Total 4 (delta 1), reused 0 (delta 0)  
18.remote: Resolving deltas: 100% (1/1), completed with 1 local object.  
19.To https://github.com/${RHT_OCP4_GITHUB_USER}/D0180-apps
```

```
f7cd896..b1324aa s2i -> s2i
```

20. Start a new Source-to-Image build process and wait for it to build and deploy:

```

21. [student@workstation php-helloworld]$ oc start-build php-helloworld
22. build.build.openshift.io/php-helloworld-2 started
23.
24. [student@workstation php-helloworld]$ oc get pods
25. NAME READY STATUS RESTARTS AGE
26. php-helloworld-1-build 0/1 Completed 0 5m7s
27. php-helloworld-2-build 0/1 Completed 0 43s
28....output omitted...
29.
30. [student@workstation php-helloworld]$ oc logs php-helloworld-2-build -f
31....output omitted...
32.
33. Successfully pushed .../php-helloworld:latest@sha256:74e757a4c0edaeda497da
   b7...

```

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.

34. After the second build has completed use the `oc get pods` command to verify that the new version of the application is running.

```

35. [student@workstation php-helloworld]$ oc get pods
36. NAME READY STATUS RESTARTS AGE
37. php-helloworld-1-build 0/1 Completed 0 11m
38. php-helloworld-1-deploy 0/1 Completed 0 10m
39. php-helloworld-2-build 0/1 Completed 0 45s
40. php-helloworld-2-deploy 0/1 Completed 0 16s

```

<code>php-helloworld-2-wq9wz</code>	1/1	Running	0	13s
-------------------------------------	-----	---------	---	-----

41. Test that the application serves the new content. Type the URL in a single line.

```
42. [student@workstation php-helloworld]$ curl -s \
```

```
43. > ${RHT_OCP4_DEV_USER}-helloworld-${RHT_OCP4_DEV_USER}-s2i.\n44. > ${RHT_OCP4_WILDCARD_DOMAIN}\n45. Hello, World! php version is 7.3.29
```

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.

[Previous](#) [Next](#)

Creating Applications with the OpenShift Web Console

Objectives

After completing this section, students should be able to:

- Create an application with the OpenShift web console.
- Manage and monitor the build cycle of an application.
- Examine resources for an application.

Accessing the OpenShift Web Console

The OpenShift web console allows users to execute many of the same tasks as the OpenShift command-line client. You can create projects, add applications to projects, view application resources, and manipulate application configurations as needed. The OpenShift web console runs as one or more pods, each pod running on a Control Plane.

Although the command-line client is more versatile than the web-console, you may find that the visual representation of the concepts in OpenShift will be helpful learning.

The web console runs in a web browser.

The default URL is of the format `https://console-openshift-console.${RHT_OCP4_WILDCARD_DOMAIN}/` By default, OpenShift generates a self-signed certificate for the web console. You must trust this certificate in order to gain access.

The web console uses a REST API to communicate with the OpenShift cluster. By default, the REST API endpoint is accessed with a different DNS name and self-signed certificate. You must also trust this certificate for the REST API endpoint.

After you have trusted the two OpenShift certificates, the console requires authentication to proceed.

Managing Projects

Upon successful login, the **Home → Projects** page displays a list of projects you can access. From this page you can create, edit, or delete a project.

If you have permission to view cluster metrics, then you are instead redirected to the **Home → Overview** page. This page shows general information and metrics about the cluster. The > Overview menu item is hidden from users without authority to view cluster metrics.

Name	Display Name	Status	Requester
PR todo-app	No display name	Active	your-user
PR hello-world	No display name	Active	your-user

Figure 6.9: OpenShift web console home page

The ellipsis icon at the end of each row provides a menu with project actions. Select the appropriate entry to edit or delete the project.

If you click a project link in this view, you are redirected to the **Project Status** page which shows all of the applications created within that project space.

Navigating the Web Console

A navigation menu is located on the left side of the web console. The menu includes two perspectives: **Administrator** and **Developer**. Each item in the menu expands to provide access to a set of related management functions. When the **Administrator** perspective is selected, these items are as follows:

Home

The home menu allows users to quickly access projects and resources. From this menu, you can browse and manage projects, search or explore cluster resources, and inspect cluster events.

Operators

The **OperatorHub** is a catalog that allows you to discover, search and install operators in the cluster. After installing an operator, you can use the **Installed Operators** option to manage the operator or search for other installed operators.

Note

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

Workloads

These options enable management of several types of Kubernetes and OpenShift resources, such as pods and deployments. Other advanced deployment options that are accessible from this menu, such as configuration maps, secrets, and cron jobs, are beyond the scope of the course.

Networking

This menu contains options to manage OpenShift resources that affect application access, such as services and routes, for a project. Other options for configuring an OpenShift Network Policy or Ingress are available, but these topics are outside the scope of this course.

Storage

This menu contains options to configure persistent storage for project applications. In particular, persistent volumes and persistent volume claims for a project are managed from the **Storage** menu.

Builds

The **Build Configs** option displays a list of project build configurations. Click a build configuration link in this view to access an overview page for the specified build configuration. From this page, you can view and edit the application's build configuration.

The **Builds** option provides a list of recent build processes for application container images in the project. Click the link for a particular build to access the build logs for that particular build process.

The **Image Streams** option provides a list of image streams defined in the project. Click an image stream entry in this list to access an overview page to view and manage that image stream.

Compute

Provides options to access and manage the compute nodes of the OpenShift cluster. From this page, you can also configure node health checks and autoscaling.

User management

From this option, you can configure authentication and authorization using users, groups, roles, role bindings and service accounts.

Administration

Provides options to manage cluster and project settings, such as resource quotas and role-based access controls. Functions in the **Administration** section are outside the scope of this course.

Creating New Applications

From the **Developer** perspective, use **+Add** to select a way to create a new application in an OpenShift project. You can add an application from the **Developer Catalog**, by using the **All services** option, which offers a selection of Source-to-Image (S2I) templates, builder images and Helm charts to create technology-specific applications. Select a desired template, and provide the necessary information to deploy the new application.

You are not limited to deploying an application from the catalog. You can also deploy an application using:

- A container image hosted on a remote container registry.
- A YAML file that specifies the Kubernetes and OpenShift resources to create.
- A builder image using the source code or a Containerfile from your own git repository.

The screenshot shows the Red Hat OpenShift Container Platform interface. The top navigation bar includes the Red Hat logo and the text "Red Hat OpenShift Container Platform". Below the header, a left sidebar menu is visible with options like "Developer", "+Add", "Topology", "Observe", "Search", "Builds", "Helm", "Project", "ConfigMaps", and "Secrets". The main content area is titled "Add" and features a "Getting started resources" section with links to "Create applications using samples", "Build with guided documentation", and "Explore new developer features". It also includes sections for "Developer Catalog" (with "All services" and "Database" options), "Git Repository" (with "Import from Git"), "Container images" (with "Samples" and "Import YAML" options), and "From Local Machine". A "Project: sample-project" dropdown is at the top, and a "Details on" toggle is in the top right corner.

Figure 6.10: OpenShift Developer Catalog page

To create an application with one of these methods, select the appropriate option in the **Add** page. Use the **Container images** option to deploy an existing container image. Use the **Import YAML** option to create the resources specified in a YAML file. Use the **Import from Git** option to deploy your source code using a builder image or your Containerfile. The **Database**, **Operator Backed** and **Helm Chart** options are shortcuts to the catalog.

Managing Application Builds

From the **Administrator** perspective, click the **Build Configs** option of the **Builds** menu after you add a Source-to-Image application to a project. The new build configuration is accessible from this view:

The screenshot shows the Red Hat OpenShift Container Platform interface. The top navigation bar includes the Red Hat logo, 'Red Hat OpenShift Container Platform', and a user dropdown for 'OpenShift Administrator'. The left sidebar, titled 'Administrator', has a 'Builds' section with 'Build Configs' selected and highlighted by a red box. Other options in the sidebar include Home, Operators, Workloads, Networking, Storage, Builds, and Image Streams. The main content area is titled 'Build Configs' and features a table with columns: Name, Namespace, Labels, and Created. A single row is shown: 'BC todoapp' (Namespace: test, Labels: app=todoapp), created on 'Aug 3, 1:42 pm'. There are also 'Filter' and 'Search by name...' search bars at the top of the table.

Figure 6.11: OpenShift build configurations page

Click a build configuration in the list to view an overview page for the selected build configuration. From the overview page, you can:

- View the build configuration parameters, such as the URL for the source code's Git repository.
- View and edit the environment variables that are set in the builder container, during an application build process.
- View a list of recent application builds, and click a selected build to access logs from the build process.

Managing Deployed Applications

The Workloads menu provides access to deployments in the project:

The screenshot shows the Red Hat OpenShift Container Platform Web Console. The top navigation bar includes the Red Hat logo, 'Red Hat OpenShift Container Platform', and a user dropdown for 'OpenShift Administrator'. The left sidebar, titled 'Administrator', has a 'Workloads' section with several options: 'Pods', 'Deployments', 'Deployment Configs' (which is highlighted with a red box), 'Stateful Sets', 'Secrets', and 'Config Maps'. The main content area is titled 'Deployment Configs' with a 'Create Deployment Config' button. It shows a table with one row for a deployment config named 'todoapp' in namespace 'test'. The table columns are 'Name' (DC todoapp), 'Namespace' (NS test), 'Status' (1 of 1 pods), 'Labels' (app=todoapp), and 'Pod Selector' (app=todoapp, deploymentconfig=todoapp). A search bar at the top of the content area is set to 'test'.

Figure 6.12: OpenShift Workloads menu

Under workloads, you will find many of the constructs discussed in the course including pods, Deployments, and Config Maps.

Click a deployments entry in the list to view an overview page for the selection. From the overview page, you can:

- View the deployments parameters, such as the specifications of an application container image.
- Change the desired number of application pods to manually scale the application.
- View and edit the environment variables that are set in the deployed application container.
- View a list of application pods, and click a selected pod to access logs for that pod.

Other Web Console Features

The web console allows you to:

- Manage resources, such as project quotas, user membership, secrets, and other advanced resources.
- Create persistent volume claims.
- Monitor builds, deployments, pods, and system events.
- Create continuous integration and deployment pipelines.

Detailed usage for the above features is outside the scope of this course.

[Previous](#) [Next](#)

Guided Exercise: Creating an Application with the Web Console

In this exercise, you will create, build, and deploy an application to an OpenShift cluster using the OpenShift web console.

Outcomes

You should be able to create, build, and deploy an application to an OpenShift cluster using the web console.

Make sure you have completed [the section called “Guided Exercise: Configuring the Classroom Environment”](#) from *Chapter 1* before executing any command of this practice.

Get the lab files by executing the lab script:

```
[student@workstation ~]$ lab openshift-webconsole start
```

The lab script verifies that the OpenShift cluster is running.

Procedure 6.4. Instructions

1. Inspect the PHP source code for the sample application and create and push a new branch named `console` to use during this exercise.

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:

2. [student@workstation ~]\$ cd ~/D0180-apps
3. [student@workstation D0180-apps]\$ git checkout master

...output omitted...

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

5. [student@workstation D0180-apps]\$ git checkout -b `console`

```
6. Switched to a new branch 'console'  
7.  
8. [student@workstation D0180-apps]$ git push -u origin console  
9. ...output omitted...  
10. * [new branch]      console -> console
```

Branch 'console' set up to track remote branch 'console' from 'origin'.

11. Review the PHP source code of the application, inside the the php-helloworld folder.

Open the index.php file in the ~/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. Open a web browser and navigate to `https://console-openshift-console.${RHT_OCP4_WILDCARD_DOMAIN}` to access the OpenShift web console. Log in and create a new project named `youruser-console`.

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. Retrieve the value of the wildcard domain specific to your cluster, using the `${RHT_OCP4_WILDCARD_DOMAIN}`

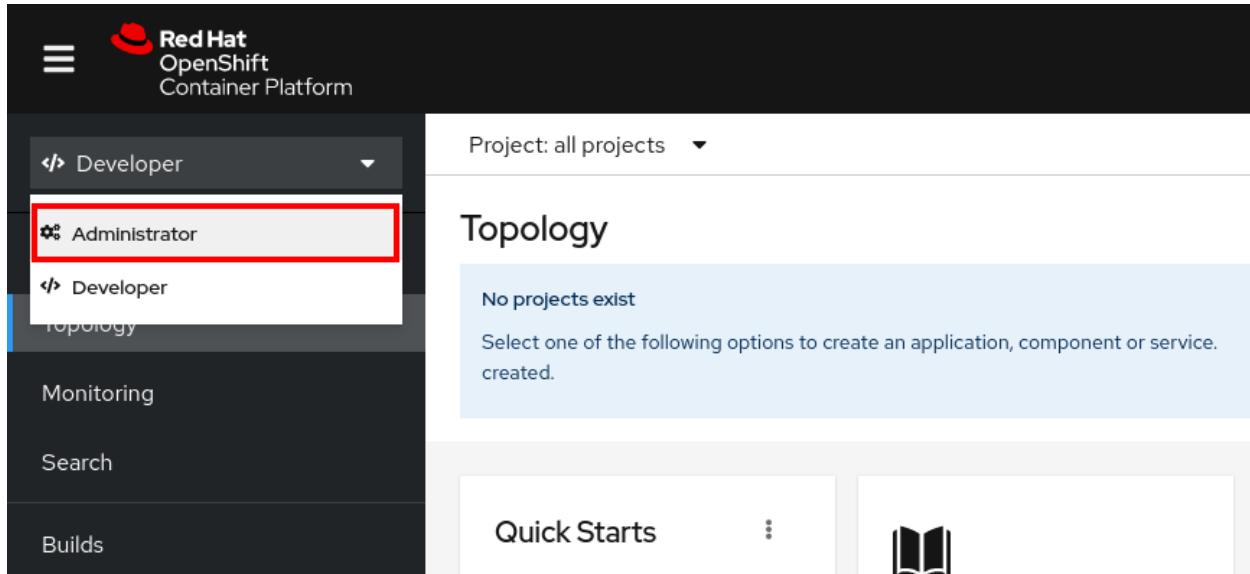
```
3. [student@workstation ~]$ echo ${RHT_OCP4_WILDCARD_DOMAIN}
```

`apps.cluster.lab.example.com`

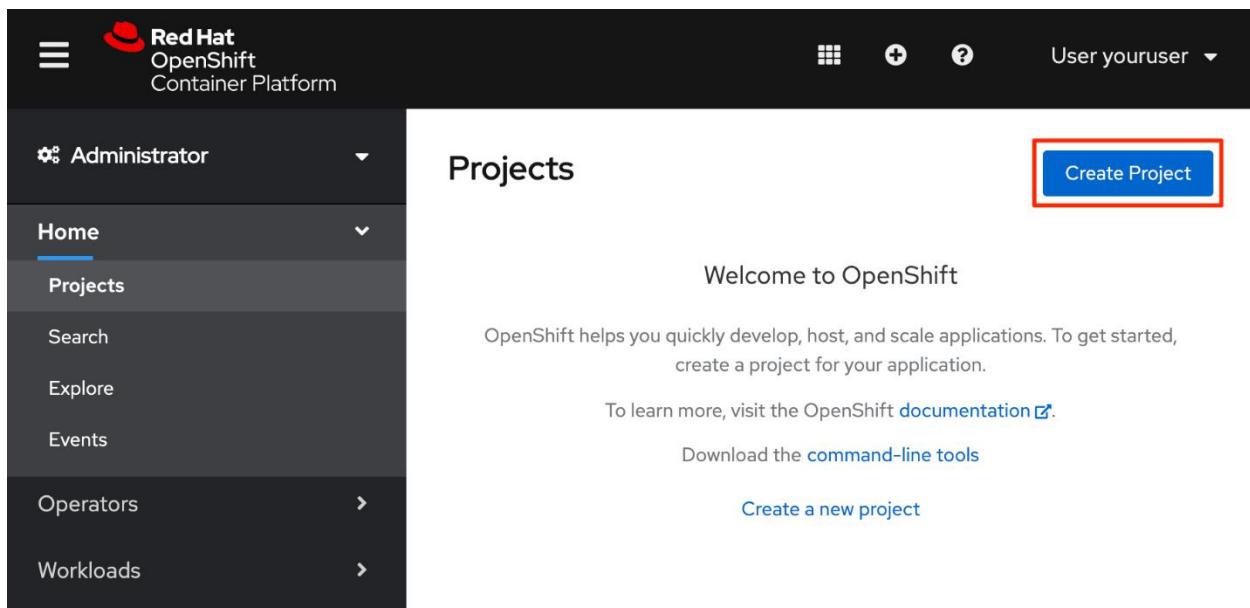
4. Open the Firefox browser and navigate to `https://console-openshift-console.${RHT_OCP4_WILDCARD_DOMAIN}` to access the OpenShift web

console. Click **RedHatTraining**, and then log in to the OpenShift console using your credentials.

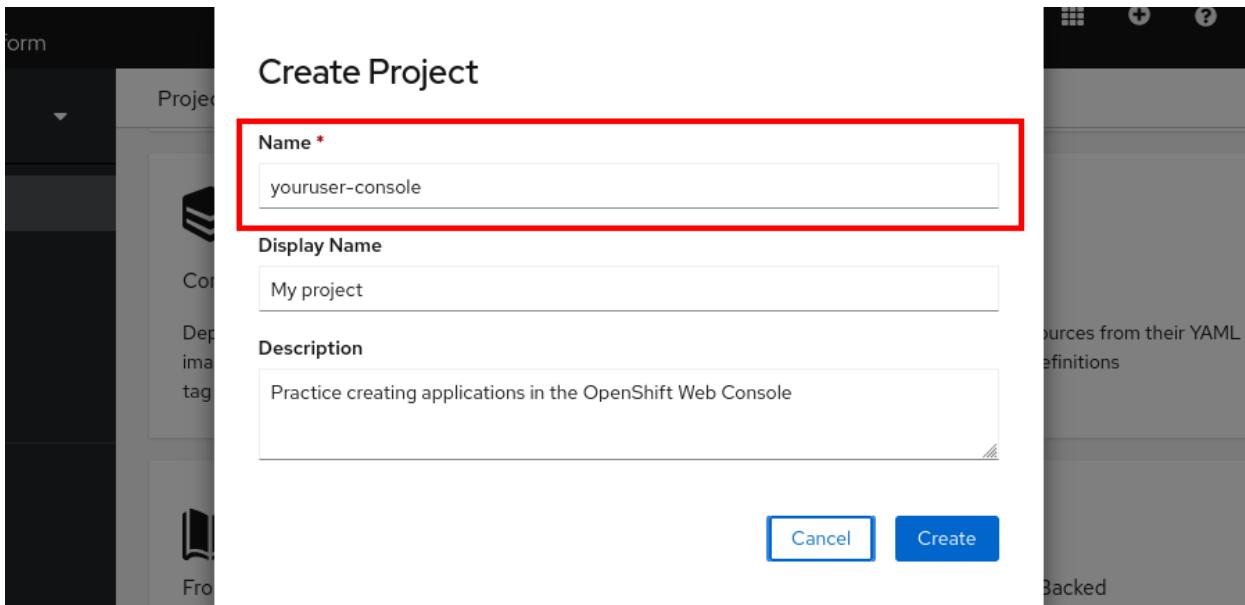
5. Select the **Administrator** perspective from the list at the top of the left menu.



6. Figure 6.13: Administrator perspective drop-down
7. Create a new project named *youruser-console*. You can type any values you prefer in the other fields.



8. Figure 6.14: Create a new project



9. Figure 6.15: Create a new project

10. After you have completed the required fields, click **Create** in the Create Project dialog box to go to the Project Status page for the *youruser-console* project:

Details		Status		Activity	
Name	youruser-console	Active		View events	
Requester	youruser	Utilization		Ongoing	
Labels	No labels	Resource	Usage	Recent Events	

11. Figure 6.16: Project Status page

3. Create the new `php-helloworld` application with a PHP template.

1. Select the **Developer** perspective from the list at the top of the left menu.

The screenshot shows the developer perspective drop-down menu on the left, with 'Developer' selected. On the right, the 'Project Details' page for 'youruser-console' is displayed, showing tabs for Overview, Details, YAML, Workloads, and Role Bindings. The Overview tab is active, showing details like Name: youruser-console and Status: Active.

2. Figure 6.17: Developer perspective drop-down
3. Click **+Add** on the left menu. Scroll on the opened page, click **From Catalog**, to open the Developer Catalog section.

The screenshot shows the 'From Catalog' section of the Developer Catalog page. It includes a description: 'Browse the catalog to discover, deploy and connect to services'. Other sections visible include 'Database' and 'Operator Backed'.

4. Figure 6.18: Developer Catalog page
5. Enter php in the **Filter by keyword** field.

The screenshot shows the 'Developer Catalog' page with a search bar containing 'php' highlighted. The results show two items: 'CakePHP + MySQL' and another 'CakePHP + MySQL' entry. The search bar has a placeholder 'All items' and a clear button 'x'.

6. Figure 6.19: Finding PHP-related templates
7. After filtering, click the **PHP builder image** to display the PHP dialog box. Click **Create Application** to display the **Create Source-to-Image Application** page.

The screenshot shows the Red Hat OpenShift web console interface. On the left, there's a dark sidebar with various developer tools: +Add, Topology, Observe, Search, Builds, Helm, Project, ConfigMaps, and Secrets. The 'Developer' option is selected. At the top, it says 'Project: youruser-console' and 'Application: all applications'. The main area is titled 'Create Source-to-Image Application'. It has a dropdown for 'Builder Image version' with 'IST 7.4-ubi8' selected. Below that, 'PHP 7.4 (UBI 8)' is highlighted with a blue background. A note says 'BUILDER PHP' and provides details about using PHP 7.4 on UBI 8. It also mentions a sample repository: <https://github.com/sclorg/cakephp-ex.git>. The next section is 'Git', which includes a 'Git Repo URL' input field that is empty and has a red border, indicating it is required. There is also a small red exclamation mark icon in the top right corner of the input field.

8. Figure 6.20: Configuring Source-to-Image for a PHP application
9. Change the **Builder Image Version** to PHP version 7.4 (UBI 8).

Specify the location of the source code git repository: <https://github.com/yourgituser/D0180-apps>

Use the **Advanced Git Options** to set the context directory to `php-helloworld` and branch `console` for this exercise

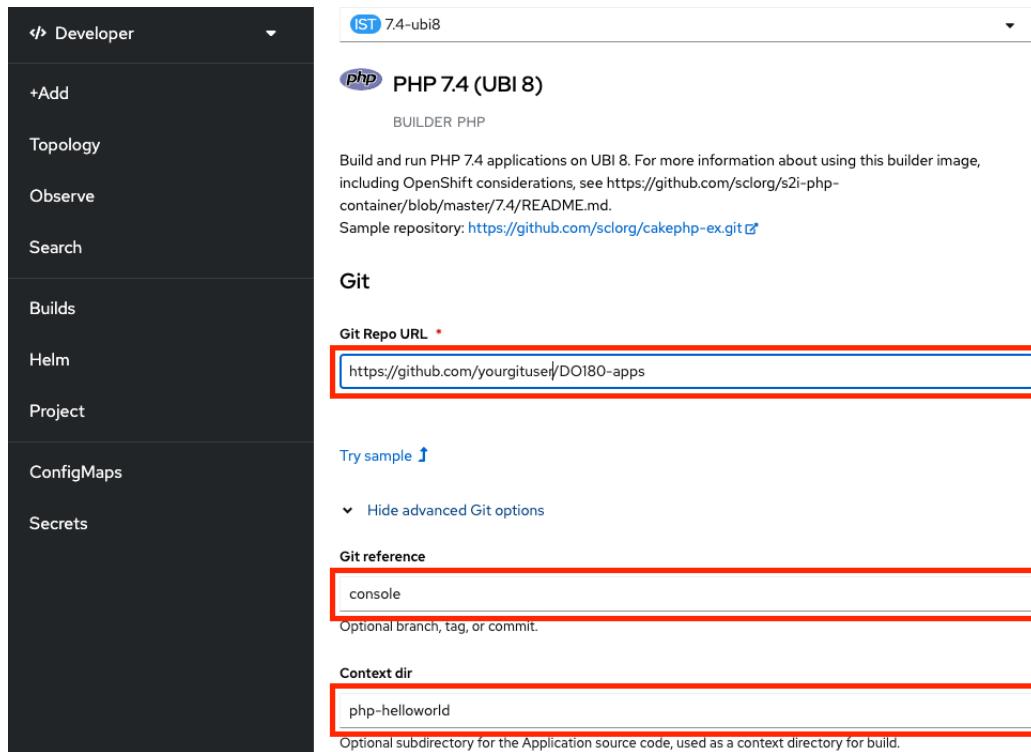


Figure 6.21: Setting Advanced Git Options for the application
Enter php-helloworld for both the application name and the name used
for associated resources. Select **Deployment** as the resource type.

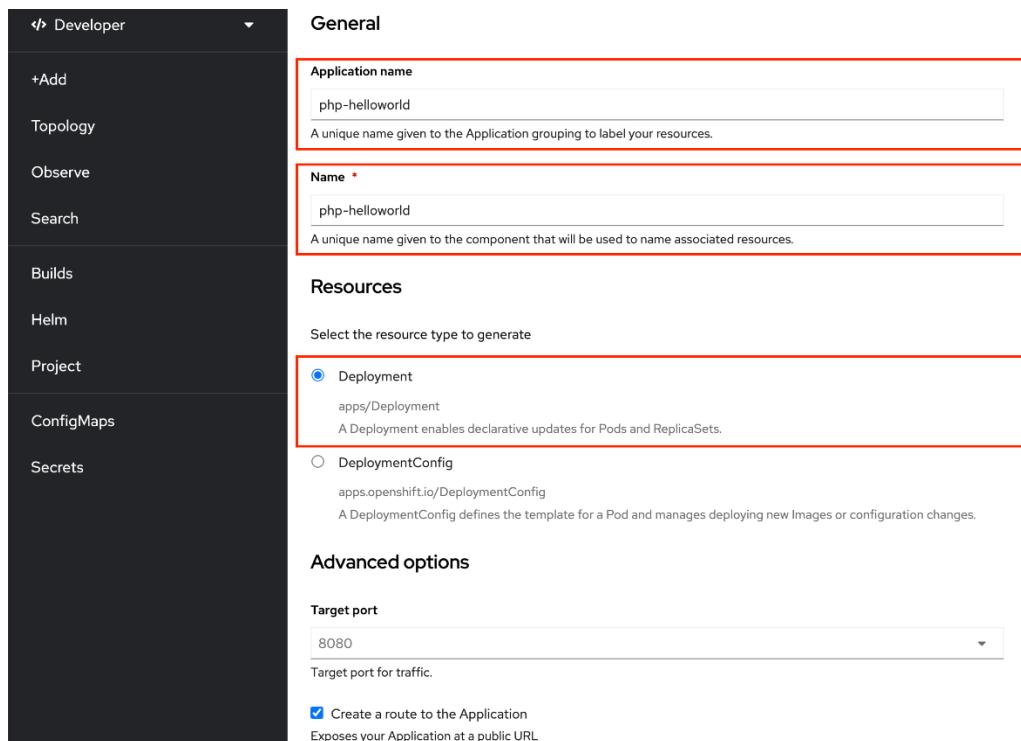


Figure 6.22: Setting application options
Scroll to the bottom of the page, and select **Create a route to the application**. Click **Create** to create the required OpenShift and Kubernetes resources for the application.

You are redirected to the Topology page:

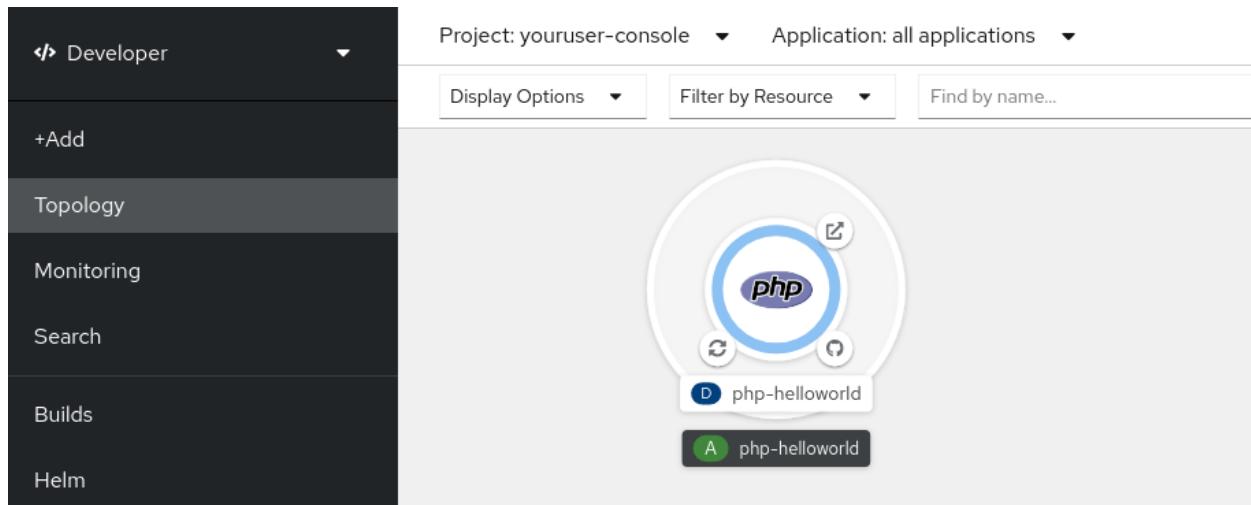
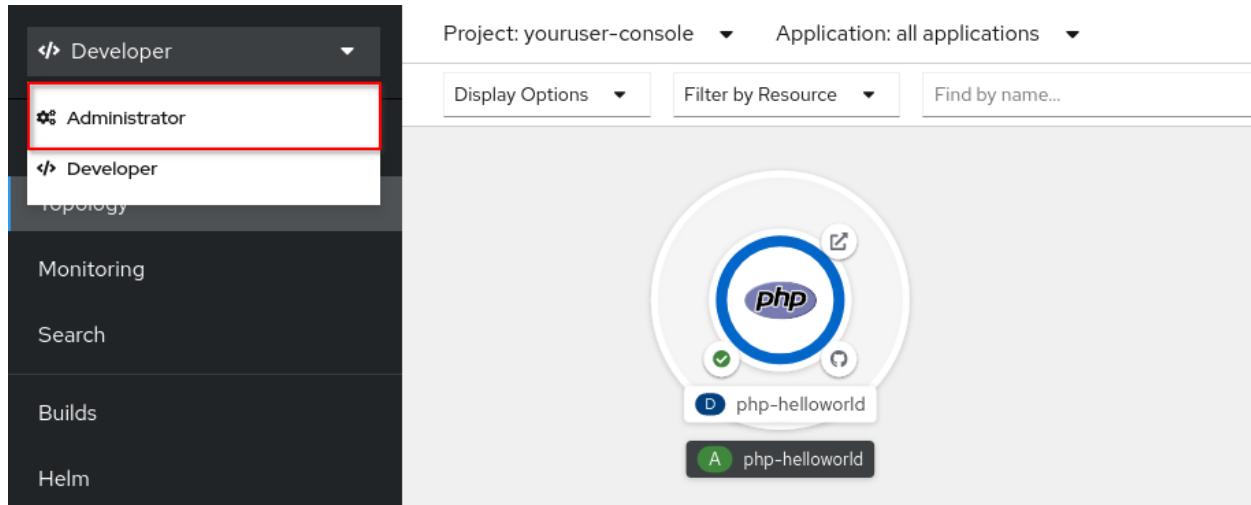


Figure 6.23: Topology page
This page indicates that the php-helloworld application is created.
The D annotation to the left of the php-helloworld link is an acronym for Deployment. This link redirects to a page containing information about the application's deployment.

10. Switch back to the **Administrator** perspective for the remainder of the exercise:



11. Figure 6.24: Administrator perspective drop-down

4. Use the navigation bar on the left side of the OpenShift web console to locate information for the application's OpenShift and Kubernetes resources:
 1. Deployments
 2. BuildConfig
 3. Build Logs
 4. Service
 5. Route
 6. Examine the deployment. In the navigation bar, click **Workloads → Deployments** to display a list of deployments for the *youruser-console* project. Click the [php-helloworld](#) link to display deployment details.

The screenshot shows the OpenShift web console interface. On the left, there is a dark sidebar with a navigation menu. The 'Workloads' section is expanded, and 'Deployments' is selected, highlighted with a blue border. Other options in this section include DeploymentConfigs, StatefulSets, Secrets, and ConfigMaps. Below this are sections for CronJobs, Jobs, DaemonSets, ReplicaSets, ReplicationControllers, and HorizontalPodAutoscalers. At the top right, the project is set to 'youruser-console'. The main content area shows the 'Deployment details' for the 'php-helloworld' deployment. The deployment has 1 pod. It is currently using a RollingUpdate strategy with 25% of 1 pod unavailable. The max surge is 25% greater than 1 pod. The progress deadline is set to 600 seconds. The deployment is in the 'Pending' state with a progress of 0%. The 'Metrics' tab is visible but not selected. The 'Edit' button is located at the bottom right of the deployment summary.

7. Figure 6.25: Application deployment details page

8. Explore the available information from the **Details** tab. The build may still be running when you reach this page, so the Deployment might not have a value of 1 pod, yet.
9. If you click the up and down arrow icons next to the doughnut chart that indicates the number of pods, you can scale the application up and down horizontally.

10. Examine the build configuration. In the navigation bar, click **Builds** → **Build Configs** to display a list of build configurations for the *youruser-console* project. Click the [php-helloworld](#) link to display the build configuration for the application.

11. Figure 6.26: Application build configuration details page
12. Explore the available information from the **Details** tab. The **YAML** tab allows you to view and edit the build configuration as a YAML file. The **Builds** tab provides an historical list of builds, along with a link to more information for each build.
13. The **Environment** tab allows you to view and edit environment variables for the application's build environment. The **Events** tab displays a list of build related events and metadata.
14. Examine the logs for the Source-to-Image build of the application. In the **Builds** menu, click Builds to display a list of recent builds for the *youruser-console* project.

Click the [php-helloworld-1](#) link to access information for the first build of the *php-helloworld* application:

The screenshot shows the 'Build details' page for the 'php-helloworld-1' build. The left sidebar lists various Kubernetes resources like Deployments, DeploymentConfigs, StatefulSets, etc. The main area shows the build status as 'Complete'. The 'Logs' tab is active, displaying the following log output:

```

Log stream ended. Search Debug container
Wrap lines | Raw | Download | Expand
63 lines
43 Copying blob sha256:7f1fd656af6e7b6afb2b3523a01c717b13d94bd9dede6c1f0061ed9e70b22d65
44 Copying config sha256:98a358743a0541f78e13a5a14a9aadd4fd4acdc3505ab5d057fc274581887212
45 Writing manifest to image destination
46 Storing signatures
--> 98a358743a0541f78e13a5a14a9aadd4fd4acdc3505ab5d057fc274581887212
48 Successfully tagged temp.builder.openshift.io/jmzzjv-console/php-helloworld-1:3454a6ca
98a358743a0541f78e13a5a14a9aadd4fd4acdc3505ab5d057fc274581887212
50 Pushing image image-registry.openshift-image-registry.svc:5000/jmzzjv-console/php-helloworld:latest ...
51 Getting image source signatures
52 Copying blob sha256:7f1fd656af6e7b6afb2b3523a01c717b13d94bd9dede6c1f0061ed9e70b22d65
53 Copying blob sha256:5dcdbc60a6b60326f98e2b49d6ebcb771df4b70c6297ddf2d7dede6692df6e
54 Copying blob sha256:79a56ba0a4a301eb949644bc29f18b1879b6f385091ef1eb8068a0f5828db863
55 Copying blob sha256:8671113e1c57d3106acae2383f9bbfe1c45a26eacb03ec82786a494e15956c3
56 Copying blob sha256:12b8b1afe30f6fe8a264840ad3f250b82b335b9ebdca922db57b48836cee0c8c
57 Copying blob sha256:aad543859364662dd264ad5752fd9449d47410b9efa0278463c0a9c578b79c6
58 Copying blob sha256:98a358743a0541f78e13a5a14a9aadd4fd4acdc3505ab5d057fc274581887212
59 Copying config sha256:98a358743a0541f78e13a5a14a9aadd4fd4acdc3505ab5d057fc274581887212
60 Writing manifest to image destination
61 Storing signatures

```

Figure 6.27: An application build details page
Explore the available information from the **Details** tab. Next, click the **Logs** tab. A scrollable text box contains output from the build process:

The screenshot shows the 'Logs' tab for the 'php-helloworld-1' build. The left sidebar shows the build status as 'Complete'. The main area displays the log output for the build process:

```

Log stream ended. Search Debug container
Wrap lines | Raw | Download | Expand
63 lines
43 Copying blob sha256:7f1fd656af6e7b6afb2b3523a01c717b13d94bd9dede6c1f0061ed9e70b22d65
44 Copying config sha256:98a358743a0541f78e13a5a14a9aadd4fd4acdc3505ab5d057fc274581887212
45 Writing manifest to image destination
46 Storing signatures
--> 98a358743a0541f78e13a5a14a9aadd4fd4acdc3505ab5d057fc274581887212
48 Successfully tagged temp.builder.openshift.io/jmzzjv-console/php-helloworld-1:3454a6ca
98a358743a0541f78e13a5a14a9aadd4fd4acdc3505ab5d057fc274581887212
50 Pushing image image-registry.openshift-image-registry.svc:5000/jmzzjv-console/php-helloworld:latest ...
51 Getting image source signatures
52 Copying blob sha256:7f1fd656af6e7b6afb2b3523a01c717b13d94bd9dede6c1f0061ed9e70b22d65
53 Copying blob sha256:5dcdbc60a6b60326f98e2b49d6ebcb771df4b70c6297ddf2d7dede6692df6e
54 Copying blob sha256:79a56ba0a4a301eb949644bc29f18b1879b6f385091ef1eb8068a0f5828db863
55 Copying blob sha256:8671113e1c57d3106acae2383f9bbfe1c45a26eacb03ec82786a494e15956c3
56 Copying blob sha256:12b8b1afe30f6fe8a264840ad3f250b82b335b9ebdca922db57b48836cee0c8c
57 Copying blob sha256:aad543859364662dd264ad5752fd9449d47410b9efa0278463c0a9c578b79c6
58 Copying blob sha256:98a358743a0541f78e13a5a14a9aadd4fd4acdc3505ab5d057fc274581887212
59 Copying config sha256:98a358743a0541f78e13a5a14a9aadd4fd4acdc3505ab5d057fc274581887212
60 Writing manifest to image destination
61 Storing signatures

```

Figure 6.28: Logs for an application build
When Podman builds a container image, similar output is observed compared with the output shown in the browser.

15. Locate information for the `php-helloworld` application's service. In the navigation bar, click **Networking** → **Services** to display a list of services for the `youruser-console` project. Click the `php-helloworld` link to display the information associated with the application's service:

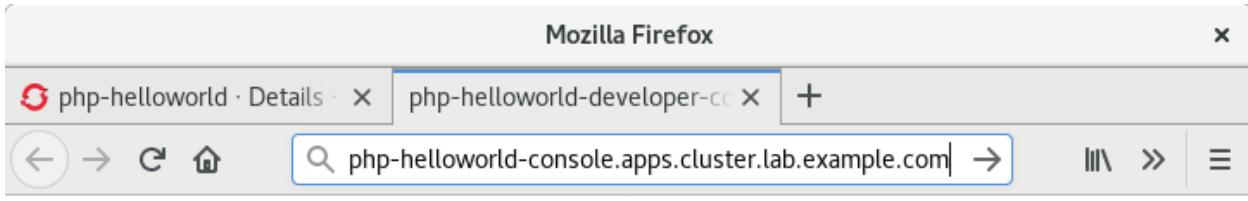
16. Figure 6.29: Service details page

17. Explore the available information from the **Details** tab. The **YAML** tab allows you to view and edit the service configuration, as a YAML file. The **Pods** tab displays the current list of pods that provide the application service.
18. Locate external route information for the application. On the navigation bar, click **Networking → Routes** to display a list of configured routes for the *youruser-console* project. Click the *php-helloworld* link to display information associated with the application's route:

19. Figure 6.30: Route details page

20. Explore the available information from the **Details** tab.

The **Location** field provides a link to the external route for the application; [https://php-helloworld-\\$\{RHT_OCP4_DEV_USER\}-console.\\$\{RHT_OCP4_WILDCARD_DOMAIN\}](https://php-helloworld-$\{RHT_OCP4_DEV_USER\}-console.$\{RHT_OCP4_WILDCARD_DOMAIN\}). Click the link to access the application in a new tab:



21. Figure 6.31: Initial PHP application results

5. Modify the application code, commit the change, push the code to the remote Git repository, and trigger a new application build.
 1. Enter the source code directory:

```
[student@workstation D0180-apps]$ cd ~/D0180-apps/php-helloworld  
  
2. Add the second print line statement in the index.php page to read "A  
change is in the air!" and save the file. Add the change to the Git index,  
commit the change, and push the changes to the remote Git repository.  
  
3. [student@workstation php-helloworld]$ vim index.php  
4.  
5. [student@workstation php-helloworld]$ cat index.php  
6. <?php  
7. print "Hello, World! php version is " . PHP_VERSION . "\n";  
8. print "A change is in the air!\n";  
9. ?>  
10. [student@workstation php-helloworld]$ git add index.php  
11. [student@workstation php-helloworld]$ git commit -m 'updated app'  
12. [console d198fb5] updated app  
13....output omitted...  
14. 1 file changed, 1 insertion(+), 1 deletion(-)  
15. [student@workstation php-helloworld]$ git push origin console  
16. Counting objects: 7, done.
```

```
17. Delta compression using up to 2 threads.  
18. Compressing objects: 100% (3/3), done.  
19. Writing objects: 100% (4/4), 409 bytes | 0 bytes/s, done.  
20. Total 4 (delta 1), reused 0 (delta 0)
```

...output omitted...

21. Trigger an application build manually from the web console.

On the navigation bar, click **Builds** → **Build Configs** and then click the `php-helloworld` link to access the Build Config Details page. From the **Actions** menu in the upper right of the screen, click **Start Build**:

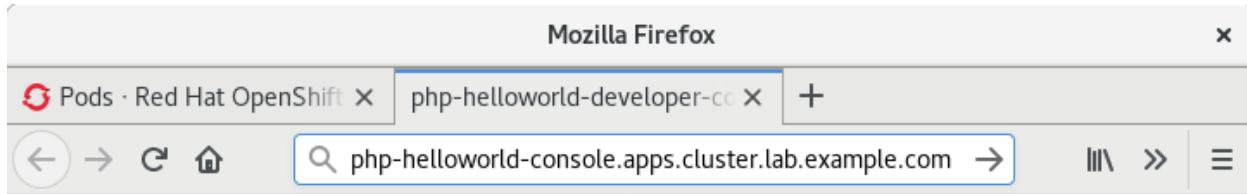
The screenshot shows the OpenShift web interface. On the left is a sidebar with sections for Administrator, Home, Operators, Workloads, and Networking. Under Networking, there are sub-options for Services, Routes, and Ingresses. The main content area has a header 'Project: youruser-console'. Below it, a breadcrumb navigation shows 'BuildConfigs > BuildConfig details' and a link to 'php-helloworld'. The main content is titled 'BuildConfig details' and shows two entries: 'Name: php-helloworld' with 'Type: Source' and 'Namespace: NS youruser-console' with 'Type: Git repository' and URL 'https://github.com/yourgithubuser/DO180-apps'. To the right of the main content is an 'Actions' dropdown menu with options: Start build (highlighted with a red box), Edit labels, Edit annotations, Edit BuildConfig, and Delete BuildConfig.

Figure 6.32: Start an application build

You are redirected to a Build Details page for the new build. Click the **Logs** tab to monitor progress of the build. The last line of a successful build contains `Push successful`.

When the build completes, the deploy starts. Go to the **Workloads** → **Pods** section, and wait for the new pod is deployed and running.

22. Reload the `http://php-helloworld-${RHT_OCP4_DEV_USER}-console.${RHT_OCP4_WILDCARD_DOMAIN}` URL in the browser. The application response corresponds to the updated source code:



23. Figure 6.33: Updated web application output

6. Delete the project. On the navigation bar, click **Home** → **Projects**. Click the icon at the right side of the row containing an entry for the *youruser-console* project. Click **Delete Project** from the menu that appears.

Name	Display Name	Status	Requester
PR youruser-console	My Project	Active	youruser

7. Figure 6.34: Delete a project

8. Enter *youruser-console* in the confirmation dialog box, and click **Delete**.

Finish

On workstation, run the following script to complete this lab.

```
[student@workstation php-helloworld]$ lab openshift-webconsole finish
```

This concludes the guided exercise.

[Previous](#) [Next](#)

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 Containerfiles, 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.

[Previous](#) [Next](#)

Chapter 7. Deploying Multi-Container Applications

Considerations for Multi-Container Applications

Guided Exercise: Deploying the Web Application and MySQL on Linux Containers

Deploying a Multi-Container Application on OpenShift

Guided Exercise: Creating an Application on OpenShift

Deploying a Multi-container Application on OpenShift Using a Template

Guided Exercise: Creating an Application with a Template

Lab: Deploying Multi-Container Applications

Summary

Abstract

Goal	Deploy applications that are containerized using multiple container images.
Objectives	<ul style="list-style-type: none">• Describe considerations for containerizing applications with multiple container images.• Deploy a multi-container application on OpenShift Platform.• Deploy an Application on OpenShift using a template.
Sections	<ul style="list-style-type: none">• Considerations for Multi-Container Applications (and Guided Exercise)• Deploying a Multi-Container Application on OpenShift (and Guided Exercise)• Deploying a Multi-container Application on OpenShift Using a Template (and Guided Exercise)
Lab	<ul style="list-style-type: none">• Deploying Multi-Container Applications

Considerations for Multi-Container Applications

Objectives

After completing this section, students should be able to:

- Describe considerations for containerizing applications with multiple container images.
- Leverage networking concepts in containers.
- Create a multi-container application with Podman.
- Describe the architecture of the To Do List application.

Leveraging Multi-Container Applications

The examples shown so far throughout this course have worked fine with a single container. A more complex application, however, can get the benefits of deploying different components into different containers. Consider an application composed of a front-end web application, a REST back end, and a database server. Those components may have different dependencies, requirements, and lifecycles.

Although it is possible to orchestrate multi-container applications' containers manually, Kubernetes and OpenShift provide tools to facilitate orchestration. Attempting to manually manage dozens or hundreds of containers quickly becomes complicated. In this section, we are going to return to using Podman to create a simple multi-container application to demonstrate the underlying manual steps for container orchestration. In later sections, you will use Kubernetes and OpenShift to orchestrate these same application containers.

Discovering Services in a Multi-Container Application

Rootfull containers

Podman uses Container Network Interface (CNI) to create a software-defined network (SDN) between all containers in the host. Unless stated otherwise, CNI assigns a new IP address to a container when it starts.

Each container exposes all ports to other containers in the same SDN. As such, services are readily accessible within the same network. The containers expose ports to external networks only by explicit configuration.

Due to the dynamic nature of container IP addresses, applications cannot rely on either fixed IP addresses or fixed DNS host names to communicate with middleware services and other application services. Containers with dynamic IP addresses can become a problem when working with multi-container applications because each container must be able to communicate with others to use services upon which it depends.

Figure 7.1: A restart breaks three-tiered application links

For example, consider an application composed of a front-end container, a back-end container, and a database. The front-end container needs to retrieve the IP address of the back-end container. Similarly, the back-end container needs to retrieve the IP address of the database container. Additionally, the IP address could change if a container restarts, so a process is needed to ensure any change in IP triggers an update to existing containers.

Both Kubernetes and OpenShift provide potential solutions to the issue of service discoverability and the dynamic nature of container networking. Some of these solutions are covered later in the chapter.

Rootless containers

Rootless containers do not support software-defined network (SDN). Therefore, the container's IP address is not available to communicate with other containers on the host. We can achieve networking between rootless containers by using port-forwarding. Port-forwarding allows external access to a container service from the host. Port-forwarding has been discussed in [Chapter 3, Managing Containers](#).

Figure 7.2: Multi-Container Applications for rootless containers

For example, consider an application that has both a front-end container and a back-end container. The back-end container has a service running on specific port 80. This service/port is not accessible from outside of the container. The front-end container needs to access the service/port from the back-end container. The first step is to use *port-forwarding* for the required service. In this example, we have forwarded service port 80 to 8080. Now the front-end container can access the service/port of the back-end container by using the host IP address and forwarded port.

Describing the To Do List Application

Many labs from this course make use of a `To Do List` application. This application is divided into three tiers, as illustrated by the following figure:

Figure 7.3: To Do List application logical architecture

- The presentation tier is built as a single-page HTML5 front-end using AngularJS.
- The business tier is composed by an HTTP API back-end, with Node.js.
- The persistence tier based on a MySQL database server.

The following figure is a screen capture of the application web interface:

The screenshot shows a web application interface for a 'To Do List Application'. On the left, there is a table titled 'To Do List' with columns: 'Id', 'Description', 'Done', and a small red 'X' icon. Two items are listed: '1 Pick up new...' (Done: false) and '2 Buy groceries' (Done: true). Below the table are navigation buttons: 'First', 'Previous', '1' (highlighted in blue), 'Next', and 'Last'. On the right, there is a form titled 'Add Task' with fields for 'Description' (a text input field) and 'Completed' (a checkbox). Below the form are 'Clear' and 'Save' buttons. The overall layout is clean and modern, using a light gray background and blue highlights for active elements.

Figure 7.4: The To Do List application

On the left is a table with items to complete, and on the right is a form to add a new item.

The classroom private registry server, `services.lab.example.com`, provides the application in two versions:

nodejs

Represents the way a typical developer would create the application as a single unit, without caring to break it into tiers or services.

nodejs_api

Shows the changes needed to break the application into presentation and business tiers. Each tier corresponds to an isolated container image

The sources of both of these application versions are available from the todoapp/nodejs folder in the Git repository at: <https://github.com/RedHatTraining/DO180-apps.git>.

[Next](#)

Guided Exercise: Deploying the Web Application and MySQL on Linux Containers

In this lab, you will create a script that runs and networks a Node.js application container and the MySQL container.

Outcomes

You should be able to network containers to create a multi-tiered application.

Make sure you have completed [the section called “Guided Exercise: Configuring the Classroom Environment”](#) from *Chapter 1* before executing any command of this practice.

You must have the To Do List application source code and lab files on workstation. To set up the environment for the exercise, run the following command:

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

Procedure 7.1. Instructions

1. Log in to the Red Hat Container Catalog with your Red Hat account. If you need to register to Red Hat, see the instructions in [Appendix D, Creating a Red Hat Account](#).
2. [student@workstation ~]\$ podman login registry.redhat.io
3. Username: *your_username*
4. Password: *your_password*

Login Succeeded!

5. Review the Containerfile.

Using your preferred editor, open and examine the completed Containerfile located at ~/D0180/labs/multicontainer-design/deploy/nodejs/Containerfile.

6. Run ip addr command to grep host IP address.

7. [student@workstation ~]\$ ip -br addr list eth0

eth0	UP	172.25.250.9/24 fe80::d1cf:b1dc:ddc8:b79b/64
------	----	--

8. Explore the Environment Variables.

Inspect the environment variables that allow the Node.js REST API container to communicate with the MySQL container.

1. View the file ~/D0180/labs/multicontainer-design/deploy/nodejs/nodejs-source/models/db.js, containing the database configuration provided below:

```
2. module.exports.params = {  
3.   dbname: process.env.MYSQL_DATABASE,  
4.   username: process.env.MYSQL_USER,  
5.   password: process.env.MYSQL_PASSWORD,  
6.   params: {  
7.     host: '172.25.250.9',  
8.     port: '30306',  
9.     dialect: 'mysql'  
10.    }  
};
```

11. Notice the environment variables used by the REST API. These variables are exposed to the container using -e options with the podman run command in this guided exercise. Those environment variables are described below.

MYSQL_DATABASE

The name of the MySQL database in the `mysql` container.

MYSQL_USER

The name of the database user used by the `todoapi` container to run MySQL commands.

MYSQL_PASSWORD

The password of the database user that the `todoapi` container uses to authenticate to the `mysql` container.

Note

The host and port details of the MySQL container are embedded with the REST API application. The host, as shown above in the `db.js` file, is the IP address of host, which we have grep in previous step.

9. Build the To Do List application child image using the provided Containerfile.
 1. Build the child image.

Examine the `~/D0180/labs/multicontainer-design/deploy/nodejs` design/`deploy/nodejs/build.sh` script to see how the image is built. Run the following commands to build the child image.

```
[student@workstation nodejs]$ cd ~/D0180/labs/multicontainer-design/deploy/nodejs
[student@workstation nodejs]$ ./build.sh
Preparing build folder
Preparing build folder
STEP 1: FROM registry.redhat.io/rhel8/nodejs-12
Getting image source signatures
Copying blob 666be21779ed done
...output omitted...
Writing manifest to image destination
Storing signatures
STEP 2: ARG NEXUS_BASE_URL
STEP 3: MAINTAINER username <username@example.com>
STEP 4: COPY run.sh build ${HOME}/
```

```
STEP 5: RUN npm install --registry=http://$NEXUS_BASE_URL/repository/nodejs/
s/
...output omitted...
Writing manifest to image destination
Storing signatures
e4901...d37eb
```

Note

The build.sh script lowers restrictions for write access to the build directory, allowing non-root users to install dependencies.

Sometimes network instability causes an ERR! with the build script. To resolve this issue, re-run the build script.

2. Wait for the build to complete and then run the following command to verify that the image has been built successfully:

```
3. [student@workstation nodejs]$ podman images \
4. > --format "table {{.ID}} {{.Repository}} {{.Tag}}"
5. IMAGE ID          REPOSITORY                      TAG
6. e4901b30413b    localhost/do180/todonodejs      latest
```

```
6b949cc5e908    registry.redhat.io/rhel8/nodejs-12   1
```

10. Modify the existing script to create containers with the appropriate ports, as defined in the previous step. In this script, the order of commands is given such that it starts the mysql container and then starts the todoapi container before connecting it to the mysql container. After invoking every container, there is a wait time of 9 seconds, so each container has time to start.

1. Edit the run.sh file located at ~/D0180/labs/multicontainer-design/deploy/nodejs/networked to insert the podman run command at the appropriate line for invoking mysql container. The following screen shows the exact podman command to insert into the file.

```
2. podman run -d --name mysql -e MYSQL_DATABASE=items -e MYSQL_USER=user1 \
3. -e MYSQL_PASSWORD=mypa55 -e MYSQL_ROOT_PASSWORD=r00tpa55 \
4. -v $PWD/work/data:/var/lib/mysql/data \
5. -p 30306:3306 \
```

```
registry.redhat.io/rhel8/mysql-80:1
```

In the previous command, the `MYSQL_DATABASE`, `MYSQL_USER`, and `MYSQL_PASSWORD` are populated with the credentials to access the MySQL database. These environment variables are required for the `mysql` container to run. Also, the `$PWD/work/data` local folder are mounted as volume into the container's file system.

6. In the same `run.sh` file, insert another `podman run` command at the appropriate line to run the `todoapi` container. The following screen shows the `podman` command to insert into the file.

```
7. podman run -d --name todoapi -e MYSQL_DATABASE=items -e MYSQL_USER=user1 \
8. -e MYSQL_PASSWORD=mypa55 \
9. -p 30080:30080 \
```

```
do180/todonodejs
```

Note

After each `podman run` command inserted into the `run.sh` script, ensure that there is also a `sleep 9` command. If you need to repeat this step, the `work` directory and its contents must be deleted before re-running the `run.sh` script.

10. Verify that your `run.sh` script matches the solution script located at `~/D0180/solutions/multicontainer-design/deploy/nodejs/networked/run.sh`.
11. Save the file and exit the editor.
11. Run the containers.
 1. Use the following command to execute the script that you updated to run the `mysql` and `todoapi` containers.

```
2. [student@workstation nodejs]$ cd \
3. > ~/D0180/labs/multicontainer-design/deploy/nodejs/networked
```

```
[student@workstation networked]$ ./run.sh
```

4. Verify that the containers started successfully.

```
5. [student@workstation networked]$ podman ps \
```

```
6. > --format="table {{.ID}} {{.Names}} {{.Image}} {{.Status}}"
7. ID          Names      Image                  Status
8. c74b4709e3ae todoapi    localhost/do180/todonodejs:latest   Up 3 minutes ago
```

```
3bc19f74254c mysql      registry.redhat.io/rhel8/mysql-80:1 Up 3 minutes ago
```

12. Populate the items database with the Projects table.

```
13. [student@workstation networked]$ mysql -uuser1 -h 172.25.250.9 \
14. > -pmypa55 -P30306 items < \
15. > ~/D0180/labs/multicontainer-design/deploy/nodejs/networked/db.sql
```

```
mysql: [Warning] Using a password on the command line interface can be insecure.
```

16. Examine the environment variables of the API container.

Run the following command to explore the environment variables exposed in the API container.

```
[student@workstation networked]$ podman exec -it todoapi env
...output omitted...
HOME=/opt/app-root/src
MYSQL_DATABASE=items
MYSQL_USER=user1
MYSQL_PASSWORD=mypa55
APP_ROOT=/opt/app-root
```

17. Test the application.

1. Run a curl command to test the REST API for the To Do List application.

```
2. [student@workstation networked]$ curl -w "\n" \
3. > http://127.0.0.1:30080/todo/api/items/1
```

```
{"id":1,"description":"Pick up newspaper","done":false}
```

The -w "\n" option with curl command lets the shell prompt appear at the next line rather than merging with the output in the same line.

4. Open Firefox on workstation and point your browser to `http://127.0.0.1:30080/todo/`. You should see the To Do List application.

Note

Make sure to append the trailing slash (/).

5. Change to the `/home/student` directory.

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

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

Finish

On workstation, run the `lab multicontainer-design finish` script to complete this exercise.

```
[student@workstation ~]$ lab multicontainer-design finish
```

This concludes the guided exercise.

[Previous](#) [Next](#)

Deploying a Multi-Container Application on OpenShift

Objectives

After completing this section, students should be able to:

- Describe differences between Podman and Kubernetes
- Deploy a multicontainer application on OpenShift.

Comparing Podman and Kubernetes

Using environment variables allows you to share information between containers with Podman. However, there are still some limitations and some manual work involved in ensuring that all environment variables stay in sync, especially when

working with many containers. Kubernetes provides an approach to solve this problem by creating services for your containers, as covered in previous chapters.

Services in Kubernetes

Pods are attached to a Kubernetes namespace, which OpenShift calls a *project*. When a pod starts, Kubernetes automatically adds a set of environment variables for each service defined on the same namespace.

Any service defined on Kubernetes generates environment variables for the IP address and port number where the service is available. Kubernetes automatically injects these environment variables into the containers from pods in the same namespace. These environment variables usually follow a convention:

Uppercase

All environment variables are set using uppercase names.

Snakecase

Any environment variable created by a service is usually composed of multiple words separated with an underscore (_).

Service name first

The first word for an environment variable created by a service is the service name.

Protocol type

Most network environment variables include the protocol type (TCP or UDP).

These are the environment variables generated by Kubernetes for a service.

<SERVICE_NAME>_SERVICE_HOST

Represents the IP address enabled by a service to access a pod.

<SERVICE_NAME>_SERVICE_PORT

Represents the port where the server port is listed.

<SERVICE_NAME>_PORT

Represents the address, port, and protocol provided by the service for external access.

<SERVICE_NAME>_PORT_<PORT_NUMBER>_<PROTOCOL>

Defines an alias for the <SERVICE_NAME>_PORT.

<SERVICE_NAME>_PORT_<PORT_NUMBER>_<PROTOCOL>_PROTO
Identifies the protocol type (TCP or UDP).

<SERVICE_NAME>_PORT_<PORT_NUMBER>_<PROTOCOL>_PORT
Defines an alias for <SERVICE_NAME>_SERVICE_PORT.

<SERVICE_NAME>_PORT_<PORT_NUMBER>_<PROTOCOL>_ADDR
Defines an alias for <SERVICE_NAME>_SERVICE_HOST.

For instance, given the following service:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    name: mysql
    name: mysql
spec:
  ports:
    - protocol: TCP
      port: 3306
  selector:
    name: mysql
```

The following environment variables are available for each pod created after the service, on the same namespace:

```
MYSQL_SERVICE_HOST=10.0.0.11
MYSQL_SERVICE_PORT=3306
MYSQL_PORT=tcp://10.0.0.11:3306
MYSQL_PORT_3306_TCP=tcp://10.0.0.11:3306
MYSQL_PORT_3306_TCP_PROTO=tcp
MYSQL_PORT_3306_TCP_PORT=3306
MYSQL_PORT_3306_TCP_ADDR=10.0.0.11
```

Note

Other relevant <SERVICE_NAME>_PORT_* environment variable names are set on the basis of the protocol. IP address and port number are set in the <SERVICE_NAME>_PORT environment variable.

For example, `MYSQL_PORT=tcp://10.0.0.11:3306` entry leads to the creation of environment variables with names such as `MYSQL_PORT_3306_TCP`, `MYSQL_PORT_3306_TCP_PROTO`, `MYSQL_PORT_3306_TCP_PORT`, and `MYSQL_PORT_3306_TCP_ADDR`. If the protocol component of an environment variable is undefined, Kubernetes uses the TCP protocol and assigns the variable names accordingly.

[Previous](#) [Next](#)

Guided Exercise: Creating an Application on OpenShift

In this exercise, you will deploy the To Do List application in OpenShift Container Platform.

Outcomes

You should be able to build and deploy an application in OpenShift Container Platform.

Make sure you have completed [the section called “Guided Exercise: Configuring the Classroom Environment”](#) from *Chapter 1* before executing any command of this practice.

You must have the To Do List application source code and lab files on workstation. To download the lab files and verify the status of the OpenShift cluster, run the following command in a new terminal window.

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

Procedure 7.2. Instructions

1. Create the To Do List application from the provided YAML file.
 1. Log in to OpenShift Container Platform.

```
2. [student@workstation ~]$ source /usr/local/etc/ocp4.config  
3.
```

```
4. [student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
5. > -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
6. Login successful.
7.
8. ...output omitted...
9.
```

Using project "default".

Note

If the `oc login` command prompts about using insecure connections, answer `y` (yes).

10. Create a new project *application* in OpenShift to use for this exercise.
Run the following command to create the `application` project.

```
11. [student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-application
```

Now using project ...*output omitted...*

12. Review the YAML file.

Using your preferred editor, open and examine the `app` file located at `~/D0180/labs/multicontainer-application/todo-app.yaml`. Notice the following resources defined in the `todo-app.yaml` and review their configurations.

- The `todoapi` pod definition defines the Node.js application.
- The `mysql` pod definition defines the MySQL database.
- The `todoapi` service provides connectivity to the Node.js application pod.
- The `mysql` service provides connectivity to the MySQL database pod.
- The `dbclaim` persistent volume claim definition defines the MySQL `/var/lib/mysql/data` volume.

13. Create application resources with given yaml file.

Use the `oc create` command to create the application resources. Run the following command in the terminal window:

```
[student@workstation ~]$ cd ~/D0180/labs/multicontainer-application  
[student@workstation multicontainer-application]$ oc create -f todo-app.yaml  
pod/mysql created  
pod/todoapi created  
service/todoapi created  
service/mysql created  
persistentvolumeclaim/dbclaim created
```

14. Review the status of the deployment using the `oc get pods` command with the `-w` option to continue to monitor the pod status. Wait until both the containers are running. It may take some time for both pods to start.

```
15. [student@workstation multicontainer-application]$ oc get pods -w
```

NAME	READY	STATUS	RESTARTS	AGE
todoapi	1/1	Running	0	27s

mysql	1/1	Running	0	27s
-------	-----	---------	---	-----

Press **Ctrl+C** to exit the command.

2. Connect to the MySQL database server and populate the data to the `item` database.

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.

```
2. [student@workstation multicontainer-application]$ oc port-forward mysql 3306:3306  
3. Forwarding from 127.0.0.1:3306 -> 3306
```

```
Forwarding from [::1]:3306 -> 3306
```

4. From the workstation machine open another terminal and populate the data to the MySQL server using the MySQL client.

```
5. [student@workstation ~]$ cd ~/D0180/labs/multicontainer-application
```

```
6. [student@workstation multicontainer-application]$ mysql -uuser1 \
7. > -h 127.0.0.1 -pmypa55 -P3306 items < db.sql
```

```
mysql: [Warning] Using a password on the command line interface can be ins
ecure.
```

8. Close the terminal and return to the previous one. Finish the port forwarding process by pressing **Ctrl+C**.

```
9. Forwarding from 127.0.0.1:3306 -> 3306
10. Forwarding from [::1]:3306 -> 3306
11. Handling connection for 3306
```

```
^C
```

3. Expose the Service.

To allow the To Do List application to be accessible through the OpenShift router and to be available as a public FQDN, use the `oc expose` command to expose the `todoapi` Service.

Run the following command in the terminal window.

```
[student@workstation multicontainer-application]$ oc expose service todoapi
route.route.openshift.io/todoapi exposed
```

4. Test the application.

1. Find the FQDN of the application by running the `oc status` command and note the FQDN for the app.

Run the following command in the terminal window.

```
[student@workstation multicontainer-application]$ oc status | grep -o "htt
p:.*com"
http://todoapi-${RHT_OCP4_DEV_USER}-application.${RHT_OCP4_WILDCARD_DOMAIN
}
```

2. Use `curl` to test the REST API for the To Do List application.

```
3. [student@workstation multicontainer-application]$ curl -w "\n" \
4. > $(oc status | grep -o "http:.*com")/todo/api/items/1
```

```
{"id":1,"description":"Pick up newspaper","done":false}
```

Note

The `-w "\n"` option with `curl` command lets the shell prompt appear at the next line rather than merging with the output in the same line.

5. Change to the `/home/student` directory.

```
6. [student@workstation multicontainer-application]$ cd ~
```

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

7. Open Firefox on workstation and point your browser to the To Do List application URL.

- `http://todoapi-${RHT_OCP4_DEV_USER}-application.${RHT_OCP4_WILDCARD_DOMAIN}/todo/`

Important

The trailing slash in the URL mentioned above is necessary. If you do not include that in the URL, you may encounter issues with the application.

To Do List Application

To Do List

ID	Description	Done	
1	Pick up new...	false	X
2	Buy groceries	true	X

First Previous **1** Next Last

Add Task

Description:

Add Description.

Completed:

Clear Save

Figure 7.5: To Do List application

Finish

On workstation, run the `lab multicontainer-application finish` script to complete this lab.

```
[student@workstation ~]$ lab multicontainer-application finish
```

This concludes the guided exercise.

[Previous](#) [Next](#)

Deploying a Multi-container Application on OpenShift Using a Template

Objectives

After completing this section, students should be able to deploy a multi-container application on OpenShift using a template.

Examining the Skeleton of a Template

Deploying an application on OpenShift Container Platform often requires creating several related resources within a Project. For example, a web application may require a `BuildConfig`, `Deployment`, `Service`, and `Route` resource to run in an OpenShift project. Often the attributes of these resources have the same value, such as a resource's name attribute.

OpenShift templates provide a way to simplify the creation of resources that an application requires. A template defines a set of related resources to be created together, as well as a set of application parameters. The attributes of template resources are typically defined in terms of the template parameters, such as a resource's name attribute.

For example, an application might consist of a front-end web application and a database server. Each consists of a service resource and a deployment resource. They share a set of credentials (parameters) for the front end to authenticate to the back end. The template can be processed by specifying parameters or by allowing them to be automatically generated (for example, for a unique database password) in order to instantiate the list of resources in the template as a cohesive application.

The OpenShift installer creates several templates by default in the `openshift` namespace. Run the `oc get templates` command with the `-n openshift` option to list these preinstalled templates:

```
[user@host ~]$ oc get templates -n openshift
NAME                  DESCRIPTION
cakephp-mysql-example  An example CakePHP application ...
cakephp-mysql-persistent  An example CakePHP application ...
dancer-mysql-example    An example Dancer application with a MySQL ...
dancer-mysql-persistent  An example Dancer application with a MySQL ...
django-psql-example     An example Django application with a PostgreSQL ...
...output omitted...
rails-pgsql-persistent   An example Rails application with a PostgreSQL ...
```

```
rails-postgresql-example An example Rails application with a PostgreSQL ...
redis-ephemeral           Redis in-memory data structure store, ...
redis-persistent           Redis in-memory data structure store, ...
```

The YAML definition of a template can be modified to suit the needs of your applications. You will do this in an upcoming lab.

The following shows a YAML template definition:

```
[user@host ~]$ oc get template mysql-persistent -n openshift -o yaml
apiVersion: template.openshift.io/v1
kind: Template
labels: ...value omitted...
message: ...message omitted ...
metadata:
  annotations:
    description: ...description omitted...
    iconClass: icon-mysql-database
    openshift.io/display-name: MySQL
    openshift.io/documentation-url: ...value omitted...
    openshift.io/long-description: ...value omitted...
    openshift.io/provider-display-name: Red Hat, Inc.
    openshift.io/support-url: https://access.redhat.com

  tags: database,mysql
  labels: ...value omitted...

  name: mysql-persistent

objects:
- apiVersion: v1
  kind: Secret
  metadata:
    annotations: ...annotations omitted...

  name: ${DATABASE_SERVICE_NAME}
```

```
stringData: ...stringData omitted...

- apiVersion: v1
  kind: Service
  metadata:
    annotations: ...annotations omitted...
    name: ${DATABASE_SERVICE_NAME}
  spec: ...spec omitted...
- apiVersion: v1
  kind: PersistentVolumeClaim
  metadata:
    name: ${DATABASE_SERVICE_NAME}
  spec: ...spec omitted...
- apiVersion: v1
  kind: Deployment
  metadata:
    annotations: ...output omitted...
    name: ${DATABASE_SERVICE_NAME}
  spec: ...output omitted...

parameters:
- ...MEMORY_LIMIT parameter omitted...
- ...NAMESPACE parameter omitted...
- description: The name of the OpenShift Service exposed for the database.
  displayName: Database Service Name

  name: DATABASE_SERVICE_NAME
  required: true
  value: mysql
- ...MYSQL_USER parameter omitted...
- description: Password for the MySQL connection user.
  displayName: MySQL Connection Password

  from: '[a-zA-Z0-9]{16}'
  generate: expression
```

```
name: MYSQL_PASSWORD
required: true
- ...MYSQL_ROOT_PASSWORD parameter omitted...
- ...MYSQL_DATABASE parameter omitted...
- ...VOLUME_CAPACITY parameter omitted...
- ...MYSQL_VERSION parameter omitted...
```

Defines a list of arbitrary tags to associate with this template. Enter any of these tags in the UI to find this template.

Defines the template name.

The `objects` section defines the list of OpenShift resources for this template. This template creates four resources: a `Secret`, a `Service`, a `PersistentVolumeClaim`, and a `Deployment`. All four resource objects have their names set to the value of the `DATABASE_SERVICE_NAME` parameter.

The `parameters` section contains a list of nine parameters.

Template resources often define their attributes using the values of these parameters, as demonstrated with the `DATABASE_SERVICE_NAME` parameter.

If you do not specify a value for the `MYSQL_PASSWORD` parameter when you create an application with this template, OpenShift generates a password that matches this regular expression.

You can publish a new template to the OpenShift cluster so that other developers can build an application from the template.

Assume you have an task list application named `todo` that requires an OpenShift Deployment, Service, and Route object for deployment. You create a YAML template definition file that defines attributes for these OpenShift resources, along with definitions for any required parameters. Assuming the template is defined in the `todo-template.yaml` file, use the `oc create` command to publish the application template:

```
[user@host deploy-multicontainer]$ oc create -f todo-template.yaml
template.template.openshift.io/todonodejs-persistent created
```

By default, the template is created under the current project unless you specify a different one using the `-n` option, as shown in the following example:

```
[user@host deploy-multicontainer]$ oc create -f todo-template.yaml \
> -n openshift
```

Important

Any template created under the `openshift` namespace (OpenShift project) is available in the web console under the **Developer Catalog** of the **Developer** perspective. Moreover, any template created under the current project is accessible from that project.

Parameters

Templates define a set of parameters, which are assigned values. OpenShift resources defined in the template can get their configuration values by referencing *named parameters*. Parameters in a template can have default values, but they are optional. Any default value can be replaced when processing the template.

Each parameter value can be set either explicitly by using the `oc process` command, or generated by OpenShift according to the parameter configuration.

There are two ways to list available parameters from a template. The first one is using the `oc describe` command:

```
[user@host ~]$ oc describe template mysql-persistent -n openshift
Name:    mysql-persistent
Namespace:  openshift
Created:  12 days ago
Labels:   samplesoperator.config.openshift.io/managed=true
Description: MySQL database service, with ...description omitted...
Annotations:  iconClass=icon-mysql-database
               openshift.io/display-name=MySQL
               ...output omitted...
tags=database,mysql
```

Parameters:

```
Name:    MEMORY_LIMIT
Display Name: Memory Limit
Description: Maximum amount of memory the container can use.
Required:  true
Value:   512Mi
```

```
Name:    NAMESPACE
Display Name: Namespace
Description: The OpenShift Namespace where the ImageStream resides.
Required:  false
Value:    openshift
```

...output omitted...

```
Name:    MYSQL_VERSION
Display Name: Version of MySQL Image
Description: Version of MySQL image to be used (8.0, or latest).
Required:  true
Value:    8.0
```

Object Labels: template=mysql-persistent-template

Message: *...output omitted...* in your project: \${DATABASE_SERVICE_NAME}.

```
Username: ${MYSQL_USER}
Password: ${MYSQL_PASSWORD}
Database Name: ${MYSQL_DATABASE}
Connection URL: mysql://${DATABASE_SERVICE_NAME}:3306/
```

For more information about using this template, *...output omitted...*

Objects:

```
Secret      ${DATABASE_SERVICE_NAME}
Service     ${DATABASE_SERVICE_NAME}
PersistentVolumeClaim ${DATABASE_SERVICE_NAME}
Deployment   ${DATABASE_SERVICE_NAME}
```

The second way is by using the oc process with the --parameters option:

NAME	DESCRIPTION	GENERATOR	VALUE
MEMORY_LIMIT	Maximum amount of memory to allocate to the application container.		512Mi
NAMESPACE	The OpenShift namespace where the application will be created.		openshift
DATABASE_SERVICE_NAME	The name of the MySQL service to be created.		mysql
MYSQL_USER	Username for the MySQL database.	expression	user[A-Z0-9]{3}
MYSQL_PASSWORD	Password for the MySQL database.	expression	[a-zA-Z0-9]{16}
MYSQL_ROOT_PASSWORD	Password for the MySQL root user.	expression	[a-zA-Z0-9]{16}
MYSQL_DATABASE	Name of the MySQL database to be created.		sampledb
VOLUME_CAPACITY	Volume size for the MySQL persistent volume.		1Gi
MYSQL_VERSION	Version of MySQL to be installed.		8.0

Processing a Template Using the CLI

When you process a template, you generate a list of resources to create a new application. To process a template, use the `oc process` command:

```
[user@host ~]$ oc process -f <filename>
```

The previous command processes a template file, in either JSON or YAML format, and returns the list of resources to standard output. The format of the output resource list is JSON. To output the resource list in YAML format, use the `-o yaml` with the `oc process` command:

```
[user@host ~]$ oc process -o yaml -f <filename>
```

Another option is to process a template from the current project or the `openshift` project:

```
[user@host ~]$ oc process <uploaded-template-name>
```

Note

The `oc process` command returns a list of resources to standard output. This output can be redirected to a file:

```
[user@host ~]$ oc process -o yaml -f filename > myapp.yaml
```

Templates often generate resources with configurable attributes that are based on the template parameters. To override a parameter, use the `-p` option followed by a `<name>=<value>` pair.

```
[user@host ~]$ oc process -o yaml -f mysql.yaml \
> -p MYSQL_USER=dev -p MYSQL_PASSWORD=$P4SSD -p MYSQL_DATABASE=bank \
> -p VOLUME_CAPACITY=10Gi > mysqlProcessed.yaml
```

To create the application, use the generated YAML resource definition file:

```
[user@host ~]$ oc create -f mysqlProcessed.yaml
```

Alternatively, it is possible to process the template and create the application without saving a resource definition file by using a UNIX pipe:

```
[user@host ~]$ oc process -f mysql.yaml -p MYSQL_USER=dev \
> -p MYSQL_PASSWORD=$P4SSD -p MYSQL_DATABASE=bank \
> -p VOLUME_CAPACITY=10Gi | oc create -f -
```

To use a template in the openshift project to create an application in your project, first export the template:

```
[user@host ~]$ oc get template mysql-persistent -o yaml \
> -n openshift > mysql-persistent-template.yaml
```

Next, identify appropriate values for the template parameters and process the template:

```
[user@host ~]$ oc process -f mysql-persistent-template.yaml \
> -p MYSQL_USER=dev -p MYSQL_PASSWORD=$P4SSD -p MYSQL_DATABASE=bank \
> -p VOLUME_CAPACITY=10Gi | oc create -f -
```

You can also use two slashes (//) to provide the namespace as part of the template name:

```
[user@host ~]$ oc process openshift//mysql-persistent \
> -p MYSQL_USER=dev -p MYSQL_PASSWORD=$P4SSD -p MYSQL_DATABASE=bank \
```

```
> -p VOLUME_CAPACITY=10Gi | oc create -f -
```

Alternatively, it is possible to create an application using the `oc new-app` command passing the template name as the `--template` option argument:

```
[user@host ~]$ oc new-app --template=mysql-persistent \
> -p MYSQL_USER=dev -p MYSQL_PASSWORD=$P4SSD -p MYSQL_DATABASE=bank \
> -p VOLUME_CAPACITY=10Gi
```

References

Developer information about templates can be found in the *Using Templates* section of the OpenShift Container Platform documentation: [Developer Guide](#)

[Previous](#) [Next](#)

Guided Exercise: Creating an Application with a Template

In this exercise, you will deploy the `To Do List` application in OpenShift Container Platform using a template to define resources your application needs to run.

Outcomes

You should be able to build and deploy an application in OpenShift Container Platform using a provided JSON template.

Make sure you have completed [the section called “Guided Exercise: Configuring the Classroom Environment”](#) from *Chapter 1* before executing any command of this practice.

You must have the `To Do List` application source code and lab files on workstation. To download the lab files and verify the status of the OpenShift cluster, run the following command in a new terminal window.

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

Procedure 7.3. Instructions

1. Create the To Do List application from the provided JSON template.
 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. Log in to OpenShift Container Platform. If the `oc login` command prompts about using insecure connections, answer `y` (yes).

3. [student@workstation ~]\$ `oc login -u ${RHT_OCP4_DEV_USER} \`
4. > `-p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}`
5. Login successful.
- 6.
7. ...output omitted...
- 8.

```
Using project "default".
```

9. Create a new project *template* in OpenShift to use for this exercise. Run the following command to create the template project.

```
10. [student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-template
```

```
Now using project ...output omitted...
```

11. Review the template.

Using your preferred editor, open and examine the template located at `~/D0180/labs/multicontainer-openshift/todo-template.json`. Notice the following resources defined in the template and review their configurations.

- The `todoapi` pod definition defines the Node.js application.
- The `mysql` pod definition defines the MySQL database.

- The `todoapi` service provides connectivity to the Node.js application pod.
- The `mysql` service provides connectivity to the MySQL database pod.
- The `dbclaim` persistent volume claim definition defines the MySQL `/var/lib/mysql/data` volume.

12. Process the template and create the application resources.

Use the `oc process` command to process the template file. This template requires the Quay.io namespace to retrieve the container images. Use the `pipe` command to send the result to the `oc create` command.

Run the following command in the terminal window:

```
[student@workstation ~]$ cd ~/D0180/labs/multicontainer-openshift
[student@workstation multicontainer-openshift]$ oc process \
> -f todo-template.json | oc create -f -
pod/mysql created
pod/todoapi created
service/todoapi created
service/mysql created
persistentvolumeclaim/dbclaim created
```

13. Review the status of the deployment using the `oc get pods` command with the `-w` option to continue to monitor the pod status. Wait until both the containers are running. It may take some time for both pods to start.

14. [student@workstation multicontainer-openshift]\$ `oc get pods -w`

NAME	READY	STATUS	RESTARTS	AGE
todoapi	1/1	Running	0	27s

mysql	1/1	Running	0	27s
-------	-----	---------	---	-----

Press **Ctrl+C** to exit the command.

2. Connect to the MySQL database server and populate the data to the `item` database.

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.

```
2. [student@workstation multicontainer-openshift]$ oc port-forward mysql 3306 :3306  
3. Forwarding from 127.0.0.1:3306 -> 3306
```

```
Forwarding from [::1]:3306 -> 3306
```

4. From the workstation machine open another terminal and populate the data to the MySQL server using the MySQL client.

```
5. [student@workstation ~]$ cd ~/D0180/labs/multicontainer-openshift  
6. [student@workstation multicontainer-openshift]$ mysql -uuser1 \  
7. > -h 127.0.0.1 -pmypa55 -P3306 items < db.sql
```

```
mysql: [Warning] Using a password on the command line interface can be insecure.
```

8. Close the terminal and return to the previous one. Finish the port forwarding process by pressing **Ctrl+C**.

```
9. Forwarding from 127.0.0.1:3306 -> 3306  
10. Forwarding from [::1]:3306 -> 3306  
11. Handling connection for 3306
```

```
^C
```

3. Expose the Service.

To allow the To Do List application to be accessible through the OpenShift router and to be available as a public FQDN, use the `oc expose` command to expose the `todoapi` service.

Run the following command in the terminal window.

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

4. Test the application.

1. Find the FQDN of the application by running the `oc status` command and note the FQDN for the app.

Run the following command in the terminal window.

```
[student@workstation multicontainer-openshift]$ oc status | grep -o "http:.*com"  
http://todoapi-${RHT_OCP4_DEV_USER}-template.${RHT_OCP4_WILDCARD_DOMAIN}
```

2. Use `curl` to test the REST API for the To Do List application.

```
3. [student@workstation multicontainer-openshift]$ curl -w "\n" \  
4. > $(oc status | grep -o "http:.*com")/todo/api/items/1
```

```
{"id":1,"description":"Pick up newspaper","done":false}
```

Note

The `-w "\n"` option with `curl` command lets the shell prompt appear at the next line rather than merging with the output in the same line.

5. Change to the `/home/student` directory.

```
6. [student@workstation multicontainer-openshift]$ cd ~
```

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

7. Open Firefox on workstation and point your browser to the application URL and you should see the To Do List application.

- `http://todoapi-${RHT_OCP4_DEV_USER}-template.${RHT_OCP4_WILDCARD_DOMAIN}/todo/`

Important

The trailing slash in the URL mentioned above is necessary. If you do not include that in the URL, you may encounter issues with the application.

To Do List Application

To Do List

ID	Description	Done	
1	Pick up new...	false	X
2	Buy groceries	true	X

First Previous **1** Next Last

Add Task

Description:

Add Description.

Completed:

Clear Save

Figure 7.6: To Do List application

Finish

On workstation, run the following script to complete this lab.

```
[student@workstation ~]$ lab multicontainer-openshift finish
```

This concludes the guided exercise.

[Previous](#) [Next](#)

Summary

In this chapter, you learned:

- Software-defined networks enable communication between containers. Containers must be attached to the same software-defined network to communicate.

- Containerized applications cannot rely on fixed IP addresses or host names to find services.
- Podman uses Container Network Interface (CNI) to create a software-defined network and attaches all containers on the host to that network. Kubernetes and OpenShift create a software-defined network between all containers in a pod.
- Within the same project, Kubernetes injects a set of variables for each service into all pods.
- OpenShift templates automate creating applications consisting of multiple resources. Template parameters allow using the same values when creating multiple resources.

[Previous](#) [Next](#)

Summary

In this chapter, you learned:

- Software-defined networks enable communication between containers. Containers must be attached to the same software-defined network to communicate.
- Containerized applications cannot rely on fixed IP addresses or host names to find services.
- Podman uses Container Network Interface (CNI) to create a software-defined network and attaches all containers on the host to that network. Kubernetes and OpenShift create a software-defined network between all containers in a pod.
- Within the same project, Kubernetes injects a set of variables for each service into all pods.
- OpenShift templates automate creating applications consisting of multiple resources. Template parameters allow using the same values when creating multiple resources.

[Previous](#) [Next](#)

Chapter 8. Troubleshooting Containerized Applications

[**Troubleshooting S2I Builds and Deployments**](#)

[**Guided Exercise: Troubleshooting an OpenShift Build**](#)

[**Troubleshooting Containerized Applications**](#)

[**Guided Exercise: Configuring Apache Container Logs for Debugging**](#)

[**Lab: Troubleshooting Containerized Applications**](#)

[**Summary**](#)

Abstract

Goal	Troubleshoot a containerized application deployed on OpenShift.
Objectives	<ul style="list-style-type: none">• Troubleshoot an application build and deployment on OpenShift.• Implement techniques for troubleshooting and debugging containerized applications.
Sections	<ul style="list-style-type: none">• Troubleshooting S2I Builds and Deployments (and Guided Exercise)• Troubleshooting Containerized Applications (and Guided Exercise)
Lab	<ul style="list-style-type: none">• Troubleshooting Containerized Applications

Troubleshooting S2I Builds and Deployments

Objectives

After completing this section, you should be able to:

- Troubleshoot an application build and deployment steps on OpenShift.
- Analyze OpenShift logs to identify problems during the build and deploy process.

Introduction to the S2I Process

The Source-to-Image (S2I) process is a simple way to automatically create images based on the programming language of the application source code in OpenShift. While this process is often a convenient way to quickly deploy applications, problems can arise during the S2I image creation process, either by the programming language characteristics or the runtime environment that require both developers and administrators to work together.

It is important to understand the basic workflow for most of the programming languages supported by OpenShift. The S2I image creation process is composed of two major steps:

- Build step: Responsible for compiling source code, downloading library dependencies, and packaging the application as a container image. Furthermore, the build step pushes the image to the OpenShift registry for the deployment step. The `BuildConfig` (BC) OpenShift resources drive the build step.
- Deployment step: Responsible for starting a pod and making the application available for OpenShift. This step executes after the build step, but only if the build step succeeded. The `Deployment` OpenShift resources drive the deployment step.

For the S2I process, each application uses its own `BuildConfig` and `Deployment` objects, the name of which matches the application name. The deployment process aborts if the build fails.

The S2I process starts each step in a separate pod. The build process creates a pod named `<application-name>-build-<number>-<string>`. For each build attempt, the entire build step executes and saves a log. Upon a successful build, the application starts on a separate pod named as `<application-name>-<string>`.

The OpenShift web console can be used to access the details for each step. To identify any build issues, the logs for a build can be evaluated and analyzed by clicking the **Builds** link from the left panel, depicted as follows.

Figure 8.1: Build instances of a project

For each build attempt, a history of the build, tagged with a number, is provided for evaluation. Clicking on the build name leads to the details page of the build:

Figure 8.2: Detailed view of a build instance

For each build attempt, a history of the build, tagged with a number, is provided for evaluation. Clicking on the build name leads to the details page of the build.

The **Logs** tab of the build details page shows the output generated by the build execution. Those logs are handy to identify build issues.

Use the **Deployment** link under **Workloads** section from the left panel to identify issues during the deployment step.

After selecting the appropriate deployment object, details show in the **Details** section.

The `oc` command-line interface has several subcommands for managing the logs. Likewise in the web interface, it has a set of commands which provides information about each step. For example, to retrieve the logs from a build configuration, run the following command:

```
[user@host ~]$ oc logs bc/<application-name>
```

If a build fails, after finding and fixing the issues, run the following command to request a new build:

```
[user@host ~]$ oc start-build <application-name>
```

By issuing that command, OpenShift automatically spawns a new pod with the build process.

Deployment logs can be checked with the `oc` command:

```
[user@host ~]$ oc logs deployment/<application-name>
```

If the deployment is running or has failed, the command returns the logs of the process deployment process. Otherwise, the command returns the logs from the application's pod.

Describing Common Problems

Sometimes, the source code requires some customization that may not be available in containerized environments, such as database credentials, file system access, or message queue information. Those values usually take the form of internal environment variables. Developers using the S2I process may need to access this information.

The `oc logs` command provides important information about the build, deploy, and run processes of an application during the execution of a pod. The logs may

indicate missing values or options that must be enabled, incorrect parameters or flags, or environment incompatibilities.

Note

Application logs must be clearly labeled to identify problems quickly without the need to learn the container internals.

Troubleshooting Permission Issues

OpenShift runs S2I containers using Red Hat Enterprise Linux as the base image, and any runtime difference may cause the S2I process to fail. Sometimes, the developer runs into permission issues, such as access denied due to the wrong permissions, or incorrect environment permissions set by administrators. S2I images enforce the use of a different user than the `root` user to access file systems and external resources. Also, Red Hat Enterprise Linux 8 enforces SELinux policies that restrict access to some file system resources, network ports, or process.

Some containers may require a specific user ID, whereas S2I is designed to run containers using a random user as per the default OpenShift security policy.

The following Dockerfile creates a Nexus container. Note the `USER` instruction indicating the `nexus` user should be used:

```
FROM ubi8/ubi:8.1
...contents omitted...
RUN chown -R nexus:nexus ${NEXUS_HOME}

USER nexus
WORKDIR ${NEXUS_HOME}

VOLUME ["/opt/nexus/sonatype-work"]
...contents omitted...
```

Trying to use the image generated by this Dockerfile without addressing volume permissions drives to errors when the container starts:

```
[user@host ~]$ oc logs nexus-1-wzjrn
...output omitted...
```

```
... org.sonatype.nexus.util.LockFile - Failed to write lock file
...FileNotFoundException: /opt/nexus/sonatype-work/nexus.lock (Permission denied)
...output omitted...
... org.sonatype.nexus.webapp.WebappBootstrap - Failed to initialize
...StateException: Nexus work directory already in use: /opt/nexus/sonatype-work
...output omitted...
```

To solve this issue, relax the OpenShift project security with the command `oc adm policy`:

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

This `oc adm policy` command enables OpenShift executing container processes with non-root users. But the file systems used in the container must also be available for the running user. This is specially important when the container contains volume mounts.

To avoid file system permission issues, local folders used for container volume mounts must satisfy the following:

- The user executing the container processes must be the owner of the folder, or have the necessary rights. Use the `chown` command to update folder ownership.
- The local folder must satisfy the SELinux requirements to be used as a container volume. Assign the `container_file_t` group to the folder by using the `semanage fcontext -a -t container_file_t <folder>` command, then refresh the permissions with the `restorecon -R <folder>` command.

Troubleshooting Invalid Parameters

Multi-container applications may share parameters, such as login credentials. Ensure that the same values for parameters reach all containers in the application. For example, for a Python application that runs in one container, connected with another container running a database, make sure that the two containers use the same user name and password for the database. Usually, logs from the application pod provide a clear idea of these problems and how to solve them.

A good practice to centralize shared parameters is to store them in `ConfigMaps`. Those `ConfigMaps` can be injected through the `Deployment` into containers as environment variables. Injecting the same `ConfigMap` into different containers

ensures that not only the same environment variables are available, but also the same values. See the following pod resource definition:

```
apiVersion: v1
kind: Pod
...output omitted...
spec:
  containers:
    - name: test-container
  ...output omitted...
  env:
    - name: ENV_1
      valueFrom:
        configMapKeyRef:
          name: configMap_name1
          key: configMap_key_1
  ...output omitted...
  envFrom:
    - configMapRef:
        name: configMap_name_2
  ...output omitted...
```

An ENV_1 environment variable is injected into the container. Its value is the value for the configMap_key_1 entry in the configMap_name1 configMap.

All entries in configMap_name_2 are injected into the container as environment variables with the same name and values.

Troubleshooting Volume Mount Errors

When redeploying an application that uses a persistent volume on a local file system, a pod might not be able to allocate a persistent volume claim even though the persistent volume indicates that the claim is released.

To resolve the issue, delete the persistent volume claim and then the persistent volume. Then recreate the persistent volume:

```
[user@host ~]$ oc delete pv <pv_name>
```

```
[user@host ~]$ oc create -f <pv_resource_file>
```

Troubleshooting Obsolete Images

OpenShift pulls images from the source indicated in an image stream unless it locates a locally-cached image on the node where the pod is scheduled to run. If you push a new image to the registry with the same name and tag, you must remove the image from each node the pod is scheduled on with the command `podman rmi`.

Run the `oc adm prune` command for an automated way to remove obsolete images and other resources.

References

More information about troubleshooting images is available in the *Images* section of the OpenShift Container Platform documentation accessible at: [Creating Images](#)

Documentation about how to consume ConfigMap to create container environment variables can be found in the *Consuming in Environment Variables* of the [Configure a Pod to use ConfigMaps](#)

[Next](#)

Guided Exercise: Troubleshooting an OpenShift Build

In this exercise, you will troubleshoot an OpenShift build and deployment process.

Outcomes

You should be able to identify and solve the problems raised during the build and deployment process of a Node.js application.

Make sure you have completed the [the section called “Guided Exercise: Configuring the Classroom Environment”](#) from *Chapter 1* before executing any command of this practice.

Retrieve the lab files and verify that Docker and the OpenShift cluster are running by running the following command.

```
[student@workstation ~]$ lab troubleshoot-s2i start
```

Procedure 8.1. Instructions

1. Load the configuration of your classroom environment. Run the following command to load the environment variables created in the first guided exercise:

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

2. 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:

3. [student@workstation ~]\$ cd ~/D0180-apps
4. [student@workstation D0180-apps]\$ git checkout master

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

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

6. [student@workstation D0180-apps]\$ git checkout -b troubleshoot-s2i
7. Switched to a new branch 'troubleshoot-s2i'
8. [student@workstation D0180-apps]\$ git push -u origin troubleshoot-s2i
9. ...output omitted...
- 10.* [new branch] troubleshoot-s2i -> s2i

```
Branch 'troubleshoot-s2i' set up to track remote branch 'troubleshoot-s2i' from 'origin'.
```

11. Log in to OpenShift using the configured user, password and Master API URL.

12. [student@workstation D0180-apps]\$ oc login -u "\${RHT_OCP4_DEV_USER}" \
13. > -p "\${RHT_OCP4_DEV_PASSWORD}" "\${RHT_OCP4_MASTER_API}"
14. Login successful.

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

Create a new project named *youruser-nodejs*.

```
[student@workstation D0180-apps]$ oc new-project ${RHT_OCP4_DEV_USER}-nodejs
Now using project "youruser-nodejs" on server ...
...output omitted...
```

15. Build a new Node.js application using the Hello World image located at <https://github.com/yourgituser/D0180-apps/> in the nodejs-helloworld directory.

1. Run the `oc new-app` command to create the Node.js application. The command is provided in the `~/D0180/labs/troubleshoot-s2i/command.txt` file.
2. [student@workstation D0180-apps]\$ `oc new-app \`
3. > `--context-dir=nodejs-helloworld \`
4. > `https://github.com/${RHT_OCP4_GITHUB_USER}/D0180-apps#troubleshoot-s2i \`
5. > `-i nodejs:16-ubi8 --name nodejs-hello --build-env \`
6. > `npm_config_registry=http://${RHT_OCP4_NEXUS_SERVER}/repository/nodejs`
7. --> Found image e9b0166 ...*output omitted...*
- 8.
9. Node.js 16
- 10....*output omitted...*
- 11.--> Creating resources ...
12. imagestream.image.openshift.io "nodejs-hello" created
13. buildconfig.build.openshift.io "nodejs-hello" created
14. deployment.apps "nodejs-hello" created
15. service "nodejs-hello" created
- 16.--> Success
17. Build scheduled, use 'oc logs -f buildconfig/nodejs-hello' to track its progress.
18. Application is not exposed. You can expose services to the outside world by executing one or more of the commands below:
19. '`oc expose svc/nodejs-hello'`

Run '`oc status`' to view your app.

The `-i` indicates the builder image to use, `nodejs:12` in this case.

The `--context-dir` option defines which folder inside the project contains the source code of the application to build.

The `--build-env` option defines an environment variable to the builder pod. In this case, it provides the `npm_config_registry` environment variable to the builder pod, so it can reach the NPM registry.

Important

In the previous command, there must be no spaces between `registry=` and the URL of the Nexus server.

20. Wait until the application finishes building by monitoring the progress with the `oc get pods -w` command. The pod transitions from a status of running to Error:

```
21. [student@workstation D0180-apps]$ oc get pods -w
22. NAME          READY   STATUS    RESTARTS   AGE
23. nodejs-hello-1-build  1/1     Running   0          15s
24. nodejs-hello-1-build  0/1     Error     0          73s
```

`^C`

The build process fails, and therefore no application is running. Build failures are usually consequences of syntax errors in the source code or missing dependencies. The next step investigates the specific causes for this failure.

25. Evaluate the errors raised during the build process.

The build is triggered by the build configuration (`bc`) created by OpenShift when the S2I process starts. By default, the OpenShift S2I process creates a build configuration named as the name given: `nodejs-hello`, which is responsible for triggering the build process.

Run the `oc` command with the `logs` subcommand in a terminal window to review the output of the build process:

```
[student@workstation D0180-apps]$ oc logs bc/nodejs-hello
Cloning "https://github.com/yourgituser/D0180-apps" ...
```

```
Commit: f7cd8963ef353d9173c3a21dccc402f3616840b ( Initial commit...
...output omitted...

STEP 8: RUN /usr/libexec/s2i/assemble
---> Installing application source ...
---> Installing all dependencies
npm ERR! code ETARGET
npm ERR! notarget No matching version found for express@~4.14.2.
npm ERR! notarget In most cases you or one of your dependencies are requesting
npm ERR! notarget a package version that doesn't exist.
npm ERR! notarget
npm ERR! notarget It was specified as a dependency of 'src'
npm ERR! notarget

npm ERR! A complete log of this run can be found in:
npm ERR!     /opt/app-root/src/.npm/_logs/2019-10-25T12_37_56_853Z-debug.log
`error: build error: error building at STEP "RUN /usr/libexec/s2i/assemble"
": error while running runtime: exit status 1
`...
...output omitted...
```

The log shows an error occurred during the build process. This output indicates that there is no compatible version for the `express` dependency. But the reason is that the format used by the `express` dependency is not valid.

16. Update the build process for the project.

The developer uses a nonstandard version of the Express framework that is available locally on each developer's workstation. Due to the company's standards, the version must be downloaded from the Node.js official registry and, from the developer's input, it is compatible with the 4.14.x version.

1. Fix the `package.json` file.

Use your preferred editor to open the `~/D0180-apps/nodejs-helloworld/package.json` file. Review the dependencies versions provided

by the developers. It uses an incorrect version of the Express dependency, which is incompatible with the supported version provided by the company (~4.14.2). Update the dependency version as follows.

```
{  
  "name": "nodejs-helloworld",  
  ...output omitted...  
  "dependencies": {  
    "express": "4.14.x"  
  }  
}
```

Note

Notice the x in the version. It indicates that the highest version should be used, but the version must begin with 4.14..

2. Commit and push the changes made to the project.

From the terminal window, run the following command to commit and push the changes:

```
[student@workstation D0180-apps]$ git commit -am "Fixed Express release"  
...output omitted...  
1 file changed, 1 insertion(+), 1 deletion(-)  
[student@workstation D0180-apps]$ git push  
...output omitted...  
To https://github.com/yourgituser/D0180-apps  
ef6557d..73a82cd troubleshoot-s2i -> troubleshoot-s2i
```

17. Relaunch the S2I process.

1. To restart the build step, execute the following command:

2. [student@workstation D0180-apps]\$ oc start-build bc/nodejs-hello

```
build.build.openshift.io/nodejs-hello-2 started
```

The build step is restarted, and a new build pod is created. Check the log by running the `oc logs` command.

```
[student@workstation D0180-apps]$ oc logs -f bc/nodejs-hello
Cloning "https://github.com/yougituser/D0180-apps" ...
Commit: ea2125c1bf4681dd9b79ddf920d8d8be38cf3b (Fixed Express release)
...output omitted...
Pushing image ...image-registry.svc:5000/nodejs/nodejs-hello:latest...
...output omitted...
Push successful
```

The build is successful, however, this does not indicate that the application is started.

- Evaluate the status of the current build process. Run the `oc get pods` command to check the status of the Node.js application.

```
[student@workstation D0180-apps]$ oc get pods
```

According to the following output, the second build completed, but the application is in error state.

NAME	READY	STATUS	RESTARTS	AGE
nodejs-hello-1-build	0/1	Error	0	7m59s
nodejs-hello-2-build	0/1	Completed	0	54s
nodejs-hello-7b99966464-fw4r8	0/1	CrashLoopBackOff	1	18s

The name of the application pod (`nodejs-hello-7b99966464-fw4r8`) is generated randomly, and may differ from yours.

- Review the logs generated by the application pod.

- `[student@workstation D0180-apps]$ oc logs nodejs-hello-7b99966464-fw4r8`
- ...output omitted...*
- `npm info using npm@8.1.2`
- `npm info using node@v16.13.1`

```
9. npm ERR! missing script: start
```

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

Note

The `oc logs nodejs-hello-7b99966464-fw4r8` command dumps the logs from the deployment pod. In the case of a successful deployment, that command dumps the logs from the application pod, as previously shown.

If the deployment logs do not display the error, then check the logs of the application pod by running the `oc logs POD` command replacing POD with the name of the pod that has a status of CrashLoopBackOff.

The application fails to start because the start script declaration is missing.

18. Fix the problem by updating the application code.

1. Update the `package.json` file to define a startup command.

The previous output indicates that the `~/D0180-apps/nodejs-helloworld/package.json` file is missing the `start` attribute in the `scripts` field. The `start` attribute defines a command to run when the application starts. It invokes the `node` binary, which runs the `app.js` application.

To fix the problem, add to the `package.json` file the following attribute. Do not forget the comma after the bracket.

```
...
  "description": "Hello World!",
  "main": "app.js",
  "scripts": { "start": "node app.js" },
  "author": "Red Hat Training",
...

```

2. Commit and push the changes made to the project:

```
3. [student@workstation D0180-apps]$ git commit -am "Added start up script"
4. ...output omitted...
5. 1 file changed, 3 insertions(+)
6. [student@workstation D0180-apps]$ git push
7. ...output omitted...
8. To https://github.com/yourgituser/D0180-apps
```

```
73a82cd..a5a0411 troubleshoot-s2i -> troubleshoot-s2i
```

Continue the deploy step from the S2I process.

9. Restart the build step.

```
10. [student@workstation D0180-apps]$ oc start-build bc/nodejs-hello
```

```
build.build.openshift.io/nodejs-hello-3 started
```

11. Evaluate the status of the current build process. Run the command to retrieve the status of the Node.js application. Wait for the latest build to finish.

```
12. [student@workstation D0180-apps]$ oc get pods -w
```

13. NAME	READY	STATUS	RESTARTS	AGE
14. nodejs-hello-1-build	0/1	Error	0	26m
15. nodejs-hello-2-build	0/1	Completed	0	19m
16. nodejs-hello-3-build	1/1	Running	0	9s
17. nodejs-hello-7b99966464-fw4r8	0/1	CrashLoopBackOff	8	19m
18. nodejs-hello-6dcd88b7b7-dvtcz	0/1	Pending	0	0s
19. nodejs-hello-6dcd88b7b7-dvtcz	0/1	Pending	0	0s
20. nodejs-hello-6dcd88b7b7-dvtcz	0/1	ContainerCreating	0	0s
21. nodejs-hello-3-build	0/1	Completed	0	37s
22. nodejs-hello-6dcd88b7b7-dvtcz	0/1	ContainerCreating	0	2s
23. nodejs-hello-6dcd88b7b7-dvtcz	1/1	Running	0	3s
24. nodejs-hello-7b99966464-fw4r8	0/1	Terminating	8	19m
25. nodejs-hello-7b99966464-fw4r8	0/1	Terminating	8	19m

nodejs-hello-7b99966464-fw4r8	0/1	Terminating	8	19m
-------------------------------	-----	-------------	---	-----

According to the output, the build is successful, and the application is able to start with no errors. The output also provides insight into how the deployment pod (`nodejs-hello-3-build`) was created, and that it completed successfully and terminated. As the new application pod is available (`nodejs-hello-7b99966464-dvtcz`), the old one (`nodejs-hello-7b99966464-fw4r8`) is rolled out.

26. Review the logs generated by the `nodejs-hello` application pod.

```
27. [student@workstation D0180-apps]$ oc logs nodejs-hello-6cd88b7b7-dvtcz
28. Environment:
29.     DEV_MODE=false
30.     NODE_ENV=production
31.     DEBUG_PORT=5858
32. Launching via npm...
33. npm info it worked if it ends with ok
34. npm info using npm@8.1.2
35. npm info using node@v16.13.1
36. npm info lifecycle nodejs-helloworld@1.0.0~prestart: nodejs-helloworld@1.0
   .0
37. npm info lifecycle nodejs-helloworld@1.0.0~start: nodejs-helloworld@1.0.0
38.
39. > nodejs-helloworld@1.0.0 start
40. > node app.js
41.
```

Example app listening on port 8080!

The application is now running on port 8080.

19. Test the application.

1. Run the `oc` command with the `expose` subcommand to expose the application:

```
2. [student@workstation D0180-apps]$ oc expose svc/nodejs-hello
```

route.route.openshift.io/nodejs-hello exposed

3. Retrieve the address associated with the application.

```
4. [student@workstation D0180-apps]$ oc get route -o yaml
5. apiVersion: v1
6. items:
7. - apiVersion: route.openshift.io/v1
8.   kind: Route
9.   ...output omitted...
10.  spec:
11.    host: nodejs-hello-${RHT_OCP4_DEV_USER}-nodejs.${RHT_OCP4_WILDCARD_DOMAIN}
12.    port:
13.      targetPort: 8080-tcp
14.    to:
15.      kind: Service
16.      name: nodejs-hello
```

...output omitted...

17. Access the application from the workstation VM by using the curl command:

```
18. [student@workstation D0180-apps]$ curl -w "\n" \
19. > http://nodejs-hello-${RHT_OCP4_DEV_USER}-nodejs.${RHT_OCP4_WILDCARD_DOMAIN}
```

Hello world!

The output demonstrates the application is up and running.

Finish

On workstation, run the lab troubleshoot-s2i finish script to complete this exercise.

```
[student@workstation D0180-apps]$ lab troubleshoot-s2i finish
```

This concludes the exercise.

[Previous](#) [Next](#)

Troubleshooting Containerized Applications

Objectives

After completing this section, you should be able to:

- Implement techniques for troubleshooting and debugging containerized applications.
- Use the port-forwarding feature of the OpenShift client tool.
- View container logs.
- View OpenShift cluster events.

Forwarding Ports for Troubleshooting

Occasionally developers and system administrators need special network access to a container that would not be needed by application users. For example, developers may need to use the administration console for a database or messaging service, or system administrators may make use of SSH access to a container to restart a terminated service. Such network access, in the form of network ports, are usually not exposed by the default container configurations, and tend to require specialized clients used by developers and system administrators.

Podman provides port forwarding features by using the `-p` option along with the `run` subcommand. In this case, there is no distinction between network access for regular application access and for troubleshooting. As a refresher, the following is an example of configuring port forwarding by mapping the port from the host to a database server running inside a container:

```
[user@host ~]$ podman run --name db -p 30306:3306 mysql
```

The previous command maps the host port 30306 to the port 3306 on the `db` container. This container is created from the `mysql` image, which starts a MySQL server that listens on port 3306.

OpenShift provides the `oc port-forward` command for forwarding a local port to a pod port. This is different than having access to a pod through a service resource:

- The port-forwarding mapping exists only in the workstation where the `oc` client runs, while a service maps a port for all network users.

- A service load balances connections to potentially multiple pods, whereas a port-forwarding mapping forwards connections to a single pod.

Here is an example of the `oc port-forward` command:

```
[user@host ~]$ oc port-forward db 30306 3306
```

The preceding command forwards port 30306 from the developer machine to port 3306 on the `db` pod, where a MySQL server (inside a container) accepts network connections.

Note

When running this command, be sure to leave the terminal window running. Closing the window or canceling the process stops the port mapping.

While the `podman run -p` method of mapping (port-forwarding) can only be configured when the container is started, the mapping with the `oc port-forward` command can be created and destroyed at any time after a pod was created.

Note

Creating a service of `NodePort` type for a database pod would be similar to running `podman run -p`. However, Red Hat discourages the usage of the `NodePort` approach to avoid exposing the service to direct connections. Mapping with port-forwarding in OpenShift is considered a more secure alternative.

Enabling Remote Debugging with Port Forwarding

Another use for the port forwarding feature is enabling remote debugging. Many integrated development environments (IDEs) provide the capability to remotely debug an application.

For example, Red Hat CodeReady Studio allows users to utilize the Java Debug Wire Protocol (JDWP) to communicate between its debugger and the Java Virtual Machine. When enabled, developers can step through each line of code as it is being executed in real time.

For JDWP to work, the Java Virtual Machine (JVM) where the application runs must be started with options enabling remote debugging. For example, WildFly and JBoss EAP users must configure these options on application server startup. The following line in the `standalone.conf` file enables remote debugging by opening the JDWP TCP port 8787, for a WildFly or EAP instance running in standalone mode:

```
JAVA_OPTS="$JAVA_OPTS \
> -agentlib:jdwp=transport=dt_socket,address=8787,server=y,suspend=n"
```

When the server starts with the debugger listening on port 8787, a port forwarding mapping needs to be created to forward connections from a local unused TCP port to port 8787 in the EAP pod. If the developer workstation has no local JVM running with remote debugging enabled, the local port can also be 8787.

The following command assumes a WildFly pod named `jappserver` running a container from an image previously configured to enable remote debugging:

```
[user@host ~]$ oc port-forward jappserver 8787:8787
```

Once the debugger is enabled and the port forwarding mapping is created, users can set breakpoints in their IDE of choice and run the debugger by pointing to the application's host name and debug port (in this instance, 8787).

Accessing Container Logs

Podman and OpenShift provide the ability to view logs in running containers and pods to facilitate troubleshooting. But neither of them is aware of application specific logs. Both expect the application to be configured to send all logging output to the standard output.

A container is simply a process tree from the host OS perspective. When Podman starts a container either directly or on the RHOC cluster, it redirects the container standard output and standard error, saving them on disk as part of the container's ephemeral storage. This way, the container logs can be viewed using `podman` and `oc` commands, even after the container was stopped, but not removed.

To retrieve the output of a running container, use the following `podman` command:

```
[user@host ~]$ podman logs containerName
```

In OpenShift, the following command returns the output for a container within a pod:

```
[user@host ~]$ oc logs podName -c containerName
```

Note

The `-c containerName` arguments are optional if there is only one container, as `oc` defaults to the only running container and returns the output.

OpenShift Events

Some developers consider Podman and OpenShift logs to be too low-level, making troubleshooting difficult. Fortunately, OpenShift provides a high-level logging and auditing facility called events.

OpenShift events signal significant actions like starting a container or destroying a pod.

To read OpenShift events, use the `get` subcommand with the `events` resource type for the `oc` command, as follows:

```
[user@host ~]$ oc get events
```

Events listed by the `oc` command this way are not filtered and span the whole RHOCP cluster. Using a pipe to standard UNIX filters such as `grep` can help, but OpenShift offers an alternative in order to consult cluster events. The approach is provided by the `describe` subcommand.

For example, to only retrieve the events that relate to a `mysql` pod, refer `Events` field from the output of `oc describe pod mysql` command:

```
[user@host ~]$ oc describe pod mysql
...output omitted...
Events:
  FirstSeen    LastSeen    Count  From           Reason          Message
  Wed, 10 ...  Wed, 10 ...  1     {scheduler}  scheduled      Successfully as...
```

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

Accessing Running Containers

The `podman logs` and `oc logs` commands can be useful for viewing output sent by any container. However, the output does not necessarily display all of the available information if the application is configured to send logs to a file. Other troubleshooting scenarios may require inspecting the container environment as seen by processes inside the container, such as verifying external connectivity.

As a solution, Podman and OpenShift provide the `exec` subcommand, allowing the creation of new processes inside a running container, with the standard output and input of these processes redirected to the user terminal. The following screen display the usage of the `podman exec` command:

```
[user@host ~]$ podman exec options container command arguments
```

The general syntax for the `oc exec` command is:

```
[user@host ~]$ oc exec options pod -c container -- command arguments
```

To execute a single interactive command or start a shell, add the `-it` options. The following example starts a Bash shell for the `myhttpd` pod:

```
[user@host ~]$ oc exec -it myhttpd /bin/bash
```

You can use this command to access application logs saved to disk (as part of the container ephemeral storage). For example, to display the Apache error log from a container, run the following command:

```
[user@host ~]$ podman exec apache-container cat /var/log/httpd/error_log
```

Overriding Container Binaries

Many container images do not contain all of the troubleshooting commands users expect to find in regular OS installations, such as `telnet`, `netcat`, `ip`, or `traceroute`. Stripping the image from basic utilities or binaries allows the image to remain slim, thus, running many containers per host.

One way to temporarily access some of these missing commands is mounting the host binaries folders, such as `/bin`, `/sbin`, and `/lib`, as volumes inside the container. This is possible because the `-v` option from `podman run` command does not require matching `VOLUME` instructions to be present in the `Containerfile` of the container image.

Note

To access these commands in OpenShift, you need to change the pod resource definition in order to define `volumeMounts` and `volumeClaims` objects. You also need to create a `hostPath` persistent volume.

The following command starts a container, and overrides the image's `/bin` folder with the one from the host. It also starts an interactive shell inside the container:

```
[user@host ~]$ podman run -it -v /bin:/bin image /bin/bash
```

Note

The directory of binaries to override depends on the base OS image. For example, some commands require shared libraries from the `/lib` directory. Some Linux distributions have different contents in `/bin`, `/usr/bin`, `/lib`, or `/usr/lib`, which would require to use the `-v` option for each directory.

As an alternative, you can include these utilities in the base image. To do so, add instructions in a `Containerfile` build definition. For example, examine the following excerpt from a `Containerfile` definition, which is a child of the `rhe17.5` image. The `RUN` instruction installs the tools that are commonly used for network troubleshooting:

```
FROM ubi7/ubi:7.7

RUN yum install -y less dig ping iputils && \
    yum clean all
```

When the image is built and the container is created, it will be identical to a `rhe17.5` container image, plus the extra available tools.

Transferring Files to and from Containers

When troubleshooting or managing an application, you might need to retrieve or transfer files to and from running containers, such as configuration files or log files. There are several ways to move files into and out of containers, as described in the following list.

Volume mounts

Another option for copying files from the host to a container is the usage of volume mounts. You can mount a local directory to copy data into a container. For example, the following command sets /conf host directory as the volume to use for the Apache configuration directory in the container. This provides a convenient way to manage the Apache server without having to rebuild the container image.

```
[user@host ~]$ podman run -v /conf:/etc/httpd/conf -d do180/apache
```

podman cp

The `cp` subcommand allows users to copy files both into and out of a running container. To copy a file into a container named `todoapi`, run the following command.

```
[user@host ~]$ podman cp standalone.conf \
> todoapi:/opt/jboss/standalone/conf/standalone.conf
```

To copy a file from the container to the host, flip the order of the previous command.

```
[user@host ~]$ podman cp todoapi:/opt/jboss/standalone/conf/standalone.conf .
```

The `podman cp` command has the advantage of working with containers that were already started, while the following alternative (volume mounts) requires changes to the command used to start a container.

podman exec

For containers that are already running, the `podman exec` command can be piped to pass files both into and out of the running container by appending commands that are executed in the container. The following example shows how to pass in and execute a SQL file inside a MySQL container:

```
[user@host ~]$ podman exec -i container mysql -uroot -proot \
```

```
> < /path/on/host/db.sql
```

Using the same concept, it is possible to retrieve data from a running container and place it in the host machine. A useful example of this is the usage of the `mysqldump` utility, which creates a backup of MySQL database from the container and places it on the host.

```
[user@host ~]$ podman exec -it containerName sh \
> -c 'exec mysqldump -h"$MYSQL_PORT_3306_TCP_ADDR" \
> -P"$MYSQL_PORT_3306_TCP_PORT" \
> -uroot -p"$MYSQL_ENV_MYSQL_ROOT_PASSWORD" items' > db_dump.sql
```

The preceding command uses the container environment variables to connect to the MySQL server to execute the `mysqldump` command and redirects the output to a file on the host machine. It assumes that the container image provides the `mysqldump` utility, so there is no need to install the MySQL administration tools on the host.

The `oc rsync` command provides functionality similar to `podman cp` for containers running under OpenShift pods.

References

More information about port-forwarding is available in the *Port Forwarding* section of the OpenShift Container Platform documentation at [Architecture](#)

More information about the CLI commands for port-forwarding are available in the *Port Forwarding* chapter of the OpenShift Container Platform documentation at [Developing Applications](#)

[Previous](#) [Next](#)

Guided Exercise: Configuring Apache Container Logs for Debugging

In this exercise, you will configure an Apache httpd container to send the logs to the `stdout`, then review Podman logs and events.

Outcomes

You should be able to configure an Apache httpd container to send debug logs to `stdout` and view them using the `podman logs` command.

Make sure you have completed the [the section called “Guided Exercise: Configuring the Classroom Environment”](#) from *Chapter 1* before executing any command of this practice.

Retrieve the lab files and verify that Docker and the OpenShift cluster are running by running the following command.

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

Procedure 8.2. Instructions

1. Configure a Apache web server to send log messages to the standard output and update the default log level.
 1. The default log level for the Apache httpd image is `warn`. Change the default log level for the container to `debug`, and redirect log messages to `stdout` by overriding the default `httpd.conf` configuration file. To do so, create a custom image from the workstation VM.

Briefly review the custom `httpd.conf` file located at `~/D0180/labs/troubleshoot-container/conf/httpd.conf`.

- Observe the `LogLevel` directive in the file. The directive changes the default log level to `debug`.
- Observe the `ErrorLog` directive in the file. The directive sends the `httpd` error log messages to the container’s standard output.
- Observe the `CustomLog` directive in the file. The directive redirects the `httpd` access log messages to the container’s standard output.

- `ErrorLog "/dev/stdout"`

-
- ...output omitted...
-
- LogLevel debug
-
- ...output omitted...
-

```
CustomLog "/dev/stdout" common
```

2. Build a custom container to save an updated configuration file to the container.

1. From the terminal window, run the following commands to build a new image.

2. [student@workstation ~]\$ cd ~/D0180/labs/troubleshoot-container
3. [student@workstation troubleshoot-container]\$ podman build \
4. > -t troubleshoot-container .
5. STEP 1: FROM quay.io/redhattraining/httpd-parent
6. ...output omitted...
7. STEP 5: COMMIT troubleshoot-container
8. e23d...c1de

```
[student@workstation troubleshoot-container]$ cd ~
```

9. Verify that the image is created.

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

The new image must be available in the local storage.

REPOSITORY SIZE	TAG	IMAGE ID	CREATED
localhost/troubleshoot-container 137MB	latest	e23df...	9 seconds ago
quay.io/redhattraining/httpd-parent 137MB	latest	0eba3...	4 weeks ago

3. Create a new httpd container from the custom image.

```
4. [student@workstation ~]$ podman run \
5. > --name troubleshoot-container -d \
6. > -p 10080:80 troubleshoot-container
```

```
4c8bb12815cc02f4eef0254632b7179bd5ce230d83373b49761b1ac41fc067a9
```

7. Review the container's log messages and events.
 1. View the debug log messages from the container using the `podman logs` command: Notice the debug logs, available in the standard output.
 2. [student@workstation ~]\$ podman logs -f troubleshoot-container
 3. ... [mpm_event:notice] [pid 1:tid...] AH00489: Apache/2.4.37 ...
 4. ... [mpm_event:info] [pid 1:tid...] AH00490: Server built: Apr 5 2019 07:31:21
 5. ... [core:notice] [pid 1:tid...] AH00094: Command line: 'httpd -D FOREGROUND'
 6. ... [core:debug] [pid 1:tid ...]: AH02639: Using SO_REUSEPORT: yes (1)
 7. ... [mpm_event:debug] [pid 6:tid ...]: AH02471: start_threads: Using epoll
 8. ... [mpm_event:debug] [pid 7:tid ...]: AH02471: start_threads: Using epoll
- ... [mpm_event:debug] [pid 8:tid ...]: AH02471: start_threads: Using epoll
9. Open a new terminal and access the home page of the web server by using the `curl` command:

```
10. [student@workstation ~]$ curl http://127.0.0.1:10080
```

```
Hello from the httpd-parent container!
```

11. Review the new entries in the log. Look in the terminal running the `podman logs` command to see the new entries.

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

```
10.0.2.2 - - [31/Mar/2022:14:06:03 +0000] "GET / HTTP/1.1" 200 39
```

13. Stop the Podman command with **Ctrl+C**.

Finish

On workstation, run the following script to complete this lab.

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

This concludes the guided exercise.

[Previous](#) [Next](#)

Summary

In this chapter, you learned:

- Applications typically log activity, such as events, warnings and errors, to aid the analysis of application behavior.
- Container applications should print log data to standard output, instead of to a file, to enable easy access to logs.
- To review the logs for a container deployed locally with Podman, use the `podman logs` command.
- Use the `oc logs` command to access logs for `BuildConfig` and `Deployment` objects, as well as individual pods within an OpenShift project.
- The `-f` option allows you to monitor the log output in near real-time for both the `podman logs` and `oc logs` commands.
- Use the `oc port-forward` command to connect directly to a port on an application pod. You should only leverage this technique on non-production pods, because interactions can alter the behavior of the pod.

[Previous](#) [Next](#)

Chapter 9. Comprehensive Review

[Comprehensive Review](#)

[Lab: Containerizing and Deploying a Software Application](#)

Abstract

Goal	Review tasks from <i>Red Hat OpenShift I: Containers & Kubernetes</i>
-------------	---

Objectives	<ul style="list-style-type: none"> • Review tasks from <i>Red Hat OpenShift I: Containers & Kubernetes</i>
Sections	<ul style="list-style-type: none"> • Comprehensive Review
Lab	<ul style="list-style-type: none"> • Comprehensive Review of Introduction to Containers, Kubernetes, and Red Hat OpenShift

Comprehensive Review

Objectives

After completing this section, you should have reviewed and refreshed the knowledge and skills learned in *Red Hat OpenShift I: Containers & Kubernetes*.

Reviewing *Red Hat OpenShift I: Containers & Kubernetes*

Before beginning the comprehensive review for this course, you should be comfortable with the topics covered in each chapter.

You can refer to earlier sections in the textbook for extra study.

Chapter 1. Introducing Container Technology

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

- 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.

Chapter 2. Creating Containerized Services

Provision a service using container technology.

- Create a database server from a container image.

Chapter 3. Managing Containers

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

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

Chapter 4. Managing Container Images

Manage the lifecycle of a container image from creation to deletion.

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

Chapter 5. Creating Custom Container Images

Design and code a Containerfile to build a custom container image.

- Describe the approaches for creating custom container images.
- Create a container image using common Containerfile commands.

Chapter 6. Deploying Containerized Applications on OpenShift

Deploy single container applications on OpenShift Container Platform.

- Describe the architecture of Kubernetes and Red Hat OpenShift Container Platform.
- Create standard Kubernetes resources.
- Create a route to a service.
- Build an application using the Source-to-Image facility of OpenShift Container Platform.
- Create an application using the OpenShift web console.

Chapter 7. Deploying Multi-Container Applications

Deploy applications that are containerized using multiple container images.

- Describe considerations for containerizing applications with multiple container images.
- Deploy a multi-container application on OpenShift Platform.
- Deploy an Application on OpenShift using a template.

Chapter 8. Troubleshooting Containerized Applications

Troubleshoot a containerized application deployed on OpenShift.

- Troubleshoot an application build and deployment on OpenShift.
- Implement techniques for troubleshooting and debugging containerized applications.

[Next](#)

Lab: Containerizing and Deploying a Software Application

In this review, you will containerize a Nexus server, build and test it using Podman, and deploy it to an OpenShift cluster.

Outcomes

You should be able to:

- Write a Containerfile that successfully containerizes a Nexus server.
- Build a Nexus server container image and deploy it using Podman.
- Deploy the Nexus server container image to an OpenShift cluster.

Make sure you have completed [the section called “Guided Exercise: Configuring the Classroom Environment”](#) from *Chapter 1* before executing any command of this practice.

Run the set-up script for this comprehensive review.

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

The lab files are located in the ~/D0180/labs/comprehensive-review directory. The solution files are located in the ~/D0180/solutions/comprehensive-review directory.

Procedure 9.1. Instructions

Use the following steps to create and test a containerized Nexus server both locally and in OpenShift:

1. Create a container image that starts an instance of a Nexus server:
 - The ~/D0180/labs/comprehensive-review/image directory contains files for building the container image. Execute the get-nexus-bundle.sh script to retrieve the Nexus server files.
 - Write a Containerfile that containerizes the Nexus server. The Containerfile must be located in the ~/D0180/labs/comprehensive-review/image directory. The Containerfile must also:

- Use a base image of `ubi8/ubi:8.5` and set an arbitrary maintainer.
- Set the ARG variable `NEXUS_VERSION` to `2.14.3-02`, and set the environment variable `NEXUS_HOME` to `/opt/nexus`.
- Install the `java-1.8.0-openjdk-devel` package.
- Run a command to create a `nexus` user and group. They both have a UID and GID of `1001`. Change the permissions of the `${NEXUS_HOME}/` directory to `775`.
- Unpack the `nexus-2.14.3-02-bundle.tar.gz` file to the `${NEXUS_HOME}/` directory. Add the `nexus-start.sh` to the same directory.

Run a command, `ln -s ${NEXUS_HOME}/nexus-${NEXUS_VERSION} ${NEXUS_HOME}/nexus2`, to create a symlink in the container. Run a command to recursively change the ownership of the Nexus home directory to `nexus:nexus`.

- Make the container run as the `nexus` user, and set the working directory to `/opt/nexus`.
- Define a volume mount point for the `/opt/nexus/sonatype-work` container directory. The Nexus server stores data in this directory.
- Set the default container command to `nexus-start.sh`.

There are two `*.snippet` files in the `~/D0180/labs/comprehensive-review/image` directory that provide the commands needed to create the `nexus` account and install Java. Use the files to assist you in writing the `Containerfile`.

- Build the container image with the name `nexus`.

Show Solution

2. Build and test the container image using Podman with a volume mount:
 - Use the script `~/D0180/labs/comprehensive-review/deploy/local/run-persistent.sh` to start a new container with a volume mount.
 - Review the container logs to verify that the server is started and running.
 - Test access to the container service using the URL: `http://127.0.0.1:18081/nexus`.
 - Remove the test container.

Show Solution

3. Deploy the Nexus server container image to the OpenShift cluster. You must:

- Tag the Nexus server container image as `quay.io/${RHT_OCP4_QUAY_USER}/nexus:latest`, and push it corresponding public repository on quay.io.
- Create an OpenShift project with a name of `${RHT_OCP4_DEV_USER}-review`.
- Edit the `deploy/openshift/resources/nexus-deployment.yaml` and replace `RHT_OCP4_QUAY_USER` with your Quay username. Create the Kubernetes resources.
- Create a route for the Nexus service. Verify that you can access `http://nexus-${RHT_OCP4_DEV_USER}-review.${RHT_OCP4_WILDCARD_DOMAIN}/nexus/` from workstation.

[Show Solution](#)

Evaluation

After deploying the Nexus server container image to the OpenShift cluster, verify your work by running the lab grading script:

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

Finish

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

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

This concludes the lab.

[Previous](#) [Next](#)

Appendix A. Implementing Microservices Architecture

Abstract

Goal	Refactor an application into microservices.
Objectives	<ul style="list-style-type: none">Divide an application across multiple containers to separate distinct layers and services.
Sections	<ul style="list-style-type: none">Implementing Microservices Architectures (with Guided Exercise)

Implementing Microservices Architectures

Objectives

After completing this section, you should be able to:

- Divide an application across multiple containers to separate distinct layers and services.
- Describe typical approaches to breaking up a monolithic application into multiple deployable units.
- Describe how to break the To Do List application into three containers matching its logical tiers.

Benefits of Breaking Up a Monolithic Application into Containers

Traditional application development typically has many distinct functions packaged as a single deployment unit, or a monolithic application. Traditional development may also deploy supporting services, such as databases and other middleware services, on the same server as the application. While monolithic applications can still be deployed into a container, many of the advantages of a container architecture, such as scalability and agility, are not as prevalent. Breaking up monoliths requires careful consideration and it is recommended that in microservices applications each microservice runs the minimum functionality that can be executed in isolation on each container.

Having smaller containers and breaking up an application and its supporting services into multiple containers provides many advantages, such as:

- Higher hardware utilization, because smaller containers are easier to fit into available host capacity.
- Easier scaling, because parts of the application can be scaled to support an increased workload without scaling other parts of the application.
- Easier upgrades, because developers can update parts of the application without affecting other parts of the same application.

Two popular ways of breaking up an application are as follows:

- Tiers: based on architectural layers.
- Services: based on application functionality.

Dividing Based on Layers (Tiers)

A common way developers organize applications is in tiers, based on how close the functions are to end users and how far from data stores. A good example of the traditional 3-tier architecture is presentation, business logic, and persistence.

This logical architecture usually corresponds to a physical deployment architecture, where the presentation layer would be deployed to a web server, the business layer to an application server, and the persistence layer to a database server.

Breaking up an application into tiers allows developers to specialize in particular technologies based on the application's tiers. For example, some developers focus on web applications, while others prefer database development. Another advantage is the ability to provide alternative tier implementations based on different technologies; for example, creating a mobile application as another front end for an existing application. The mobile application would be an alternative presentation tier, reusing the business and persistence tiers of the original web application.

Smaller applications usually have the presentation and business tiers deployed as a single unit. For example, to the same web server, but as the load increases, the presentation layer is moved to its own deployment unit to spread the load. Smaller applications might even embed the database. Developers often build and deploy more demanding applications in this monolithic fashion.

When developers break up a monolithic application into tiers, they usually apply several changes:

- Connection parameters to a database and other middleware services, such as messaging, were hard-coded to fixed IP addresses or host names, usually `localhost`. They need to be parameterized to point to external servers that might be different from development to production.
- In the case of web applications, Ajax calls cannot be made using relative URLs. They need to use an absolute URL pointing to a fixed public DNS host name.
- Modern web browsers refuse Ajax calls to servers different from the one containing the script that makes the call, as a security measure. The application needs to have permissions for cross-origin resource sharing (CORS).

After application tiers are divided so that they can run from different servers, there should be no problem running them from different containers.

Dividing Based on Discrete Services

Most complex applications are composed of many semi-independent services. For example, an online store would have a product catalog, shopping cart, payment, shipping, and so on.

When a particular service in a monolithic application degrades, scaling the service to improve performance implies scaling all of the other constituent application services. If however the degraded service is part of a microservices architecture, the affected service is scaled independent of the other application services. The following figure illustrates service scaling for both a monolithic and microservices-based architecture:

Figure A.1: Comparison of application scaling in a monolithic architecture versus a microservices architecture

Note

In the previous diagram:

- Red dashed line figures represent services that are overutilized.
- Yellow solid line figures represent services that are underutilized.
- Green solid line figures represent services that are correctly sized.

Both traditional service-oriented architectures (SOA) and more recent microservices architectures package and deploy those function sets as distinct units. This allows each function set to be developed by its own team, updated, and scaled without disturbing other function sets (or services). Cross-functional concerns such as authentication can also be packaged and deployed as services that are consumed by other service implementations.

Splitting each concern into a separated server might result in many applications. They are logically architected, packaged, and deployed as a small number of units, sometimes even as a single monolithic unit using a service approach.

Containers enable architectures based on services to be materialized during deployment. That is the reason microservices and containers usually come together. However, containers alone are not enough; they need to be complemented by orchestration tools to manage dependencies among services.

The microservices architecture takes service-based architectures to the extreme. A service is as small as it can be (without breaking a function set) and is deployed and managed as an independent unit, instead of part of a bigger application. This allows existing microservices to be reused to create new applications.

To break an application into services, it needs the same kind of change as when breaking into tiers; for example, parameterize connection parameters to databases and other middleware services and deal with web browser security protections.

Refactoring the To Do List Application

The To Do List application is a simple application with a single function set, so breaking it up into services is not really meaningful. However, refactoring it into presentation and business tiers, that is, into a front end and a back end to deploy

into distinct containers, illustrates the same kind of changes that breaking up a typical application into services would need.

The following figure shows the `To Do List` application deployed into three containers, one for each tier:

Figure A.2: To Do List application broken into tiers and each deployed as containers
Comparing the source code of the original monolithic application with the refactored one, this is an overview of the changes:

- The front-end JavaScript in `script/items.js` uses `workstation.lab.example.com` as the host name to reach the back end.
- The back end uses environment variables to get the database connection parameters.
- The back end has to reply to requests using the HTTP `OPTIONS` verb with headers telling the web browser to accept requests coming from different DNS domains using CORS .

Other versions of the back end service might have similar changes. Each programming language and REST framework have their own syntax and features.

References

[Monolithic application page in Wikipedia](#)

[CORS page in Wikipedia](#)

[Next](#)

Guided Exercise: Refactoring the To Do List Application

In this lab, you will refactor the `To Do List` application into multiple containers that are linked together, allowing the front-end HTML 5 application, the Node.js REST API, and the MySQL server to run in their own containers.

Outcomes

You should be able to refactor a monolithic application into its tiers and deploy each tier as a microservice.

Run the following command to set up the working directories for the lab with the `To Do List` application files:

```
[student@workstation ~]$ lab appendix-microservices start
```

Procedure A.1. Instructions

1. Move the HTML Files

The first step in refactoring the To Do List application is to move the front-end code from the application into its own running container. This step guides you through moving the HTML application and its dependent files into their own directory for deployment to an Apache server running in a container.

1. Move the HTML and static files to the `src/` directory from the monolithic Node.js To Do List application:

```
2. [student@workstation ~]$ cd ~/D0180/labs/appendix-microservices/apps/html5 /  
3. [student@workstation html5]$ mv \  
4. > ~/D0180/labs/appendix-microservices/apps/nodejs/todo/* \  
> ~/D0180/labs/appendix-microservices/apps/html5/src/
```

5. The current front-end application interacts with the API service using a relative URL. Because the API and front-end code will now run in separate containers, the front-end needs to be adjusted to point to the absolute URL of the To Do List application API.

Open the `/home/student/D0180/labs/appendix-microservices/apps/html5/src/script/item.js` file. At the bottom of the file, look for the following method:

```
app.factory('itemService', function ($resource) {  
    return $resource('api/items/:id');  
});
```

Replace that code with the following content:

```
app.factory('itemService', function ($resource) {  
    return $resource('http://workstation.lab.example.com:30080/todo/api/items/:id');
```

```
});
```

Make sure there are no line breaks in the new URL, save the file, and exit the editor.

2. Build the HTML Image

1. Login to the Red Hat Container Catalog with your Red Hat account.
2. [student@workstation html15]\$ podman login registry.redhat.io
3. Username: *your_username*
4. Password: *your_password*

Login Succeeded!

5. Build the child Apache image:

6. [student@workstation html15]\$ cd ~/D0180/labs/appendix-microservices/deploy/html15
7. [student@workstation html15]\$./build.sh
8. STEP 1: FROM registry.redhat.io/rhel8/httpd-24:1
9. Getting image source signatures
10. Copying blob 1b6a9fa02570 done
- 11....output omitted...
12. Storing signatures
13. STEP 2: COPY ./src/ \${HOME}/
14. STEP 3: COMMIT do180/todo_frontend

ddaa10a4bf12ff987331e482e5cd87658abc5724da4b5b7d136874a9e471acf71

15. Verify that the image is built correctly:

16. [student@workstation html15]\$ podman images
17. REPOSITORY TAG IMAGE ID CREATED
SIZE
18. localhost/do180/todo_frontend latest dda10a4bf12f About a minute
ago 438 MB
19. registry.redhat.io/rhel8/httpd-24 1 adcf72f31a0b 13 days ago
438 MB

...

3. Modify the REST API to Connect to External Containers

1. The REST API currently uses hard-coded values to connect to the MySQL database. Edit the `/home/student/D0180/labs/appendix-microservices/apps/nodejs/models/db.js` file, which holds the database configuration. Update the `dbname`, `username`, and `password` values to use environment variables instead. Also, update the `params.host` to point to the host name of the host running the MySQL container and update the `params.port` to reflect the redirected port to the container. Both values are available as the `MYSQL_SERVICE_HOST` and `MYSQL_SERVICE_PORT` environment variables, respectively. Replaced contents should look like this:

```
2. module.exports.params = {  
3.     dbname: process.env.MYSQL_DATABASE,  
4.     username: process.env.MYSQL_USER,  
5.     password: process.env.MYSQL_PASSWORD,  
6.     params: {  
7.         host: process.env.MYSQL_SERVICE_HOST,  
8.         port: process.env.MYSQL_SERVICE_PORT,  
9.         dialect: 'mysql'  
10.    }  
  
};
```

Note

This file can be copied and pasted from `/home/student/D0180/solutions/appendix-microservices/apps/nodejs/models/db.js`.

11. Configure the back end to handle Cross-origin resource sharing (CORS). This occurs when a resource request is made from a different domain from the one in which the request was made. Because the API needs to handle requests from a different DNS domain (the front-end application), it is necessary to create security exceptions to allow these requests to succeed. Make the following modifications to the application in the editor of your preference in order to handle CORS.

Add "restify-cors-middleware": "1.1.1" as a new dependency to the package.json file located at /home/student/D0180/labs/appendix-microservices/apps/nodejs/package.json. Remember to put a comma at the end of the previous dependency. Make sure the end of the file looks like this:

```
    "sequelize": "5.21.1",
    "mysql2": "2.0.0",
    "restify-cors-middleware": "1.1.1"
}
}
```

Update the app.js file located at /home/student/D0180/labs/appendix-microservices/apps/nodejs/app.js to configure CORS usage. Require the restify-cors-middleware module at the second line, and then update the contents of the file to match the following:

```
var restify = require('restify');
var corsMiddleware = require('restify-cors-middleware');
var controller = require('./controllers/items');
...output omitted...
var server = restify.createServer()
    .use(restify.plugins.fullResponse())
    .use(restify.plugins.queryParser())
    .use(restify.plugins.bodyParser());

const cors = corsMiddleware({ origins: ['*'] });

server.pre(cors.preflight);
server.use(cors.actual);

controller.context(server, '/todo/api', model);
```

The origins with value ["*"] instructs the server to allow any domains. In a production server, this value would usually be an array of domains known to require access to the API.

4. Build the REST API Image

1. Build the REST API child image using the following command. This image uses the Node.js image.

```
2. [student@workstation html5]$ cd ~/D0180/labs/appendix-microservices/deploy/nodejs  
3. [student@workstation nodejs]$ ./build.sh  
4. STEP 1: FROM quay.io/redhattraining/do180-todonodejs-12  
5. Getting image source signatures  
6. ...output omitted...  
7. STEP 6: CMD [ "/bin/bash", "-c", "./run.sh" ]  
8. STEP 7: COMMIT do180/todonodejs
```

```
18f8d5dd8e25ed19a54750c8b96ae075adf437f5f0ba5ebbf7b9fe4b75526b3
```

9. Run the `podman images` command to verify that all of the required images are built successfully:

10. [student@workstation nodejs]\$ podman images	REPOSITORY	TAG	IMAGE ID	CREAT
	ED	SIZE		
12. localhost/do180/todonodejs		latest	18f8d5dd8e25	About
a minute ago	852 MB			
13. localhost/do180/todo_frontend		latest	dda10a4bf12f	10 mi
minutes ago	438 MB			

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

5. Run the Containers

1. Use the `run.sh` script to run the containers:

```
2. [student@workstation nodejs]$ cd linked/  
3. [student@workstation linked]$ ./run.sh  
4. • Creating database volume: OK  
5. • Launching database: OK  
6. • Importing database: OK
```

```
• Launching To Do application: OK
```

7. Run the `podman ps` command to confirm that all three containers are running:

```
8. [student@workstation linked]$ podman ps  
9. ... IMAGE ... PORTS NA  
MES  
10.... localhost/do180/todo_frontend:latest ... 0.0.0.0:30000->80/tcp to  
do_frontend  
11.... localhost/do180/todonodejs:latest ... 0.0.0.0:30080->30080/tcp to  
doapi
```

```
... registry.redhat.io/rhel8/mysql-80:1 ... 0.0.0.0:30306->3306/tcp my  
sql
```

6. Test the Application

1. Use the `curl` command to verify that the REST API for the To Do List application is working correctly:

```
2. [student@workstation linked]$ curl -w "\n" 127.0.0.1:30080/todo/api/items/  
1
```

```
{"description": "Pick up newspaper", "done": false, "id":1}
```

3. Open Firefox on the workstation machine and navigate to 127.0.0.1:30000, where you should see the To Do List application.
4. Change back to the user's home folder.

```
5. [student@workstation linked]$ cd ~
```

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

Finish

On workstation, run the `lab appendix-microservices finish` script to complete this lab.

```
[student@workstation ~]$ lab appendix-microservices finish
```

This concludes the guided exercise.

[Previous](#) [Next](#)

Summary

In this chapter, you learned:

- Breaking a monolithic application into multiple containers allows for greater application scalability, makes upgrades easier, and allows higher hardware utilization.
- The three common tiers for logical division of an application are the presentation tier, the business tier, and the persistence tier.
- Cross-Origin Resource Sharing (CORS) can prevent Ajax calls to servers different from the one where the pages were downloaded. Be sure to make provisions to allow CORS from other containers in the application.
- Container images are intended to be immutable, but configurations can be passed in either at image build time or by creating persistent storage for configurations.

[Previous](#) [Next](#)

Appendix B. Creating a GitHub Account

Abstract

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

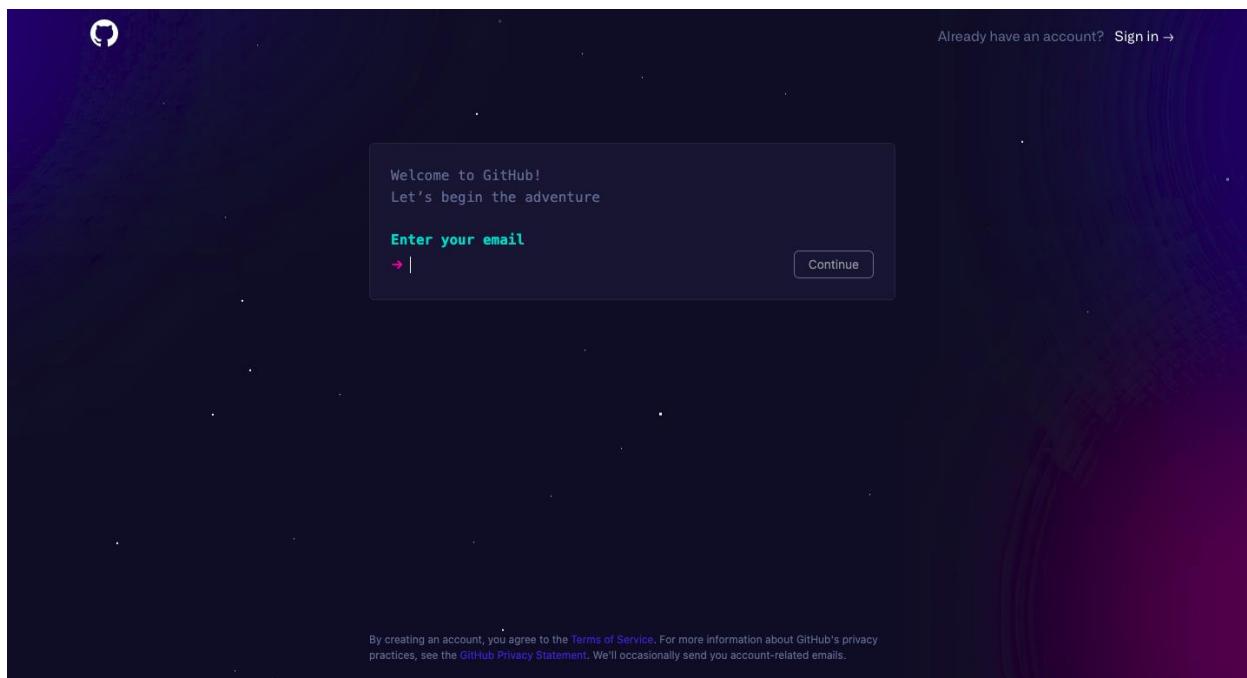
You need a GitHub account to create one or more *public* git repositories for the labs in this course. If you already have a GitHub account, you can skip the steps listed in this appendix.

Important

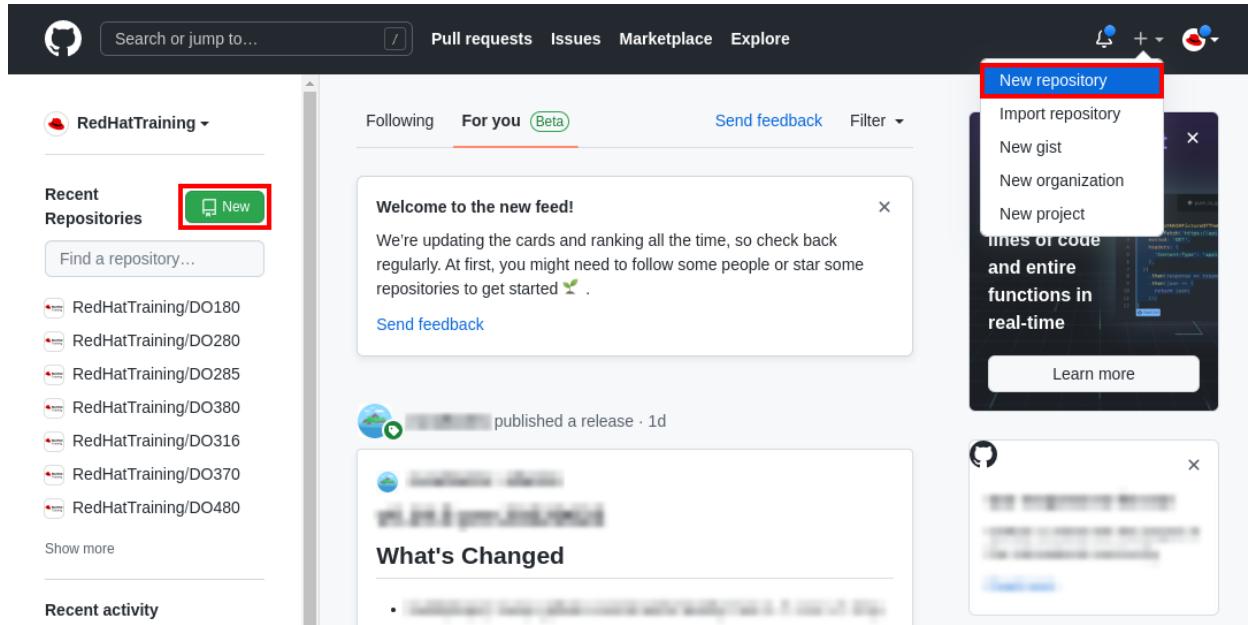
If you already have a GitHub account, ensure that you only create *public* git repositories for the labs in this course. The lab grading scripts and instructions require unauthenticated access to clone the repository. The repositories must be accessible without providing passwords, SSH keys, or GPG keys.

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 **Create account**.



3. Figure B.1: Creating a GitHub account
4. 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.
5. 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.



6. Figure B.2: Creating a new Git repository
7. Alternatively, click the plus icon (+) in the upper-right corner (to the right of the bell icon) and then click **New repository**.

References

[Signing up for a new GitHub account](#) [Creating a new repository](#)

[Next](#)

Appendix C. Creating a Quay Account

Abstract

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 credentials (created in [Appendix D, Creating a Red Hat Account](#)).

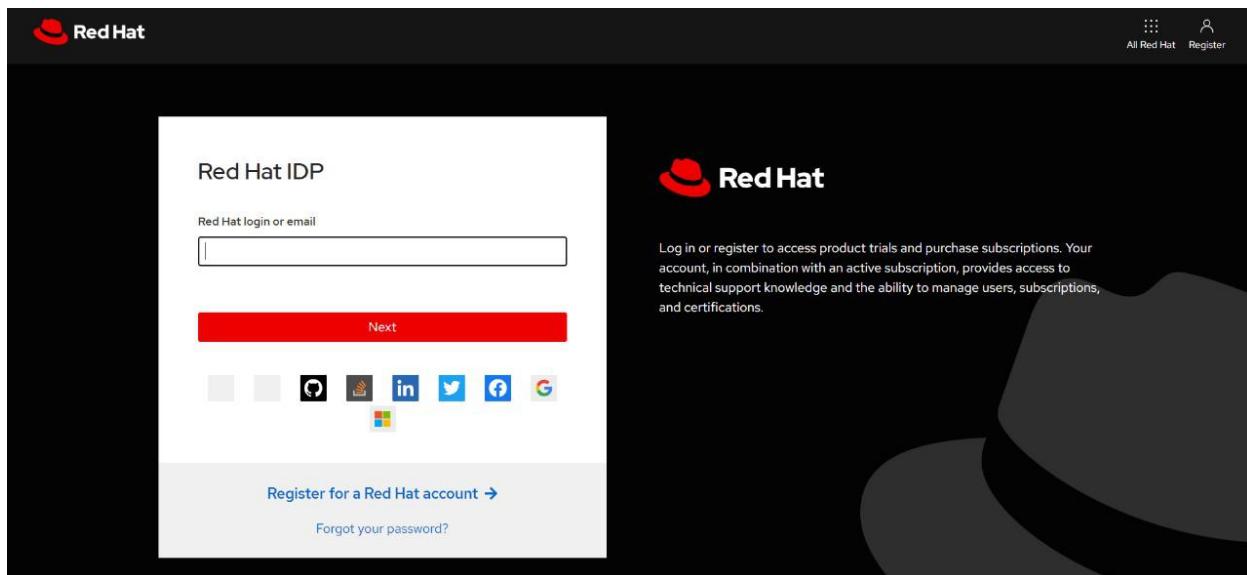


Figure C.1: Sign in using Red Hat credentials.

After you have logged in to Quay you can create new image repositories by clicking **Create New Repository** on the **Repositories** page.

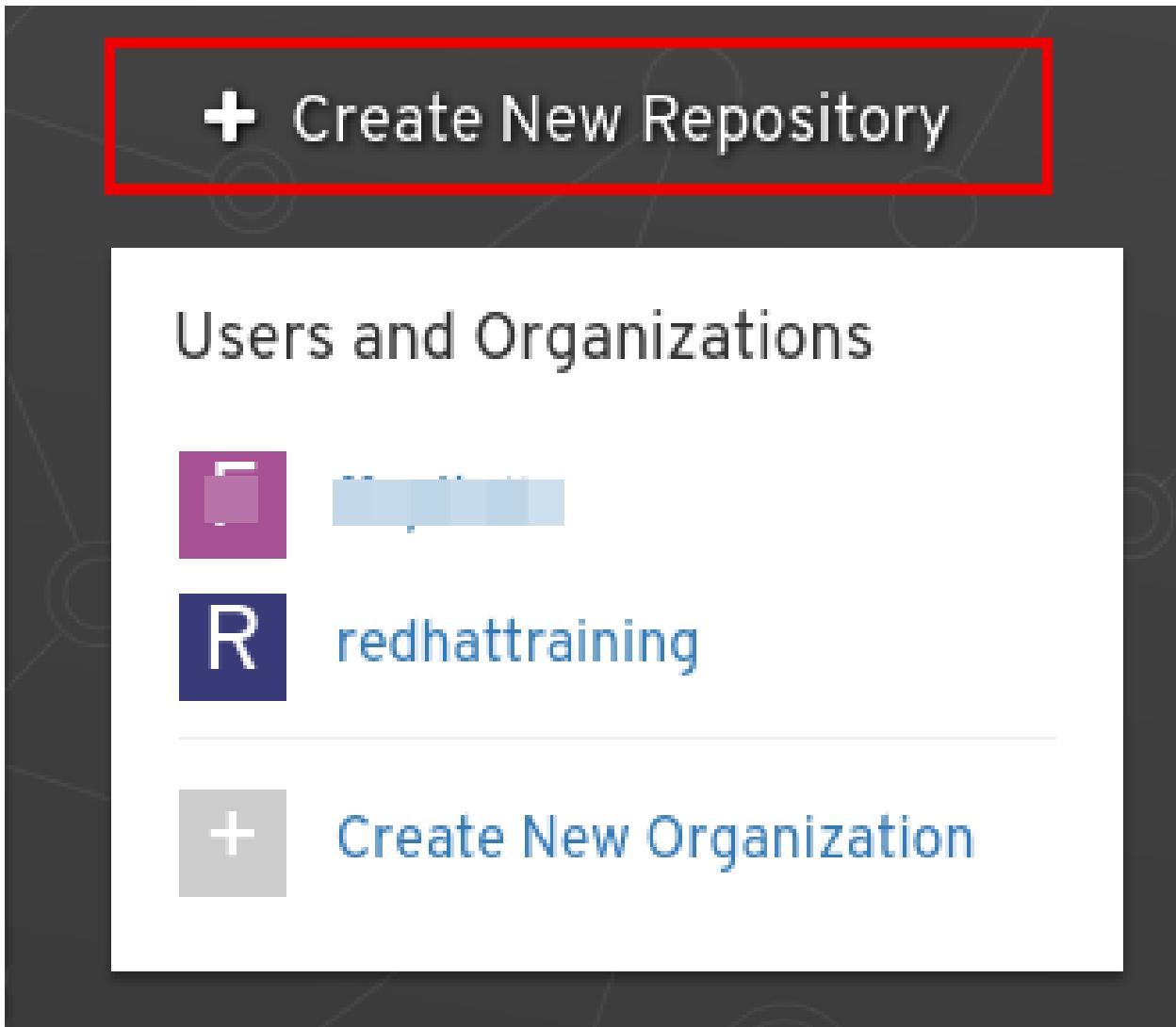


Figure C.2: Creating a new image repository

Alternatively, click the plus icon (+) in the upper-right corner (to the left of the bell icon), and then click **New Repository**.

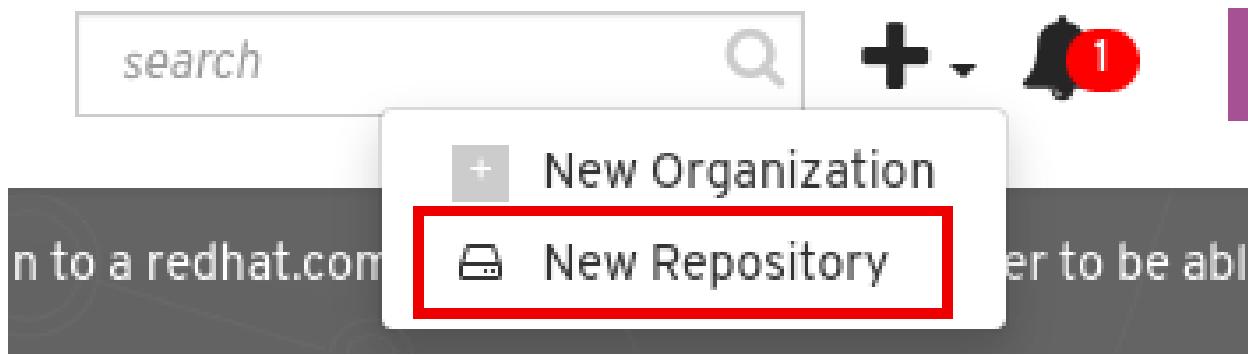


Figure C.3: Creating a new image repository

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 **User Settings** on the left column.
4. Click the **Set password** link.

References

[Getting Started with Quay.io](#)

[Next](#)

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.

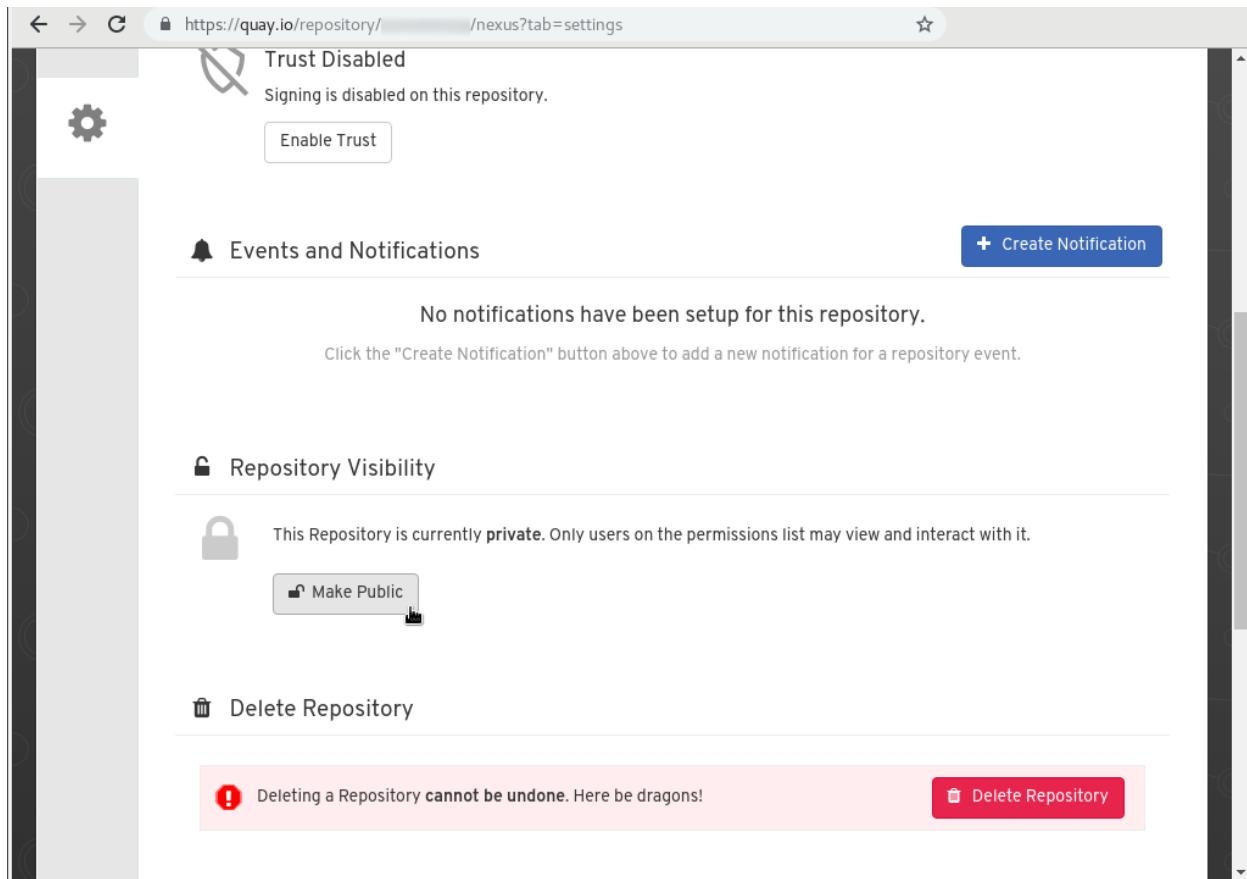
The screenshot shows the Quay.io web interface. At the top, there is a browser header with back, forward, and refresh buttons, and the URL <https://quay.io/repository/>. Below the header, the Red Hat Quay.io logo is on the left, followed by navigation links: EXPLORE, APPLICATIONS, REPOSITORIES (which is highlighted in red), and TUTORIAL. The main title "Repositories" is centered at the top of the page. On the left, there is a section titled "Starred" with a star icon. To the right of this section are two small square icons with grid and list symbols. The central area displays a message: "You haven't starred any repositories yet." Below this message is a sub-instruction: "Stars allow you to easily access your favorite repositories." A user profile section for "RHT_OCP4_QUAY_USER" is shown, featuring a small profile picture and the user's name. Below this, four repository cards are listed in a 2x2 grid:

- do180-mysql-57-rhel7
- do180-todonodejs
- do180-quote-php
- nexus

Each repository card includes a small icon representing the repository type (e.g., MySQL, Node.js, PHP, Nexus) and a star icon to its right.

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.



Get back to the list of repositories. The lock icon besides the repository name have disappeared, indicating the repository became public.

[Previous](#) [Next](#)

Appendix D. Creating a Red Hat Account

Abstract

Goal	Describe how to create a Red Hat account for labs in the course.
-------------	--

Creating a Red Hat Account

Objectives

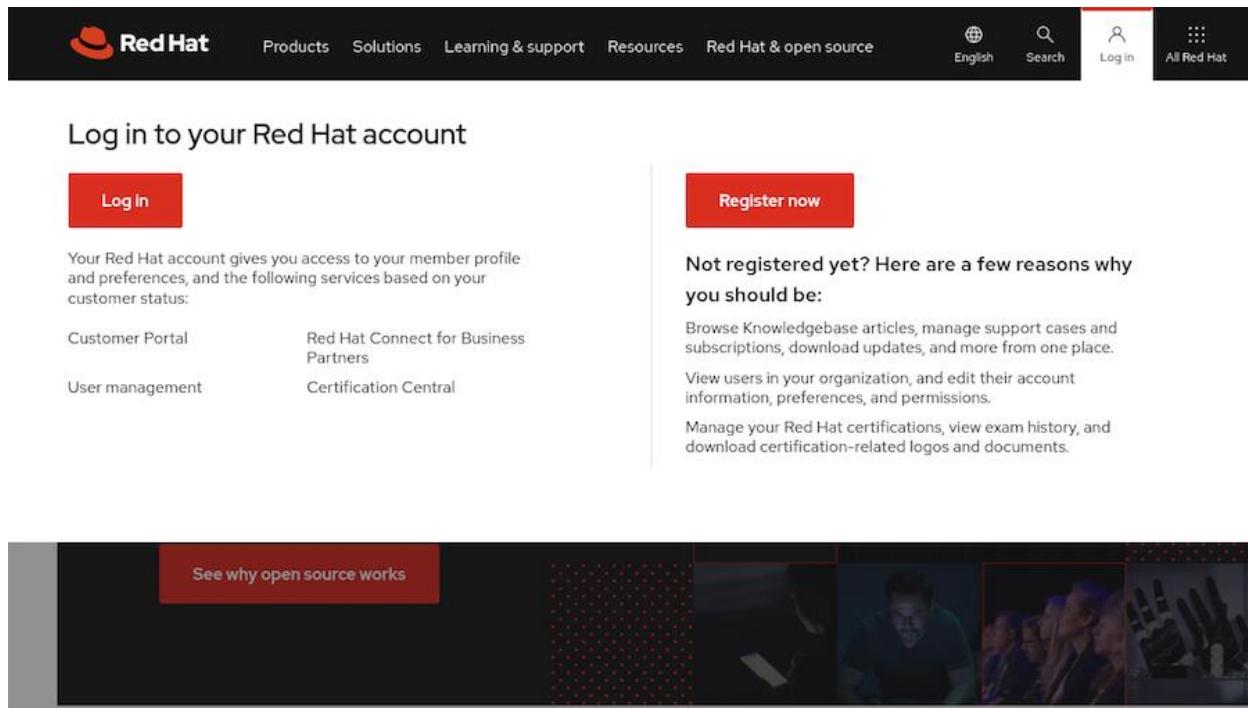
After completing this section, you should be able to create a Red Hat account to access the Red Hat Container Catalog images for the labs in the course.

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.



4. Figure D.1: Register a new account
5. Set Account type as Personal.

6. Figure D.2: Set account type
7. Fill in the remainder of the form with your personal information. In this form you also select the username and password for this account.
8. Click **CREATE MY ACCOUNT**.

The image shows a screenshot of a web-based account creation form. At the top, there are two input fields: 'City *' and 'State/Province *'. Both fields have placeholder text ('[REDACTED]') and are enclosed in light gray boxes. Below these fields is a large red rectangular button with the white text 'CREATE MY ACCOUNT'. At the bottom left of the form area, there is a blue link that reads '« Back to login page'.

9. Figure D.3: Fill in personal information form

References

[Red Hat Customer Portal](#)

[Next](#)

Appendix E. Useful Git Commands

Abstract

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:

```
2. [student@workstation ~]$ lab your-exercise finish
```

...output omitted...

3. Change to your local clone of the `D0180-apps` repository and switch to the `master` branch:

```
4. [student@workstation ~]$ cd ~/D0180-apps
```

5.

```
6. [student@workstation D0180-apps]$ git checkout master
```

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

7. Delete your local branch:

```
8. [student@workstation D0180-apps]$ git branch -d your-branch
```

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

9. Delete the remote branch on your personal GitHub account:

```
10. [student@workstation D0180-apps]$ git push origin --delete your-branch
```

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

11. Use the `start` subcommand to restart the exercise:

```
12. [student@workstation D0180-apps]$ cd ~
```

```
13.
```

```
14. [student@workstation ~]$ lab your-exercise start
```

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

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.

```
2. [student@workstation ~]$ lab your-exercise finish
```

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

3. Enter your local clone of the `D0180-apps` repository and discard any pending changes on the current branch using `git stash`:

```
4. [student@workstation ~]$ cd ~/D0180-apps
```

```
5.
```

```
6. [student@workstation D0180-apps]$ git stash
```

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

7. Switch to the `master` branch of your local repository:

```
8. [student@workstation D0180-apps]$ git checkout master
```

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

9. Delete your local branch:

```
10. [student@workstation D0180-apps]$ git branch -d your-branch
```

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

11. Delete the remote branch on your personal GitHub account:

```
12. [student@workstation D0180-apps]$ git push origin --delete your-branch
```

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

13. You can now restart the exercise by running its `start` subcommand:

```
14. [student@workstation D0180-apps]$ cd ~
```

```
15.
```

```
16. [student@workstation ~]$ lab your-exercise start
```

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

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:

```
2. [student@workstation ~]$ cd ~/D0180-apps  
3.  
4. [student@workstation D0180-apps]$ git commit -a -m "Paused at step X.Y"  
5. ...output omitted...  
6. [student@workstation D0180-apps]$ git push
```

...output omitted...

7. 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.

8. Start the next exercise by running its `start` subcommand:

```
9. [student@workstation ~]$ lab your-exercise start
```

...output omitted...

10. The next exercise switches to the `master` branch and optionally creates a new branch for its change. This means the changes made to the original exercise in the original branch are left untouched.

11. Later, after you have completed the next exercise, and you want to go back to the original exercise, switch back to its branch:

```
12. [student@workstation ~]$ git checkout original-branch
```

...output omitted...

Then you can continue with the original exercise at the step where you left off.

References

[Git branch man page](#)

[What is a Git branch?](#)

Git Tools - Stashing