RED HAT®
CONSULTING

redhat.

DO180 - Introduction to Containers, Kubernetes, and Red Hat Openshift (OCP 4.2)

Travis Michette

Version 1.0

Table of Contents

Before You Begin

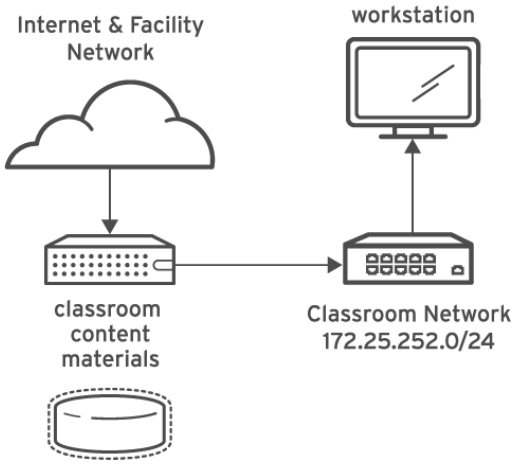

Figure 1. DO180 Classroom Layout

Table 1. Classroom Machines

Machine name	IP addresses	Role
classroom.lab.example.com	172.25.252.254, 172.25.253.254, 172.25.254.254	Classroom utility server and **content.example.com** and **materials.example.com**
workstation.lab.example.com	172.25.250.254, 172.25.252.1	Graphical workstation used for system administration

Table 2. Classroom Credentials

Username	Password
student	student
root	redhat

OpenShift and ROL Credentials

Participants will be provisioned an OpenShift 4 (OCP4) cluster with their environment. Credentials for the OCP4 environment are provided by the Red Hat Online interface and include:

- API endpoint of OCP4
- Cluster-ID
- Username
- Password

These credentials will be used to access the OpenShift environment for guided exercises and labs.

Required Accounts

This course requires that a participant have an account with the following Internet services:

- **Quay.io**: https://quay.io/
- **Github**: https://www.github.com

Before starting Chapter 1, have class look at Appendix B and Appendix C so that accounts can be setup and ready to go.

1. Introducing Container Technology

1.1. Overview of Container Technology

The goal of this section is to understand differences between container applications and traditional application deployment.

1.1.1. Containerized Applications

Many software applications depend on libraries, configuration files, and services provided by a runtime environment. These environments are traditionally physical or virtual machines with dependencies installed as part of the host.

In traditional system deployments, applications are dependent upon the host operating system and applications and upgrades to the underlying OS and system might break the application.

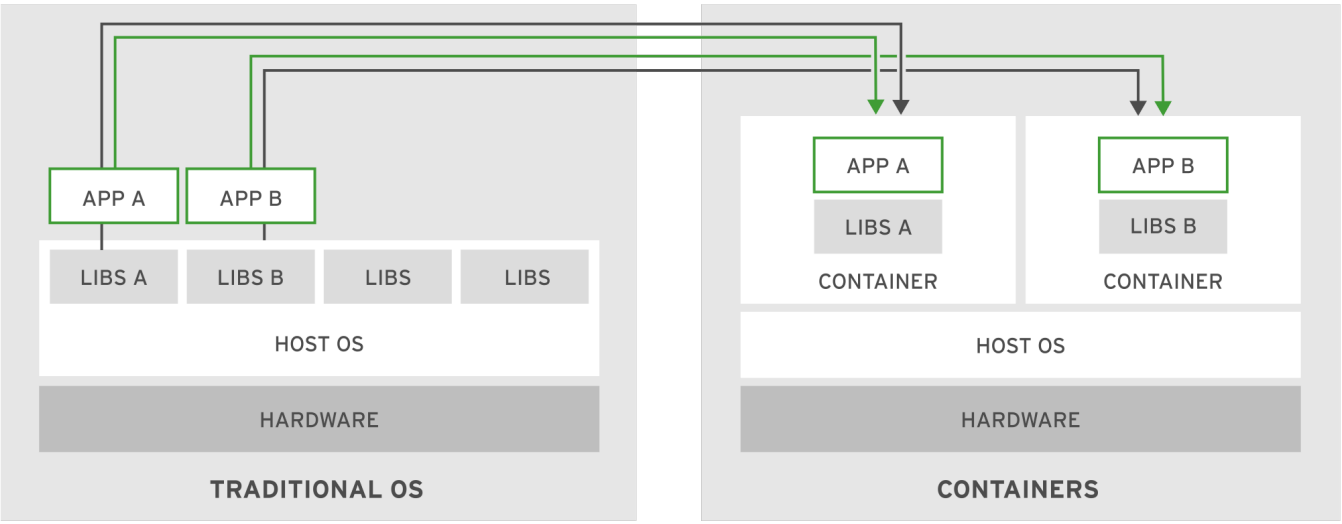

Figure 2. Container versus Operating System Differences

A newer, alternative solution is to deploy the application as a **container**. Containers are a set of one or more processes (applications and libraries) that are bundled together and isolated from the rest of the operating system and hardware. These containers provide some of the same benefits of virtual machines in that many containers can run on a single host and can leverage storage, security, and network isolation. However, containers can further separate applications by isolating other resources required by the application (runtime libraries, runtime resources, etc.) which minimizes impact of the underlying system updates breaking the application.

OCI - Open Container Initiative: Set of industry standards to define a container runtime specification and container image specification.

The container image spec defines the format for the bundle of files and metadata forming a container image. When built, the image will comply with the OCI standard and can use any OCI-compliant container engine.

Container Engines

- Rocket

- Drawbridge
- LXC
- Docker
- Podman (RHEL 7.6+)

Isolation allows the container to be portable and provides many benefits when using a container.

Container Advantages

- Low Hardware Footprint (less memory/CPU required)
- Environment Isolation
- Quick Deployment
- Multi-Environment Deployment
- Reusability (version control of images)
- Security/Stability

References

Open Containers Initiative: https://www.opencontainers.org/

1.2. Overview of Container Architecture

Goals

- Describe Linux container architecture
- Install **podman** to manage containers

1.2.1. Introducing Container History

Containers had beginnings in 2001 as the concept was introduced under a project called **VServer**. This project attempted to run a complete set of processes inside a single server. This project provided the idea of isolated processes and formed basis for the following Linux kernel features:

- **Namespaces**: Location provided by the Linux kernel to isolate specific resources which prevents these resources from being visible to all processes. By placing these resources inside a namespace, only members of the namespace can see the resources. Resources included in a namespace include: network interfaces, process ID list, mount points, IPC resources, and system hostname information.
- **Control Groups (cgroups)**: A partitioning of processes and child processes into groups providing management and allowing limits to put on the resources the group consumes.
- **Seccomp**: Limits how processes use system calls. Provides a process to whitelist system calls.
- **SELinux**: Security Enhanced Linux provides mandatory access controls (MAC) for processes. SELinux protects processes from each other and ensures they run as a confined SELinux type.

The concepts above focus on the basic concept of isolation and enabling isolation while providing access to system resources. These concepts provide the foundation for Linux containers.

1.2.2. Describing Linux Container Architecture

To the Linux Kernel, a container is a process with restrictions. A container runs an image, which is a file-system bundle containing all dependencies to execute a process. This means an image contains all files it needs including configuration files and libraries to run the process on the system.

Image Bundle

Images provide a repeatable process and allow a container to be deployed across multiple systems.

Container images are reusable and generally storage in an **image repository**. An image repository is a service which provides container images to a container runtime.

Image Repositories

- RedHatContainerCatalog[https://registry.redhat.io]
- DockerHub[https://hub.docker.com]
- RedHatQuay[https://quay.io/]
- GoogleContainerRegistry[https://cloud.google.com/container-registry/]
- AmazonElasticContainerRegistry[https://aws.amazon.com/ecr/]

This course will use the Quay image repository.

Important Header

If you haven't already created a Quay.IO account, this should be completed now as it will be used throughout the course.

RedHatQuay: https://quay.io/

Red Hat Training has created several images and placed in an image repository for this course. You will need to use Quay.IO in order to complete the Guided Exercises and End of Chapter labs.

The best way to approach the use of containers is with Microservices. It is best to break apart larger applications to have a singular function provided by several smaller pieces.

1.2.3. Managing Containers with Podman

Containers, images, and registries need to interact with each other. There are various tools out there to interact with these registries. **Podman** is an open source too for managing containers and container images as well as it allows you to interact with image registries.

Podman Feautures

- Uses image format specified by OCI
- Stores local images in local filesystem
- Uses same command structure/pattern as Docker CLI
- Compatible with Kubernetes
- Doesn't require a client/server pattern

Important Header

podman is only available on Linux systems. Podman is RPM-based and can be installed with YUM or DNF on RPM-based systems.

Listing 1. Installing Podman

```
# yum install podman
```

References

Red Hat Quay Container Registry: https://quay.io

Podman site: https://podman.io/

Open Container Initiative: https://www.opencontainers.org

1.3. Overview of Kubernetes and Openshift

Goals:

- Identify container limitations and need for container orchestration
- Describe Kubernetes container orchestration tool
- Describe Red Hat OpenShift Container Platform (RHOCP or OCP)

1.3.1. Limitations of Containers

Containers provide a quick and easy method to package and run services. However, as the number of containers grows, the complexity in system management grows.

Production Environment Requirements for Containers

- Easy communication between services
- Resource limits on applications regardless of number of containers
- Ability to respond to usage variations (spikes/decreases) and ability to adjust containers accordingly
- Resolve and react to service degradation
- Ability to support gradual rollout of new releases

Container Orchestration Needs

When used in an enterprise environment, container orchestration is needed because container runtimes like Podman and Docker don't address the above requirements.

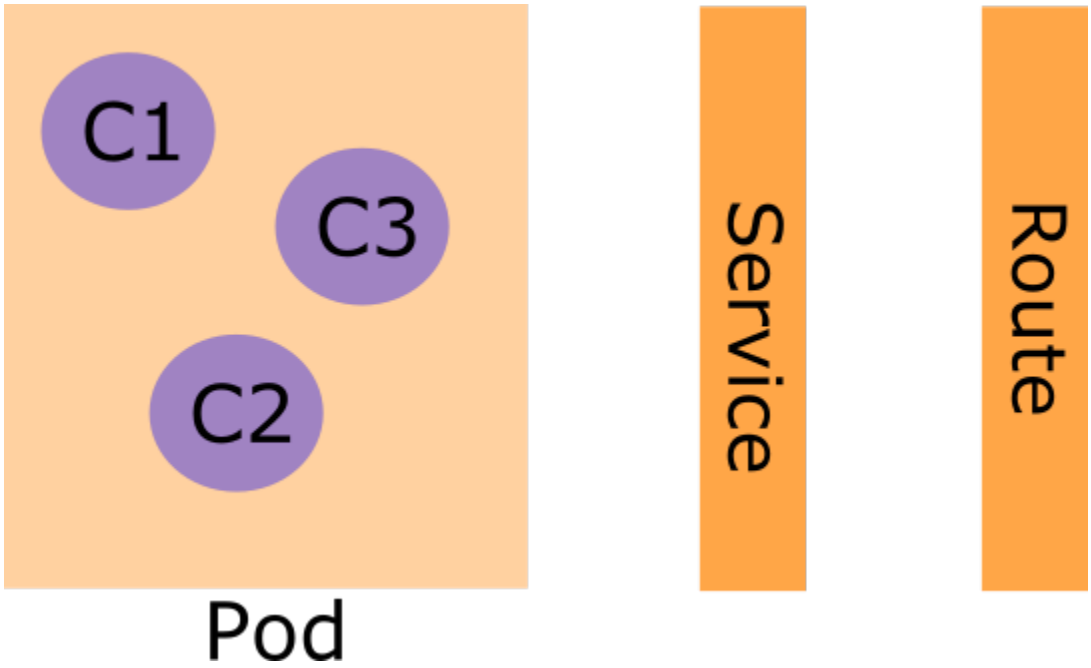

Figure 3. Accessing Containers

1.3.2. Kubernetes Overview

Kubernetes provides an orchestration service simplifying deployment, management, and scaling of containerized applications. The smallest manageable unit in Kubernets is a pod.

Pod: Consists of one r more containers with storage resources and IP addresses representing a single application.

Service: Way to load balance and discover services. The Kubernetes service directs traffic to pods.

1.3.3. Kubernetes Features

Kubernetes adds many features on top of a container infrastructure.

- **Service Discovery and Load Balancing**: Enables inter-service communication by assigning DNS entries to each set of containers
- **Horizontal scaling**: Allows applications to scale up/scale down
- **Self-Healing**: Performs health checks and monitors containers to restart and reschedule automatically in case of failure
- **Automated rollout**: Allows gradual updates to roll out while checking status. If a failure occurs, automatically rolls back to previous deployment.

- **Secrets and Configuration Management**: Allows managing application secrets without rebuilding containers
- **Operators**: Pre-packaged Kubernetes applications which can use the Kubernetes APO to update clusters states. These are typically used for applications getting feedback from the cluster.

1.3.4. OpenShift Overview

OCP is a set of modular components and services built on top of Kubernetes. OCP adds additional capababilities and management on top of the Kubernetes orchestration platform.

CoreOS

Beginning with Red Hat OpenShift 4.x (OCP4), Red Hat Linux CoreOS is used as the underlying operating system.

1.3.5. OpenShift Features

OpenShift adds several features to the Kubernetes orchestration platform with the most useful being a **route**.

- **Integrated Developer Workflow**: Provides built-in container registry, CI/CD pipelines, S2I, and a tool for building artifacts from source repos to container images
- **Route**: Allows exposing of services easily to outside world
- **Metrics and Logging** Built-in metrics service and aggregated logging functions
- **Unified UI**: Provides unified tools and UI to manage all capabilities

References

Production-Grade Container Orchestration - Kubernetes: https://kubernetes.io/

OpenShift: Container Application Platform by Red Hat, Built on Docker and Kubernetes: https://www.openshift.com/

2. Creating Containerized Services

2.1. Provisioning Containerized Services

Goals

- Search and download container images using **Podman**
- Run and configure containers locally
- Use the Red Hat Container Catalog

2.1.1. Fetching Container Images with Podman

Applications run in containers as a way to provide an isolated and controlled environment. Running the application requires a container image which provides a filesystem bundle with all application files, libraries, and dependencies needed for the application to properly run. Container images are provided by registries which allow users to find an appropriate container image to run the desired application.

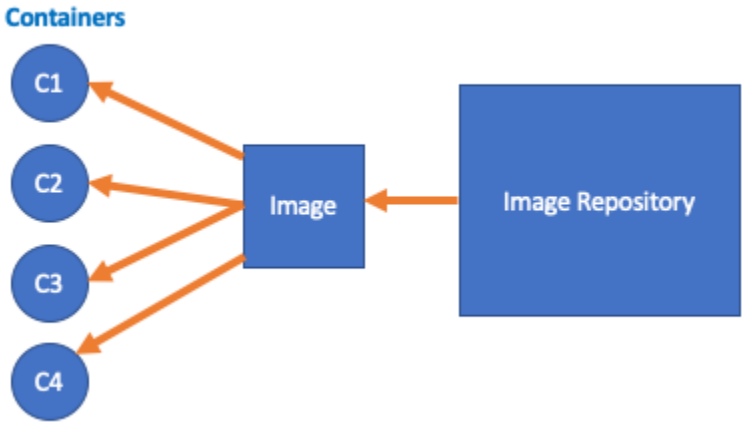

Figure 4. DO447 Classroom Layout

Podman: Allows users to search and retrieve images from remote or local registries.

Listing 2. Searching for a container image using Podman

```
[student@workstation ~]$ sudo podman search rhel
```

Listing 3. Obtaining a container image using Podman

```
[student@workstation ~]$ sudo podman pull rhel
```

Listing 4. Listing available images using Podman

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

Container Image Naming Syntax

registry_name/user_name/image_name:tag

- **Registry Name**: Name of registry storing the image.
- **User Name**: User or organization that the image belongs to
- **Image Name**: Unique name in the user namespace
- **Tag**: Identifies image version

2.1.2. Running Containers

The **podman run** command is used to run a container locally based on a specified image. The container image should specify a process to start in the container known as the **entry point**.

Listing 5. Running a Container Image

```
[student@workstation ~]$ sudo podman run ubi7/ubi:7.7 echo "Hello World!!!"
```

It is also possible to run a container image in the background as a process by passing the **-d** option to the **podman run** command.

Listing 6. Running a Container Image in the Background

```
[student@workstation ~]$ sudo podman run -d rhscl/httpd-24-rhel7:2.4-36.8

[student@workstation ~]$ sudo podman inspect -l \
> -f "{{.NetworkSettings.IPAddress}}"
```

podman inspect

The **podman inspect** command can retrieve information about a container. Specifically, when **podman inspect** is provided with the **-f**, it is possible to filter the information you want to return.

podman *Tips*

Most **podman** subcommands accept the -l flag (l for latest) as a replacement for the container id.

If the image to be executed isn't locally available when using **podman run** it will automatically be downloaded by **podman** using **podman pull**.

When referencing containers using **podman** a container can be referenced by either the **container name** or the **container id**. The **--name** option can set the container name when using **podman**.

Unique Container Names

Container names MUST be unique when specified with the **podman run** command.

It is possible that some images require user interaction with console input/output. **podman** has some **run** subcommands and

flags which will support interactivity with the end user, typically utilized by specifying **-it**.

podman *Run Sub-Commands*

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.

Listing 7. **podman** *Interactive Shell*

```
[student@workstation ~]$ sudo podman run -it ubi7/ubi:7.7 /bin/bash

bash-4.2# ls
```

Some containers require environment variables to be set in order for the container to properly initialize and run. The most common approach to provide and inject these variables is using **podman** with the **-e** flag as a **run** subcommand to specify the extra environment variables.

Listing 8. **podman** *with Runtime Variables Provided*

```
[root@workstation ~]# sudo podman run --name mysql-custom \
> -e MYSQL_USER=redhat -e MYSQL_PASSWORD=r3dh4t \
> -d rhmap47/mysql:5.5
```

2.1.3. Using the Red Hat Container Catalog

Red Hat maintains a repository of container images. The **podman** command can be used with the Red Hat Container Catalog. It is easiest to explore container catalogs using graphical utilities provided by a web browser.

Red Hat Maintained Container Registries

- https://registry.redhat.io
- https://quay.io
- https://registry.access.redhat.com

2.2. Demonstration

The following demonstration will show how to use the Universal Base Image (UBI) for RHEL8. I will allow you to see various **podman** commands in action that were demonstrated throughout the chapter. There will be a few new commands that are introduced as well.

Example 1. DEMO - Using the RHEL8 Universal Boot Image

1. Search for Containers (specifically UBI)

Listing 9. Using **podman** *to search for container images*

```
[student@workstation Demos]$ podman search ubi8
INDEX        NAME                                            DESCRIPTION                                     STARS
OFFICIAL   AUTOMATED
redhat.com   registry.access.redhat.com/ubi8                 The Universal Base Image is designed and eng...   0

... output omitted ...

quay.io      quay.io/tradisso/kogito-springboot-ubi8-s2i                                                     0
[student@workstation Demos]$
```

2. Choose and run a container accessing the **bash** shell

Listing 10. Running a Container and Accessing the Shell

```
[student@workstation Demos]$ sudo podman run -it registry.access.redhat.com/ubi8/ubi /bin/bash
```

Getting an Interactive Shell

When running a container, it is possible to pass **-it** as a command line option and then specify an interactive shell such as **/bin/bash**

3. Verify that you are running the container and accessing the container shell.

Listing 11. Verifying we are in the Container

```
[root@811b30c61a99 /]# cat /etc/redhat-release
Red Hat Enterprise Linux release 8.2 (Ootpa)
```

4. Change or Modify the Container - Install a package

Listing 12. Installing Packages in the Container

```
[root@811b30c61a99 /]# yum install httpd

... output omitted ...

  redhat-logos-httpd-81.1-1.el8.noarch

Complete!
```

5. Attempt to Enable Daemon for HTTPD with SystemD

Listing 13. Failure of **systemd** *and* **httpd** *as a Daemon*

```
[root@811b30c61a99 /]# systemctl enable httpd --now
Created symlink /etc/systemd/system/multi-user.target.wants/httpd.service → /usr/lib/systemd/system/httpd.service.
System has not been booted with systemd as init system (PID 1). Can't operate.
Failed to connect to bus: Host is down
```

Containers and Services as Daemons

The container is running, but isn't a full blown virtual machine. Therefore, the systemd functionality and init system isn't running on a back-end. For apache to run, you can use specialized HTTPD containers. In order to run for this container, you will need to use **httpd &** to run the service in the background.

6. Run the HTTP package

Listing 14. Executing Applications in the Background

```
[root@811b30c61a99 /]# httpd&
[1] 67
[root@811b30c61a99 /]# AH00558: httpd: Could not reliably determine the server's fully qualified domain name, using
fe80::5c7a:a2ff:fe6b:d180. Set the 'ServerName' directive globally to suppress this message

[1]+  Done                    httpd
```

7. Test the Webserver

Listing 15. Testing Apache HTTPD with Curl

```
[root@811b30c61a99 /]# curl http://localhost
<!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 ...

        </div>
    </body>
</html>
[root@811b30c61a99 /]#
```

8. Attempt to run old container

Listing 16. Use Podman to Launch Container

```
[student@workstation Demos]$ sudo podman run -it registry.access.redhat.com/ubi8/ubi /bin/bash
[root@604e03f02d11 /]# exit
exit
```

Container ID Changed

New container is **root@604e03f02d11 /]** and the container with HTTPD was **root@811b30c61a99**

9. List Containers

Listing 17. Podman to list containers

```
[student@workstation Demos]$ sudo podman ps -a
CONTAINER ID   IMAGE                                        COMMAND     CREATED         STATUS                     PORTS   NAMES
IS INFRA
604e03f02d11   registry.access.redhat.com/ubi8/ubi:latest   /bin/bash   2 minutes ago   Exited (0) 2 minutes ago
gallant_johnson       false
811b30c61a99   registry.access.redhat.com/ubi8/ubi:latest   /bin/bash   5 hours ago     Exited (0) 5 minutes ago
suspicious_einstein   false
```

10. Launch container with HTTPD package

Listing 18. Launching Original Container

```
[student@workstation Demos]$ sudo podman start 811b30c61a99
811b30c61a9988beca57bbee769987a43f393ec5add6f44fd84244091547b926

[student@workstation Demos]$ sudo podman exec -it 811b30c61a99 /bin/bash
[root@811b30c61a99 /]#

[root@811b30c61a99 /]# httpd &
[1] 21
[root@811b30c61a99 /]# AH00558: httpd: Could not reliably determine the server's fully qualified domain name, using
fe80::bcea:baff:fe20:ac4b. Set the 'ServerName' directive globally to suppress this message

[1]+  Done                    httpd

[root@811b30c61a99 /]# curl localhost
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">

... output omitted ...

        </div>
    </body>
</html>
[root@811b30c61a99 /]#
```

Container Management

Stopped containers don't appear as running. Stopped containers can be seen with the **podman ps -a** command. It is possible to launch/start a stopped container with the **podman start** command, but you must provide the container name/ID in order to start the container. The **podman exec** command will allow a command to be executed interactively in the container.

It is good practice to cleanup containers and images that are no longer needed.

Listing 19. Removing Containers

```
[student@workstation Demos]$ sudo podman ps -a
CONTAINER ID   IMAGE                                         COMMAND     CREATED          STATUS
PORTS   NAMES                IS INFRA
604e03f02d11   registry.access.redhat.com/ubi8/ubi:latest   /bin/bash   14 minutes ago   Exited (0) 6
minutes ago           gallant_johnson       false
811b30c61a99   registry.access.redhat.com/ubi8/ubi:latest   /bin/bash   5 hours ago      Exited (0) 2
seconds ago           suspicious_einstein   false


[student@workstation Demos]$ sudo podman rm 604e03f02d11
604e03f02d112008c0b75989055f9461fccc7db89d0efaadfbd7a2950cba9be4

[student@workstation Demos]$ sudo podman ps -a
CONTAINER ID   IMAGE                                         COMMAND     CREATED       STATUS
PORTS   NAMES                IS INFRA
811b30c61a99   registry.access.redhat.com/ubi8/ubi:latest   /bin/bash   5 hours ago   Exited (0) About a
minute ago           suspicious_einstein   false
```

References

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

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

Excellent Podman References

https://podman.io/getting-started/

https://developers.redhat.com/blog/2019/01/15/podman-managing-containers-pods/

podman *Commands Covered in this Chapter*

This chapter provided a brief overview of the **podman** command and using it to access container images to create containerized services.

Listing 20. Searching for a container image using Podman

```
[student@workstation ~]$ sudo podman search rhel
```

Listing 21. Obtaining a container image using Podman

```
[student@workstation ~]$ sudo podman pull rhel
```

Listing 22. Listing available images using Podman

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

Listing 23. Running a Container Image

```
[student@workstation ~]$ sudo podman run ubi7/ubi:7.7 echo "Hello World!!!"
```

The **podman run** command can be used with **-it** to open an interactive session with the container. It can also be used with a **-d** to run the container in the background. The **-e** option can specify environment variables as part of the **podman run** command to initialize required environment variables.

3. Managing Containers

3.1. Managing the Lifecycle of Containers

Goal: Manage the lifecycle of a container with **podman**.

3.1.1. Container Life Cycle Management with Podman

podman can be used to manage container life-cycle management. **podman** provides a set of subcommands to create and manage containers.

podman *Sub-commands*

- pull
- push
- run
- exec
- rmi
- rm
- inspect
- stop
- kill
- restart

The image below shows a summary of the most common subcommands used to change container and image states. **podman** provides additional sub-commands which can extract and obtain information about stopped and running containers.

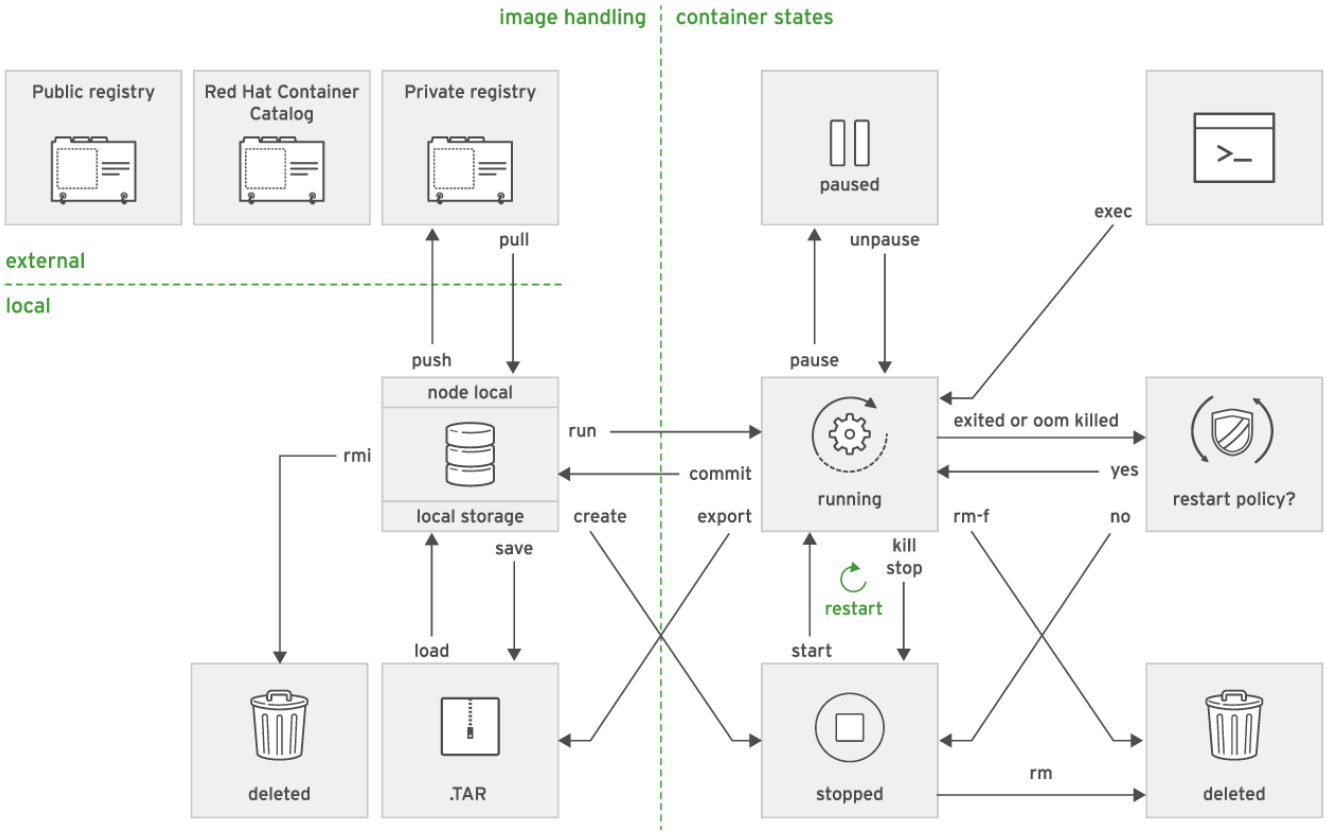

Figure 5. Podman Managing Subcommands

Listing Containers

The **podman ps -a** command is important because it returns all containers (running and stopped), whereas the **podman ps** command will only return running containers.

Additionally, **podman** provides subcommands that can query information from containers and images. The image below can be used to see subcommands which can provide information about container images as well as container states.

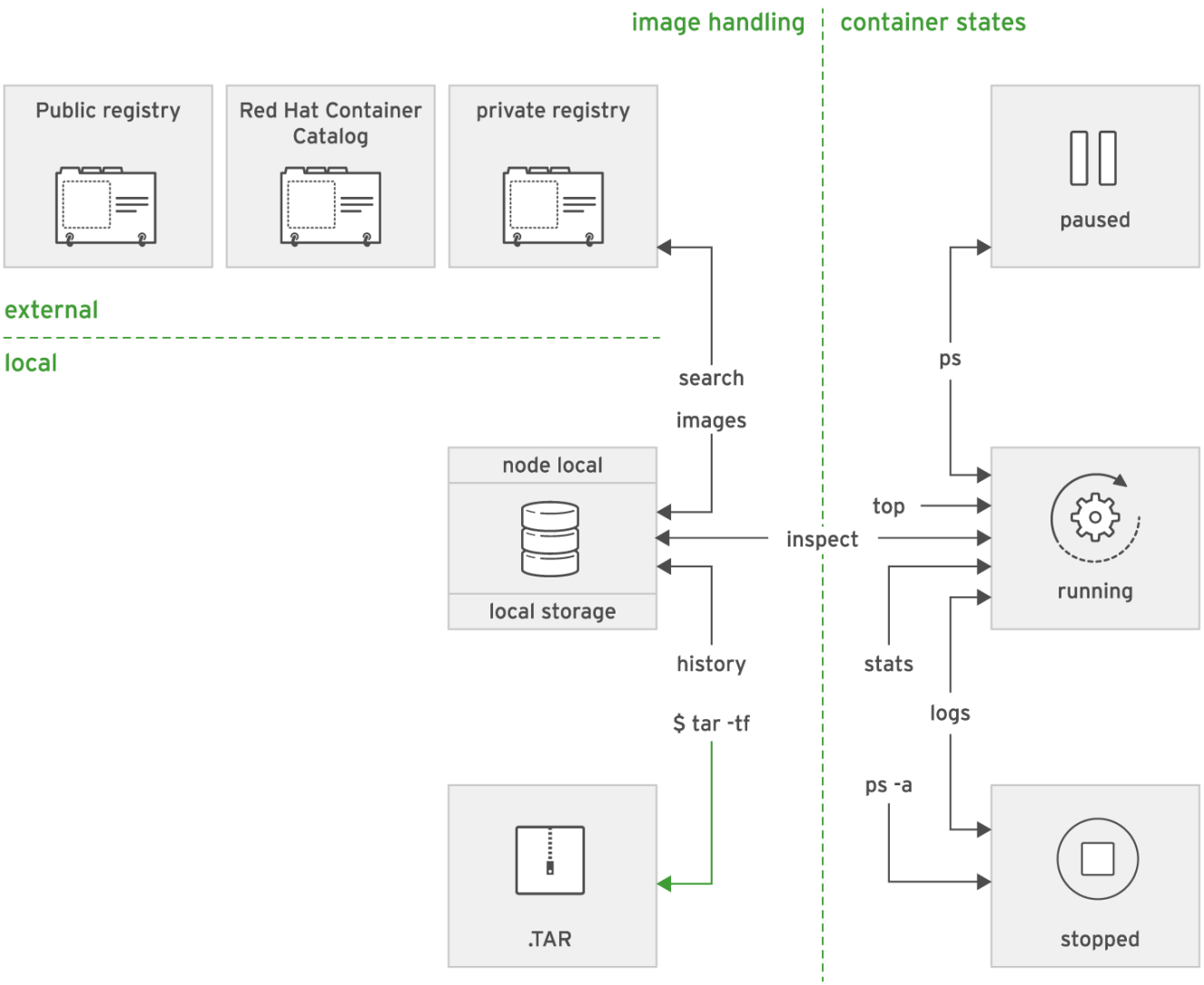

Figure 6. Podman Query Subcommands

3.1.2. Creating Containers

The **podman run** command creates a new container from an image and starts processes in the new container. If there is no image available for the container locally, **podman** will download the image.

Listing 24. Running an Apache HTPPD Container

```
[student@workstation ~]$ sudo podman run rhscl/httpd-24-rhel7
Trying to pull regist...httpd-24-rhel7:latest...Getting image source signatures Copying blob sha256:23113...b0be82
72.21 MB / 72.21 MB [======================================================] 7s ...output omitted...AH00094: Command line: 'httpd -D
FOREGROUND'
```

Listing 25. Listing Running Containers

```
[student@workstation ~]$ sudo podman ps -a
```

Listing 26. Running an Apache HTPPD Container and Specifying a Name

```
[student@workstation ~]$ sudo podman run --name travis-httpd-container rhscl/httpd-24-rhel7
Trying to pull regist...httpd-24-rhel7:latest...Getting image source signatures Copying blob sha256:23113...b0be82
72.21 MB / 72.21 MB [=====================================================] 7s ...output omitted...AH00094: Command line: 'httpd -D
FOREGROUND'
```

Unique Container Names

It is important to remember that container names must be unique. This includes the reuse of any *stopped* container names.

It is possible to run the container as a daemon process in the background using the **podman run -d** option to run the container in detached mode.

Listing 27. Running a container in the Background

```
[student@workstation ~]$ sudo podman run --name my-httpd-container -d rhscl/ httpd-24-rhel7
```

Lastly, it is possible to run the container with an interactive shell by using **podman run -it *container* /bin/bash** to start an interactive shell. The **-it** option lets **podman** know to provide an interactive terminal to the user.

Listing 28. Running a Shell from a Container

```
[student@workstation ~]$ sudo podman run -it rhscl/httpd-24-rhel7 /bin/bash

bash-4.2#
```

3.1.3. Running Commands in a Container

The **exec** subcommand allows commands to be specified and run within a container. It is important to note that the container must already be running for the **exec** command to work.

Listing 29. Executing Commands in a Container

```
[student@workstation ~]$ sudo podman exec 7ed6e671a600 cat /etc/hostname 7ed6e671a600
```

Using the Last Container Reference

The **podman exec** command can use the **-l** option to reference the last used container. This allows you to skip the container ID or container name with **podman** as the **-l** will use the previously referenced container.

3.1.4. Managing Containers

Creating and starting containers are a first step in the lifecycle of a container. It is also necessary to stop, restart, and remove

containers. In order to manage containers, the **podman ps** command can be very handy.

Listing 30. Listing Running Containers

```
[student@workstation ~]$ sudo podman ps
```

Listing 31. Listing All Containers

```
[student@workstation ~]$ sudo podman ps -a
```

Dealing with Stopped Containers

It is important to note that **podman** doesn't discard stopped containers. Instead, they are preserved for analysis. The **podman ps -a** can list all containers and then it is possible to manually delete any stopped containers.

It is possible to get additional information for a container. The **podman inspect** command will list metadata about running and stopped containers. The output of **podman inspect** is in JSON format. The **-f** option allows formatting and specifying particular output to retrieve.

Listing 32. Inspecting a Container

```
[student@workstation ~]$ sudo podman inspect my-httpd-container
```

Listing 33. Inspecting a Container to get IP Address

```
[student@workstation ~]$ sudo podman inspect \
> -f '{{ .NetworkSettings.IPAddress }}' my-httpd-container
```

There are a couple ways to stop a running container. The preferred method is **podman stop** to gracefully stop a running container. However, it is possible to use **podman kill** and **podman kill -s SIGKILL** to send the kill commands to forcefully stop a running container.

Listing 34. Gracefully Stopping a Container

```
[student@workstation ~]$ sudo podman stop my-httpd-container
```

*Listing 35. Using **kill** to stop a container*

```
[student@workstation ~]$ sudo podman kill my-httpd-container
```

It is possible to restart a stopped container with the same container state and filesystem. The **podman restart** command creates a new container with the same container ID and filesystem.

Listing 36. Restarting a Container

```
[student@workstation ~]$ sudo podman restart my-httpd-container
```

To complete container lifecycle management, it is necessary to remove the container. This can be done using **podman rm** to remove the container. It should be noted that **podman rm** deletes the container and discards both the container state and filesystem.

Listing 37. Removing a Container

```
[student@workstation ~]$ sudo podman rm my-httpd-container
```

Removing Running Containers

It is important to note that containers must be stopped before using **podman rm**. It is possible to forcefully remove containers by specifying the **-f** option to **podman rm**. This will forcefully stop the running container and remove it as a single command. This is equivalent to running **podman kill** and **podman rm**.

Stopping and Removing Multiple Containers

It is possible to stop and remove all containers by specifying **-a** option to **podman stop** and **podman rm**.

Listing 38. Stopping all Containers

```
[student@workstation ~]$ sudo podman stop -a
```

Listing 39. Removing all Containers

```
[student@workstation ~]$ sudo podman rm -a
```

Subcommand Syntax and Options

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

3.2. Demonstration - Container Lifecycles

Example 2. DEMONSTRATION - Container Lifecycles

1. Locate HTTPD Container to run

*Listing 40. Use **podman** to find an apache container*

```
[student@workstation Demos]$ sudo podman search httpd
INDEX        NAME                                                    DESCRIPTION
STARS   OFFICIAL   AUTOMATED
redhat.com   registry.access.redhat.com/rhscl/httpd-24-rhel7         Apache HTTP 2.4 Server

... output omitted ...
```

2. Run apache container

Listing 41. Running container with **podman run** *and specifying a* **name**

```
[student@workstation ~]$ sudo podman run --name HTTPD-Demo1 redhattraining/httpd-parent:2.4
AH00558: httpd: Could not reliably determine the server's fully qualified domain name, using 10.88.100.121. Set the 'ServerName'
directive globally to suppress this message
```

3. Verifying container statuses

Listing 42. Using **podman ps -a** *to see all containers*

```
[student@workstation Demos]$ sudo podman ps -a
CONTAINER ID   IMAGE                                                COMMAND                  CREATED          STATUS
PORTS   NAMES                IS INFRA
778d7093eb8c   registry.access.redhat.com/rhscl/httpd-24-rhel7:latest   container-entrypoin...   46 seconds ago   Exited (0) 7
seconds ago           HTTPD-Demo1           false
811b30c61a99   registry.access.redhat.com/ubi8/ubi:latest               /bin/bash                24 hours ago     Exited (0) 19
hours ago            suspicious_einstein   false
```

4. Removing the **HTTPD** Demo Container

Listing 43. Using **podman rm** *to remove stopped containers*

```
[student@workstation Demos]$ sudo podman rm HTTPD-Demo1
```

5. Run as HTTPD container as a daemon in background

Listing 44. .Running container with **podman run** *and specifying a* **name** *as well as the* **-d** *to run as daemon*

```
[student@workstation ~]$ sudo podman run --name HTTPD-Demo1 -d redhattraining/httpd-parent:2.4
07d19e01c64d6cd44db8a9ddbdde341119b148078ec14684e9bea4a18b521550
```

6. Inspecting running containers

Listing 45. Using **podman inspect** *to get container details*

```
[student@workstation Demos]$ sudo podman inspect HTTPD-Demo1
[
    {
        "ID": "a874387ac7ee8bbce0b65badedd5db6b5e608ca21e9e9f294a957056fb4b11b7",

... output omitted ...

            "GlobalIPv6PrefixLen": 0,
            "IPAddress": "10.88.100.111",
            "IPPrefixLen": 16,
            "IPv6Gateway": "",

... output omitted ...
```

Filtering/Formatting with Go Templates

It is possible to use the **-f** option to utilize a formatting template to filter and retrieve a single piece of desired information.

Listing 46. Using **grep** *to filter/format results*

```
[student@workstation Demos]$ sudo podman inspect HTTPD-Demo1 | grep -i ipaddress
            "SecondaryIPAddresses": null,
            "IPAddress": "10.88.100.111",
```

Listing 47. Using **podman inspect -f** *options*

```
[student@workstation Demos]$ sudo podman inspect -f '{{ .NetworkSettings.IPAddress }}' HTTPD-Demo1
10.88.100.111
```

7. Open interactive shell in container

Listing 48. Using **podman exec** *to get a shell*

```
[student@workstation Demos]$ sudo podman exec -it HTTPD-Demo1 /bin/bash
bash-4.2$ cat /etc/redhat-release
Red Hat Enterprise Linux Server release 7.8 (Maipo)

bash-4.4# curl localhost
Hello from the httpd-parent container!
```

8. Stop the container

Listing 49. Use **podman stop** *to stop the running container*

```
[student@workstation Demos]$ sudo podman stop HTTPD-Demo1
a874387ac7ee8bbce0b65badedd5db6b5e608ca21e9e9f294a957056fb4b11b7
```

9. Delete the container

Listing 50. Use **podman rm** *to Delete the container*

```
[student@workstation Demos]$ sudo podman rm HTTPD-Demo1
a874387ac7ee8bbce0b65badedd5db6b5e608ca21e9e9f294a957056fb4b11b7
```

Cleanup of container images

It is important to cleanup the actual container images if the there are no containers using the images and there are no plans to use the container image for other containers. This will conserve disk space on the system.

Listing 51. Use **podman rmi** *to remove a container image*

```
[student@workstation Demos]$ sudo podman rmi rhscl/httpd-24-rhel7
a0cb054ab975a1d93ffff3ff932052ce5f92d7bff0a918f9f3a9f430f83a3acd
```

3.3. Attaching Persistent Storage to Containers

Goals

- Save application data across containers by using persistent storage
- Configure host directories for use as container volumes
- Mount a volume inside a container

3.3.1. Preparing Permananent Storage Locations

By default, container storage is ephemeral. That means that no data is preserved when containers are stopped and removed. Each running container gets a new layer over the base container image for the container storage. The storage is read/write available for the container and should be considered volatile. When the container is removed, the storage layer is deleted and anything on the ephemeral storage is lost. This is fine for several scenarios, however, ephemeral only storage is not sufficient for all containerized application and there is a need to provide persistent storage to containers.

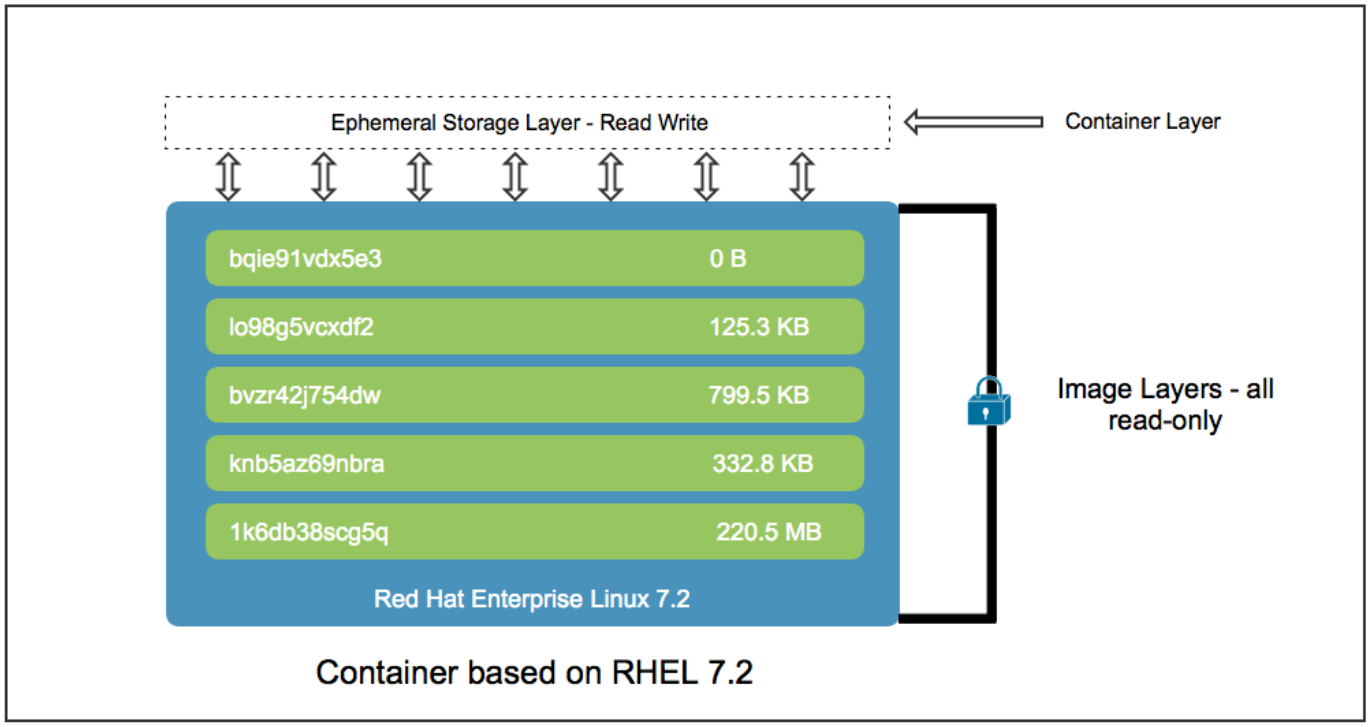

Figure 7. Container Storage Layers

Important Header

All storage layers in the image are read-only. At runtime, a container gets a new layer of ephemeral storage which is read/write non-persistent storage. This allows the container to write temporary files and have full access to the filesystem area while the container is running.

3.3.2. Reclaiming Storage

podman doesn't delete or remove stopped containers. Instead these container images are storage for later review. If an administrator needs to reclaim storage space and remove a container, the **podman rm container_ID** command must be run. This will remove the container and delete the storage associated with the container. Keep in mind, this doesn't delete the original image that was used to create the container.

3.3.3. Preparing the Host Directory

Persistent storage can be achieved by mounting host directories inside a running container. The application in the container sees the host directories as part of the container storage and has full access to these directories. It is important to note that SELinux should be running as this provides a large portion of the built-in container security. In order for the container to access and utilize the storage, the directory must be properly configured with the correct ownership and group permissions as well as the correct SELinux context **container_file_t**.

podman users **container_file_t** SELinux context to restrict files that the container can access from the host system. It further requires that the UID/GID be numerically set based on the required permissions from the applicaiton running in the container.

Important Header

It is important to remember that after using **semanage** to define the SELinux contexts that you run a **restorecon -Rv /Dir_Being_Used** to restore the SELinux contexts on the /Dir_Being_Used directory.

3.3.4. Mounting a Volume

After creating and defining the directory, it is necessary to provide **podman** the options to mount and bind the shared filesystem to the container. In order to bind and mount a host directory, the **-v** option should be used by the **podman run** command. This option requires both the host directory path and the container storage path separated by a colon (**:**).

Listing 52. Sample Mounting Syntax

```
# podman run -v /host_path:/container_path container_image_name
```

3.4. Demonstration - Attaching Persistent Storage to Containers

Example 3. Demonstration with Persistent Storage

The purpose of this demonstration is to provide persistent storage to a container. We will be mounting local storage to the container as the web host storage.

1. Prepare directory to host local storage and copy files

Listing 53. Create a Directory

```
[student@workstation Chapter3]$ sudo mkdir /Webhosting
```

*Listing 54. Copy files to **/Webhosting** directory*

```
[student@workstation Chapter3]$ sudo cp Website/* /Webhosting
```

2. Change ownership and set SELinux permissions on the directory and files.

Listing 55. Set Ownership and Permissions

```
[student@workstation Chapter3]$ sudo chown -R 48:48 /Webhosting
```

Listing 56. Set and Restore SELinux Contexts

```
[student@workstation Chapter3]$ sudo semanage fcontext -a -t container_file_t '/Webhosting(/.*)?'

[student@workstation Chapter3]$ sudo restorecon -Rv /Webhosting/
restorecon reset /Webhosting context unconfined_u:object_r:default_t:s0->unconfined_u:object_r:container_file_t:s0
restorecon reset /Webhosting/index2.html context unconfined_u:object_r:default_t:s0->unconfined_u:object_r:container_file_t:s0
restorecon reset /Webhosting/index3.html context unconfined_u:object_r:default_t:s0->unconfined_u:object_r:container_file_t:s0
restorecon reset /Webhosting/index.html context unconfined_u:object_r:default_t:s0->unconfined_u:object_r:container_file_t:s0
```

3. Start Apache container

*Listing 57. Using **podman run** to start a container with persistent storage*

```
[student@workstation Chapter3]$ sudo podman run --name HTTPD-Demo2 -v /Webhosting:/var/www/html -d redhattraining/httpd-parent:2.4

Trying to pull registry.access.redhat.com/rhscl/httpd-24-rhel7:latest...Getting image source signatures
Copying blob sha256:a03401a44180b6581a149376d6fd2d5bd85d938445fd5b5ad270e14ddde4937c

... output omitted ...
```

4. Show website from container

Listing 58. Attempt to connect to container

```
[student@workstation Chapter3]$ curl http://localhost
curl: (7) Failed connect to localhost:80; Connection refused
```

Network traffic not setup

At this point, there is no network connection so it is expected that we don't have any connection.

In order to verify the files exist, you must open a bash shell and look for the files from the mounted directory.

Listing 59. Verifying Mounted Drive and Files

```
[student@workstation Chapter3]$ sudo podman exec -it HTTPD-Demo2 /bin/bash
bash-4.2$ ls -alR /var/www/html
/var/www/html:
total 12
drwxr-xr-x. 2 apache  apache 62 Jul 16 23:10 .
drwxr-xr-x. 4 default root   33 Jul  1 12:43 ..
-rw-r--r--. 1 apache  apache 35 Jul 16 23:10 index.html
-rw-r--r--. 1 apache  apache 36 Jul 16 23:10 index2.html
-rw-r--r--. 1 apache  apache 36 Jul 16 23:10 index3.html
```

*Listing 60. Verify **apache** Service and Website*

```
bash-4.4# curl localhost
This is the Demo2 Index.html file.

bash-4.4# curl http://localhost/index2.html
This is the Demo2 Index2.html file.

bash-4.4# curl http://localhost/index3.html
This is the Demo2 Index3.html file.
```

3.5. Accessing Containers

Goals

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

3.5.1. Introducing Networking with Containers

Networking is possible by the Container Networking Interface (CNI) project. This project was created to standardize network interface for containers in cloud native, Kubernetes, and OCP environments. The CNI project uses software-defined networking (SDN) for containers on each host. Podman attaches virtual bridges to containers as well as a private IP address. The CNI settings are defined for **podman** in the **/etc/cni/net.d/87-podman-bridge.conflist** file.

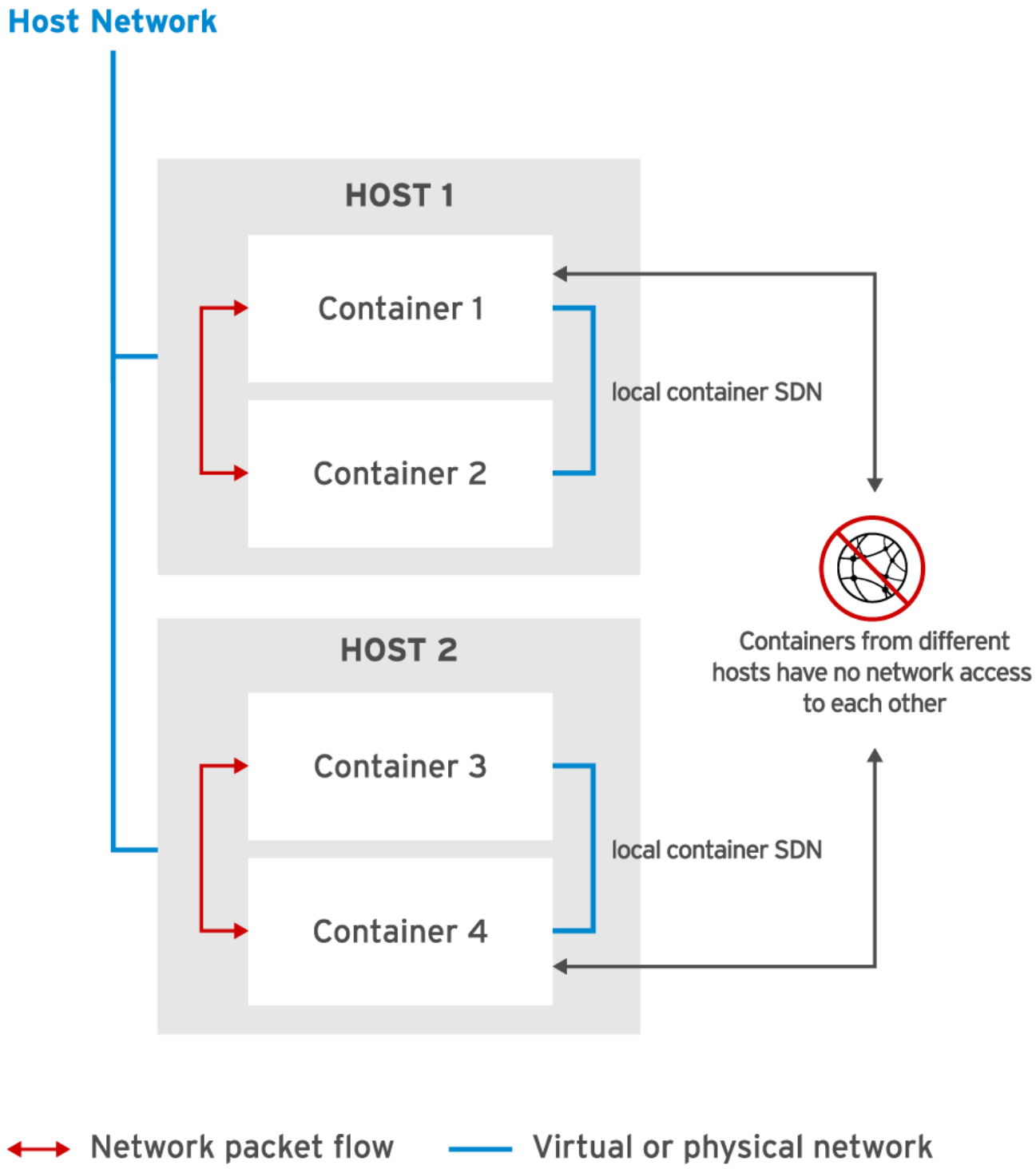

Figure 8. Basic Linux Container Networking

podman creates containers on a host and assigns each host a unique IP address connecting them all to the same SDN. Containers on the same host can communicate freely by IP address. Containers created by **podman** on other hosts belong to a different **SDN** and are generally prevented from interacting with containers running on a remote host because SDN isolates containers to the locally defined SDN preventing communication between different networks.

Important Header

By default, all container networks are hidden from the host network. SDN provides complete network isolation for the containers one the host with containers on a different host. This isolation also allows a container in one SDN to have the same IP address in a different SDN.

3.5.2. Mapping Network Ports

Accessing containers from the host network must be specifically granted using the SDN commands. In order to solve the problem of allowing access to a container network, the **-p** option can be used to allow external access through port forwarding. Specifically, when using **podman**, you would specify the **-p [<IP address>:][<host port:]<container port>** option when using the **podman run** command.

Note Header

Container IP addresses are assigned from an IP address pool. When a container is deleted, the IP address is returned to the pool and becomes available for another container. Because the IP addresses are reused and randomly assigned, port forwarding is the easiest method to allow access to a container application using the network.

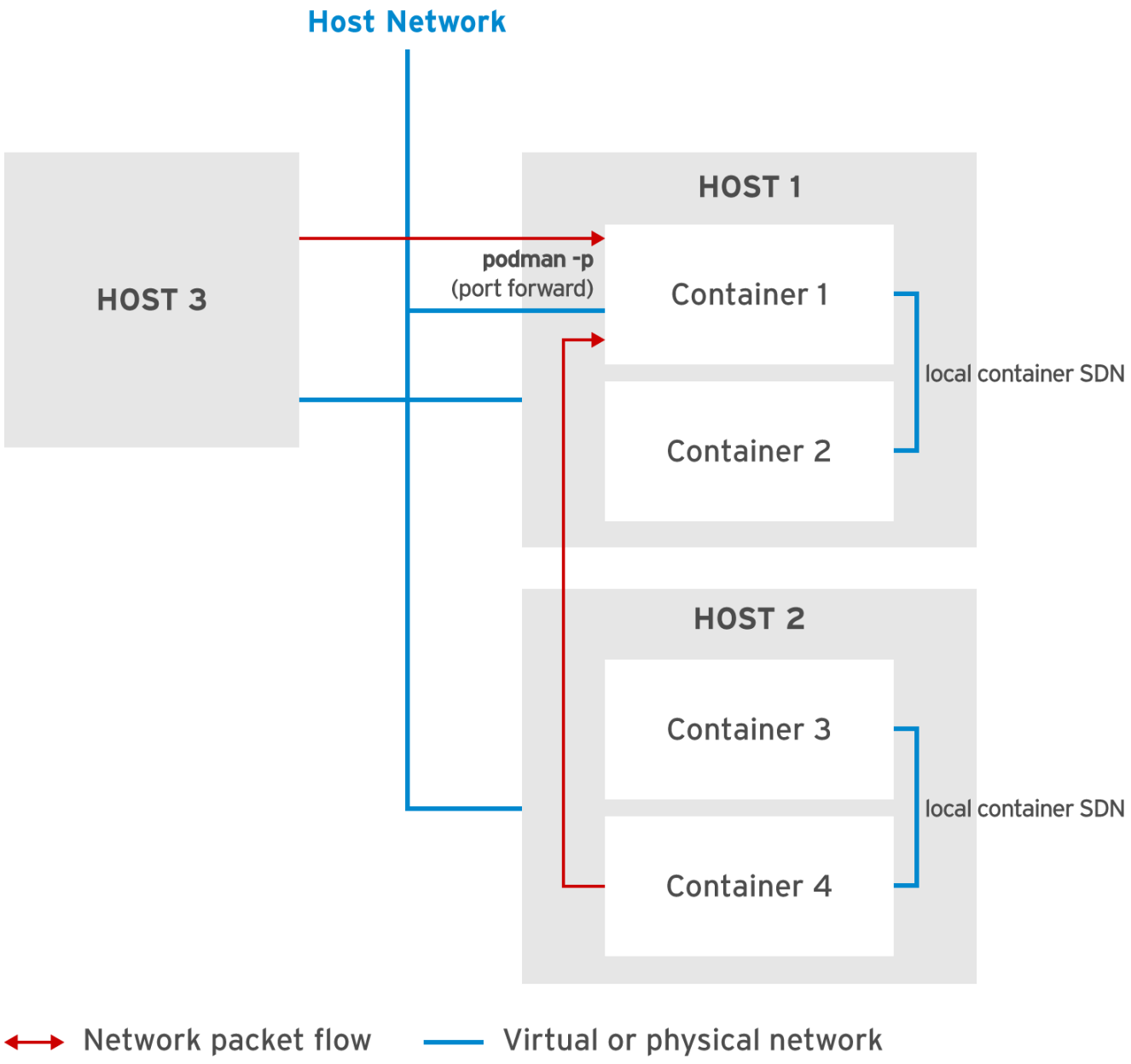

Figure 9. External Access to Linux Containers

It is possible to specify the ports for forwarding. However, it is also possible to port forward requests only if the requests originate from a specific and specified IP address. The **-p** option is capable of performing both port forwarding options.

3.6. Demonstration - Accessing Containers over the Network

Example 4. Demonstration - Allowing Network Ports

Building from Demonstration #2, we will restart the container with the network port forwarding.

1. Start the container as Demo3 with network port forwarding.

Listing 61. Use **podman run** *with the* **-p** *option to port forward*

```
[student@workstation ~]$ sudo podman run --name HTTPD-Demo3 -v /Webhosting:/var/www/html -p 80:80 -d redhattraining/httpd-
parent:2.4
4b2c8dc1f89ffb22071d0d171b2c33838dd09f5da6d41c4d9c56bc904b91284b
```

2. Verify port has been opened

Listing 62. Source Description

```
[student@workstation ~]$ curl localhost
This is the Demo2 Index.html file.

[student@workstation ~]$ curl http://localhost/index2.html
This is the Demo2 Index2.html file.

[student@workstation ~]$ curl http://localhost/index3.html
This is the Demo2 Index3.html file.
```

References

Container Network Interface - networking for Linux containers: https://github.com/containernetworking/cni **Cloud Native Computing Foundation**: https://www.cncf.io/

4. Managing Container Images

4.1. Accessing Registries

Goals

- Search for and pull images from remote registries using Podman commands and the registry API
- Customize Podman configuration to access alternative container image registries
- List images downloaded from registries to the local file system
- Manage tags to pull tagged images

4.1.1. Public Registries

Public registries are the most likely registry to use as a download source for containers. Image registries provide container images to download and allow image creators/maintainers to store and distribute container images to larger audiences.

podman can be used as a search tool for both public and private image registries. The Red Hat Container Catalog is the public image registry maintained by Red Hat. Another Red Hat container registry is known as Quay.IO which is a public image registry containing user created images.

Red Hat Container Image Benefits

- **Trusted Source**: All images come from known sources and trusted by Red Hat
- **Original Dependencies**: No container packages have been altered and only include known and required libraries
- **Vulnerability-Free**: Images are free from known vulnerabilities in platform layer components
- **Runtime Protection**: All images run as non-root users
- **Red Hat Enterprise Linux (RHEL) Compatible**: Images compatible with all RHEL platforms
- **Red Hat Support**: Images are commercially supported by Red Hat

Important Header

Keep in mind, **Quay.io** is a public image repository maintained by Red HAt. However, these images are not verified by Red Hat like with Red Hat Container Catalog. Quay.io allows users to create and publish their own images, so be aware there could be potential issues.

4.1.2. Private Registries

Private registries give image creators control about image placement, distribution, and usage. A private registry works the same way as a public registry except the administrators have full control.

4.1.3. Configuring Registries in Podman

The registries for the **podman** command are configured in the **/etc/containers/registries.conf** file. The **[registries.search]** section contains registry entries.

Listing 63. Sample of Registry Sources from **/etc/containers/registries.conf**

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

Specifying Registries in the **/etc/containers/registries.conf** *file.*

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.

Connections to registries require a trusted certificate. It is possible to support insecure connections by modifying the **[registries.insecure]** section of **/etc/ containers/registries.conf** file.

Listing 64. Allowing Insecure Registries

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

4.1.4. Accessing Registries

The **podman search** command is capabable of searching registries defined by **/etc/ containers/registries.conf** for images to run as containers.

Listing 65. **podman search** *Syntax*

```
[student@workstation ~]$ sudo podman search [OPTIONS] <term>
```

Table 3. **podman search** *Sub-commands*

Option	Description
--limit <number>	*Limits the number of listed images per registry.
--filter <filter=value>	Filter output based on conditions provided. Supported filters are • **stars=<number>**:Showonlyimageswith at least this number of stars. • **is-automated=<true/false>**:Show only images automatically built. • **is-official=<true/false>**:Showonly images flagged as official.
*--tls-verify <true	false>*

4.1.4.1. Registry HTTP API

Remote registries expose web services APIs to the registry. Many of these registries conform to **Docker Registry HTTP API v2** specifications which expose a standardized REST interface for interactions with the registry.

4.1.4.2. Registry Authentication

Some public and private registries require authorization. The **podman login** command allows a user to specify a UN/PW combination for logging into a registry.

Listing 66. Using **podman login** *to Login to a Registry*

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

4.1.4.3. Pulling Images

The **podman pull** command is used to obtain images from a registry that were located with the **podman search** command. The **podman pull** subcommand also supports adding the registry name to the image to specify exactly where you wish to obtain the image.

Listing 67. **podman pull** *Syntax*

```
[student@workstation ~]$ sudo podman pull [OPTIONS] [REGISTRY[:PORT]/]NAME[:TAG]
```

Listing 68. **podman pull** *Syntax from Specific Registry*

```
[student@workstation ~]$ sudo podman pull quay.io/bitnami/nginx
```

podman search *and* **podman pull** *Registry Order*

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

4.1.4.4. Listing Local Copies of Images

All container images obtained from a container registry are downloaded and locally stored on the host running **podman**. This allows **podman** to store images for later use. It is possible to list images using the **podman images** command to list all locally stored images.

Listing 69. Using **podman images** *to List Downloaded Images*

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

podman *Image Storage*

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

4.1.4.5. Image Tags

Image tags are ways to support multiple releases of a single image. The image tag can be used to support multiple versions of the same software for container images. The image tag can be provided to the end of the **podman pull** command using a (**:**).

Listing 70. Using **podman pull** *and* **podman run** *to Run a Specific Image*

```
[student@workstation ~]$ sudo podman pull rhscl/mysql-57-rhel7:5.7

[student@workstation ~]$ sudo podman run rhscl/mysql-57-rhel7:5.7
```

The **latest** *Image Tag*

It is important to note differences between images with multiple versions. If no tag is specified the image will default to the image with the **latest** tag.

4.2. Demonstration - Accessing and Searching Registries

Example 5. Demonstration - Accessing and Searching Registries

1. Logging into a Registry

Listing 71. Registry Login without Password in History

```
[student@workstation Demos]$ sudo podman login -u tmichett quay.io
Password:
Login Succeeded!
```

Password with **-p**

When using the **-p** the password is specified on the command line. This will end up in the BASH history so in secure environments, you might not want to use that option.

```
[student@workstation Demos]$ sudo podman login -u tmichett -p Secret_Password_Text quay.io
Login Succeeded!
```

Leaving out **-u** *and* **-p**

It is possible to use **podman login** without the **u** username option and the **-p** password option. If this is the method used, it will issue an interactive prompt for the username and the password. This will prevent both the username and password from appearing in BASH history.

Listing 72. Interactive Login

```
[student@workstation Demos]$ sudo podman login quay.io
Username (tmichett): tmichett
Password:
Login Succeeded!
```

2. Using **podman** to list images

Listing 73. Use **podman images** *to list container images stored locally*

```
[student@workstation Demos]$ sudo podman images
REPOSITORY                                      TAG      IMAGE ID       CREATED         SIZE
registry.access.redhat.com/ubi8/ubi             latest   7923da9ba983   10 days ago     212MB
registry.access.redhat.com/rhscl/httpd-24-rhel7 latest   a0cb054ab975   2 weeks ago     328MB
localhost/nexus                                 latest   b7d5e59cdc5e   3 months ago    550MB
localhost/do180/icrm2                           latest   9692b730e6f5   3 months ago    252MB
quay.io/tmichett/do180/icrm2                    v2.2     9692b730e6f5   3 months ago    252MB
quay.io/tmichett/icrm2                          v2.2     9692b730e6f5   3 months ago    252MB
localhost/do180/icrm                            latest   166e944a04ac   3 months ago    252MB
registry.access.redhat.com/ubi7/ubi             7.7      0355cd652bd1   4 months ago    215MB
localhost:5000/httpd-demo                       latest   3639ce1374d3   13 months ago   236MB
localhost/httpd-demo                            latest   3639ce1374d3   13 months ago   236MB
quay.io/tmichett/do180-demos/httpd-demo         latest   3639ce1374d3   13 months ago   236MB
quay.io/tmichett/do180-demos/httpd-demo         v3       3639ce1374d3   13 months ago   236MB
quay.io/redhattraining/httpd-parent             2.4      3639ce1374d3   13 months ago   236MB
registry.access.redhat.com/rhel7.4              latest   33a3ad89f9ab   2 years ago     206MB
```

3. Examine Config File for Container storage

Listing 74. Examining **/etc/containers/storage.conf**

```
[student@workstation ~]$ sudo vim /etc/containers/storage.conf
 storage.conf is the configuration file for all tools
# that share the containers/storage libraries
# See man 5 containers-storage.conf for more information
# The "container storage" table contains all of the server options.
[storage]

# Default Storage Driver
driver = "overlay"

# Temporary storage location
runroot = "/var/run/containers/storage"

# Primary Read/Write location of container storage
graphroot = "/var/lib/containers/storage"

[storage.options]
# Storage options to be passed to underlying storage drivers

# AdditionalImageStores is used to pass paths to additional Read/Only image stores
# Must be comma separated list.
additionalimagestores = [
]

... output omitted ...

# xfs_nospace_max_retries specifies the maximum number of retries XFS should
# attempt to complete IO when ENOSPC (no space) error is returned by
# underlying storage device.
# xfs_nospace_max_retries = "0"
```

4. Look at the **registries.conf** file

Listing 75. Examinging Configured Registries

```
[student@workstation ~]$ vim /etc/containers/registries.conf
# This is a system-wide configuration file used to
# keep track of registries for various container backends.
# It adheres to TOML format and does not support recursive
# lists of registries.

# The default location for this configuration file is /etc/containers/registries.conf.

# The only valid categories are: 'registries.search', 'registries.insecure',
# and 'registries.block'.

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

# If you need to access insecure registries, add the registry's fully-qualified name.
# An insecure registry is one that does not have a valid SSL certificate or only does HTTP.
[registries.insecure]
registries = ["sat6.michettetech.com:5000/michettetech-red_hat_training_containers-httpd-parent"]

... output omitted ...

# If you need to block pull access from a registry, uncomment the section below
# and add the registries fully-qualified name.
#
# Docker only
[registries.block]
registries = []
```

References

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

Quay.io: https://quay.io

Docker Registry HTTP API V2: https://github.com/docker/distribution/blob/master/docs/spec/api.md

4.3. Manipulating Container Images

Goals

- 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 fro distribution purposes

This section will cover four (4) ways to create and share images.

4.3.1. Introduction

There are two methods for transferring container images to other hosts.

Image Transfer Methods

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

4.3.2. Saving and Loading Images

Existing images can be saved to a **.tar** file using the **podman save** command. This is a specialized TAR file containing image metadata and preserving original image layers. By using **podman save** to save the image as a TAR file, **podman** can recreate the original image exactly.

Listing 76. **podman save** *Syntax*

```
[student@workstation ~]$ sudo podman save [-o FILE_NAME] IMAGE_NAME[:TAG]
```

The **podman save** command uses the **-o** option to designate the output file. It is possible to save a file from a container registry to a TAR file using the **podman save** command.

Listing 77. Using **podman save** *to capture an image from Red Hat Container Catalog and Save Locally*

```
[student@workstation ~]$ sudo podman save \
> -o mysql.tar registry.access.redhat.com/rhscl/mysql-57-rhel7:5.7
```

Image files saved locally with **podman save** can be restored with the **podman load** command.

Listing 78. Loading an image file from **podman save**

```
[student@workstation ~]$ sudo podman load [-i FILE_NAME]
```

podman load *Warning*

If the **TAR** file provided to **podman load** is not a container image with metadata, the command fails.

Saving and Compressing a Container Image

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.

4.3.3. Deleting Images

Podman keeps all downloaded images on local storage. Even images not being used on a container remain in local storage. The **podman rmi** command is capable of removing images. This command can accept both the *name* or the *image ID* of the image.

Listing 79. Removing an Image

```
[student@workstation ~]$ sudo podman rmi [OPTIONS] IMAGE [IMAGE...]
```

Image Updates and Changes

Images must be removed and pulled to guarantee they are the latest version of the image. Any updates to images in a registry are not updated automatically to locally stored images. Additionally, images currently being used by a container cannot be removed.

It is possible to use the **--force** option to force the removal of the image. This option will remove an image regardless if it is being used by a container and regardless of how many containers might be using the image. The **podman rmi --force** will stop and remove all containers forcefully and will then remove the image.

4.3.4. Deleting all Images

It is possible to delete all images not being used by a container by using **podman rmi -a** command. This will delete all images not currently being used by any container.

Listing 80. Using **podman rmi -a** *to Remove all Images*

```
[student@workstation ~]$ sudo podman rmi -a
```

The **podman rmi -a** command will return all image IDs available in local storage and pass them to **podman rmi** for removal.

4.3.5. Modifying Images

A **Dockerfile** is the ideal way to modify and create custom, clean images. A **Dockerfile** allows creation of a clean, lightweight image set of layers without added log files, temporary files, or other artifacts that can be created during custom image file creation.

An alternative to using a **Dockerfile** is using **podman commit** to save changes made to a running container and save those layers to create a new container image.

Warning about **podman commit**

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 **Dockerfile** 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.

Use of **podman commit**

The **podman commit** command takes a few options as described in the table below.

Table 4. **podman commit** *Options*

Option	Description
--author ""	Identifies who created the container image.

Option	Description
--message ""	Includes a commit message to the registry.
--format	Selects the format of the image. Valid options are **oci** and **docker**.

Listing 81. Using **podman commit**

```
[student@workstation ~]$ sudo podman commit [OPTIONS] CONTAINER \ > [REPOSITORY[:PORT]/]IMAGE_NAME[:TAG]
```

Note about **podman commit** *Options*

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

It is possible to use a **podman diff** command to identify changes with the running container image and the original image. The **diff** subcommand requires the name or container ID. The **diff** subcommand uses **A** for files that have been added, **C** for files that were changed, and **D** for files that were deleted.

Listing 82. **podman diff** *Usage*

```
[student@workstation ~]$ sudo podman diff mysql-basic
```

podman diff *and Mounted Filesystems*

IT should be noted that **diff** only reports differences in the container filesystem. Files that are part of a mounted filesystem to a container are not included as part of the **podman diff** command.

It is possible to save changes to another image (new image) using the **podman commit** command.

Listing 83. Sample using **podman commit**

```
[student@workstation ~]$ sudo podman commit mysql-basic mysql-custom
```

4.3.6. Tagging Images

Projects with multiple images based on the same software requires a maintenance and management approach for deploying images correctly and to the correct locations. This approach requires images to be tagged in order to manage multiple versions. The **podman tag** command is used to tag images.

Listing 84. Tagging images with **podman tag**

```
[student@workstation ~]$ sudo podman tag [OPTIONS] IMAGE[:TAG] \ > [REGISTRYHOST/][USERNAME/]NAME[:TAG]
```

The **IMAGE** argument is the image name with the optional tag which is managed by Podman.

Note Header

podman will always assume that the latest version of the image is to be used (normally indicated by the **latest** tag) if there is no tag value specified.

4.3.6.1. Removing Tags from Images

Single images can have multiple tags assigned by the **podman tag** command. In order to remove them use the **podman rmi** command.

Removing Images with Multiple Tags

Because multiple tags point to the same image, in order to remove an image referenced by a multiple tags, each tag must be removed individually.

4.3.7. Best Practices for Tagging Images

Podman automatically adds the **latest** tag when tagging an image if you don't specify a tag. Because tags can have different meaning, it is often a common practice to place multiple tags on an image to allow an end-user to easily pull an image based on either tag.

4.3.8. Publishing Images to a Registry

Images can be published to registries using the **podman push** command. In order to use **podman push** the image must reside on Podman's local storage and be properly tagged for identification purposes.

Listing 85. Using **podman push** *to Publish an Image*

```
[student@workstation ~]$ sudo podman push [OPTIONS] IMAGE [DESTINATION]
```

4.4. Demonstration - Manipulating Container Images

Example 6. DEMO - Manipulating Container Images

1. Save the **httpd** container image to a file

Listing 86. Using **podman save** *to Save a Container Image*

```
[student@workstation Chapter3]$ sudo podman save -o httpd-demo.tar redhattraining/httpd-parent:2.4
Getting image source signatures
Copying blob sha256:24d85c895b6b870f6b84327a5e31aa567a5d30588de0a0bdd9a669ec5012339c
 205.76 MB / 205.76 MB [=====================================================] 4s
Copying blob sha256:c613b100be1645941fded703dd6037e5aba7c9388fd1fcb37c2f9f73bc438126
 20.00 KB / 20.00 KB [=======================================================] 0s
Copying blob sha256:a3ed95caeb02ffe68cdd9fd84406680ae93d633cb16422d00e8a7c22955b46d4
 32 B / 32 B [===============================================================] 0s
Copying blob sha256:a3ed95caeb02ffe68cdd9fd84406680ae93d633cb16422d00e8a7c22955b46d4
 32 B / 32 B [===============================================================] 0s
Copying blob sha256:574bcc187eda95dd171dc76c3b8a8b39e07f38fda12accee6ea8477ea2c5f95a
 19.35 MB / 19.35 MB [=======================================================] 0s
Copying blob sha256:a3ed95caeb02ffe68cdd9fd84406680ae93d633cb16422d00e8a7c22955b46d4
 32 B / 32 B [===============================================================] 0s
Copying blob sha256:a3ed95caeb02ffe68cdd9fd84406680ae93d633cb16422d00e8a7c22955b46d4
 32 B / 32 B [===============================================================] 0s
Copying blob sha256:7f9108fde4a18112ccc28fd421097680ad890ebad48ca3e635063e5ac1adbfb3
 2.50 KB / 2.50 KB [=======================================================] 0s
Copying blob sha256:a3ed95caeb02ffe68cdd9fd84406680ae93d633cb16422d00e8a7c22955b46d4
 32 B / 32 B [===============================================================] 0s
Copying blob sha256:a3ed95caeb02ffe68cdd9fd84406680ae93d633cb16422d00e8a7c22955b46d4
 32 B / 32 B [===============================================================] 0s
Copying config sha256:3639ce1374d3611e80ed66decd7d5467b72d010c21e19e4f193cd8b944e8c9f5
 6.50 KB / 6.50 KB [=======================================================] 0s
Writing manifest to image destination
Storing signatures
```

2. Load an image from a **tar** file

Listing 87. Load **httpd-demo.tar** *for running*

```
[student@workstation Chapter3]$ sudo podman load -i httpd-demo.tar
Getting image source signatures
Skipping fetch of repeat blob sha256:24d85c895b6b870f6b84327a5e31aa567a5d30588de0a0bdd9a669ec5012339c
Skipping fetch of repeat blob sha256:c613b100be1645941fded703dd6037e5aba7c9388fd1fcb37c2f9f73bc438126
Skipping fetch of repeat blob sha256:574bcc187eda95dd171dc76c3b8a8b39e07f38fda12accee6ea8477ea2c5f95a
Skipping fetch of repeat blob sha256:7f9108fde4a18112ccc28fd421097680ad890ebad48ca3e635063e5ac1adbfb3
Copying config sha256:3639ce1374d3611e80ed66decd7d5467b72d010c21e19e4f193cd8b944e8c9f5
 6.50 KB / 6.50 KB [=======================================================] 0s
Writing manifest to image destination
Storing signatures
Loaded image(s): docker.io/redhattraining/httpd-parent:2.4
```

3. Run the imported image in a container

Listing 88. Run imported image in a container

```
[student@workstation Chapter3]$ sudo podman run -d --name HTTPD-Custom-Demo -p 8080:80 httpd-demo
3b9443763f778e8f23e2076cf3a8dbeb8528a265e065f089c9a93de7c567dbe7
```

4. Test webserver and customize content

Listing 89. Testing current container image

```
[student@workstation Chapter3]$ curl localhost:8080
Hello from the httpd-parent container!
```

Listing 90. Modifying Container Content

```
[student@workstation Chapter3]$ sudo podman exec -it HTTPD-Custom-Demo /bin/bash
bash-4.4# echo "I am custom material for the DO180 course" > /var/www/html/index.html
bash-4.4# exit
exit
```

Listing 91. Verifying Modified Container Content

```
[student@workstation Chapter3]$ curl localhost:8080
I am custom material for the DO180 course
```

5. Stop the container

Listing 92. Stopping the running container

```
[student@workstation Chapter3]$ sudo podman stop HTTPD-Custom-Demo
3b9443763f778e8f23e2076cf3a8dbeb8528a265e065f089c9a93de7c567dbe7
```

6. Commit the changes to a new container

Listing 93. Using **podman commit** *to commit changes*

```
[student@workstation Chapter3]$ sudo podman commit -a 'Travis Michette' HTTPD-Custom-Demo httpd-custom-demo-new
Getting image source signatures
Skipping fetch of repeat blob sha256:24d85c895b6b870f6b84327a5e31aa567a5d30588de0a0bdd9a669ec5012339c
Skipping fetch of repeat blob sha256:c613b100be1645941fded703dd6037e5aba7c9388fd1fcb37c2f9f73bc438126
Skipping fetch of repeat blob sha256:574bcc187eda95dd171dc76c3b8a8b39e07f38fda12accee6ea8477ea2c5f95a
Skipping fetch of repeat blob sha256:7f9108fde4a18112ccc28fd421097680ad890ebad48ca3e635063e5ac1adbfb3
Copying blob sha256:732708ab470496136cceeee875ca4a2fb767f88e291b7f5a233e82cd153c56c2
 10.50 KB / 10.50 KB [=======================================================] 0s
Copying config sha256:61044a30ab7d831fb211b61e78653563779e3a3ef0dde33a84d496aeb93eeac7
 4.14 KB / 4.14 KB [=========================================================] 0s
Writing manifest to image destination
Storing signatures
61044a30ab7d831fb211b61e78653563779e3a3ef0dde33a84d496aeb93eeac7
```

Container Names

It is important to remember that the name for the container is lowercase and without spaces.

7. List available images to verify **localhost/httpd-custom-demo-new** was created

Listing 94. Use **podman images** *to list available container images*

```
[student@workstation Chapter3]$ sudo podman images
REPOSITORY                                  TAG      IMAGE ID       CREATED              SIZE
localhost/httpd-custom-demo-new             latest   61044a30ab7d   About a minute ago   236MB

... output omitted ...
```

8. Tag the image and push to a repository

Listing 95. Tagging the container image

```
[student@workstation Chapter3]$ sudo podman tag httpd-custom-demo-new quay.io/tmichett/httpd-custom-demo-new:v1.0
```

Listing 96. Pushing image to the repository

```
[student@workstation Chapter3]$ sudo podman push httpd-custom-demo-new quay.io/tmichett/httpd-custom-demo-new:v1.0
Getting image source signatures
Copying blob sha256:24d85c895b6b870f6b84327a5e31aa567a5d30588de0a0bdd9a669ec5012339c
 205.76 MB / 205.76 MB [============================================] 5m43s
Copying blob sha256:c613b100be1645941fded703dd6037e5aba7c9388fd1fcb37c2f9f73bc438126
 20.00 KB / 20.00 KB [==============================================] 1s
Copying blob sha256:574bcc187eda95dd171dc76c3b8a8b39e07f38fda12accee6ea8477ea2c5f95a
 19.35 MB / 19.35 MB [==============================================] 32s
Copying blob sha256:7f9108fde4a18112ccc28fd421097680ad890ebad48ca3e635063e5ac1adbfb3
 2.50 KB / 2.50 KB [================================================] 1s
Copying blob sha256:732708ab470496136cceeee875ca4a2fb767f88e291b7f5a233e82cd153c56c2
 10.50 KB / 10.50 KB [==============================================] 1s
Copying config sha256:61044a30ab7d831fb211b61e78653563779e3a3ef0dde33a84d496aeb93eeac7
 4.14 KB / 4.14 KB [================================================] 2s
Writing manifest to image destination
Copying config sha256:61044a30ab7d831fb211b61e78653563779e3a3ef0dde33a84d496aeb93eeac7
 0 B / 4.14 KB [------------------------------------------------------] 0s
Writing manifest to image destination
Storing signatures
```

References

Podman site: https://podman.io/

5. Creating Custom Container Images

5.1. Designing Custom Container Images

Goals

- Describe approaches for creating custom container images
- Find existing Dockerfiles to use as starting point for creating custom container images
- Define the role played by Red Hat Software Collections Library (RHSCL) in designing container images from the Red Hat registry
- Describe the Source-to-Image (S2I) alternative to Dockerfiles

5.1.1. Reusing Existing Dockerfiles

Dockerfile: List of instructions of what you want in an image. Allows taking an image from a trusted source and extending it to meet specific purposes. Dockerfiles are also easy to share and be be placed in version control for reuse and later changes/modifications/extensions.

Dockerfile Image Customization

Dockerfiles allow existing images to be extended by using the base image (*parent image*) to create a new customized image (*child image*). Two great sources for parent images are Docker Hub and Red Hat Software Collections Library (RHSCL).

5.1.2. Working with the Red Hat Software Collections Library

The Red Hat Software Collections Library (RHSCL) provides an additional repository as part of a RHEL subscription with access to the latests development tools (not normally distributed as part of the standard RHEL release schedule). Packages in this repository don't conflict with default RHEL packages and can normall be installed side-by-side.

5.1.3. Finding Dockerfiles from the Red Hat Software Collections Library

Red Hat provides RHSCL Dockerfiles and related sources in the rhscl-dockerfiles package available from the RHSCL repository.

Source-to-Image 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.

5.1.4. Container Images in Red Hat Container Catalog (RHCC)

5.1.5. Searching for Images Using Quay.io

5.1.6. Finding Dockerfiles on Docker Hub

5.1.7. Describing How to use the OpenShift Source-to-Image Tool

Source-to-Image S2I provides alternatives to Dockerfiles for creating a new container. S2I can be used as a standalone **s2i** utility or can be used with OpenShift. S2I allows developers to use existing tools without requiring use or learning of a Dockerfile. These tools often work with version control systems such as **git** and start from a container base image called a **builder image**.

S2I Process to Build Custom Container Images

1. Start the container from a base container image (builder image)
2. Fetch source code from version control
3. Build application binary files inside the container
4. Save the container as a new image

The **s2i** command runs the S2I process outside OpenShift in a Docker-only environment. It is part of the source-to-image RPM package in RHEL and also available from the S2I project on Github.

References

Red Hat Software Collections Library (RHSCL): https://access.redhat.com/documentation/en/red-hat-software-collections/

Red Hat Container Catalog (RHCC): https://access.redhat.com/containers/

RHSCL Dockerfiles on GitHub: https://github.com/sclorg?q=-container

Using Red Hat Software Collections Container Images: https://access.redhat.com/articles/1752723

Quay.io: https://quay.io/search

Docker Hub: https://hub.docker.com/

Docker Library GitHub project: https://github.com/docker-library

The S2I GitHub project: https://github.com/openshift/source-to-image

5.2. Building Custom Container Images with Dockerfiles

Goal: Create a container image using common Dockerfile commands

5.2.1. Buidling Base Containers

The **Dockerfile** will be the mechanism to automate building of container images and specify the components of the container. There is a three-step process to build images from a Dockerfile.

Dockerfile 3 Step Process

1. Create working directory

2. Write **Dockerfile**
3. Build image with Podman

5.2.1.1. Create a Working Directory

The **working directory** is the directory containing all files needed to build an image. It is best to create an empty working directory to avoid accidentally including extra files.

5.2.1.2. Write the Dockerfile Specification

The **Dockerfile** is a text file that must exist in the working directory. It contains instructions needing to build an image. It is also possible to include comments in a Dockerfile using the # sign.

Listing 97. **Dockerfile** *Specification Syntax*

```
# Comment
INSTRUCTION arguments
```

Dockerfile *Instructions*

- **FROM** - specifies the base image to be used by the Dockerfile
- **CMD** - Default command arguments/options which are used with the **ENTRYPOINT**. Easily overridden by Podman when starting a container
- **ENTRYPOINT** - Command to be executed when the container starts. This can be the command and additional arguments/options/parameters or just the initial command with no parameters. The **ENTRYPOINT** cannot be overridden by Podman on the command line.
- **ADD** -
- **COPY** -
- **RUN** - Executes commands for image layering. This executes commands using **/bin/sh** and creates a new layer on top of the current image.

The **FROM** *Instruction*

The first non-comment instruction must be the **FROM** instruction specifying the base image to be used by the Dockerfile. Subsequent instructions are then executed in the newly created image using the specified image as the base image. Instructions in the Dockerfile execute in sequential order.

TIP Header

Each instruction from a **Dockerfile** runs in an independent container using intermediate images built from a previous command. Therefore all Dockerfile instructions are independent from other instructions in the Dockerfile.

Listing 98. Sample Dockerfile

```
# This is a comment line
FROM ubi7/ubi:7.7  # Defines the Universal Base image as the default image
LABEL description="This is a custom httpd container image" # Provides metadata for the image
MAINTAINER John Doe <jdoe@xyz.com> # Provides metadata for the image
RUN yum install -y httpd # Installs packages in the image
EXPOSE 80 # Allows a port to be exposed when running
ENV LogLevel "info"
ADD http://someserver.com/filename.pdf /var/www/html  # Copies a files from a webserver into the containers
COPY ./src/ /var/www/html/  # Copies files from the current working directory of system building the container into the container
USER apache  # Specifies User/UID of the user running the container image and applies to RUN,CMD, and ENTRYPOINT.
ENTRYPOINT ["/usr/sbin/httpd"]   # Default command to execure when image is started and run as a container
CMD ["-D", "FOREGROUND"]   # Default arguments for the ENTRYPOINT
```

5.2.2. CMD and ENTRYPOINT

Dockerfiles should container at most one **ENTRYPOINT** and one **CMD** instruction.

Important Header

CMD parameters can be overridden by command line. **ENTRYPOINT** cannot be overridden by **podman** on the command line.

5.2.3. ADD and COPY

Both the **ADD** and **COPY** instructions have two forms and both can be used to copy files. It should be noted however, that **ADD** has additional functionality.

ADD *Instruction Functionality*

- If source is compressed, **ADD** will decompress the file to the destination folder in the container
- If the source is a URL, **ADD** can be used to copy the file from the URL to the destination folder in the container

Filesystem Path and **source** *Files*

If the *source* for **COPY** or **ADD** is a filesystem path, it must be located in the working directory.

File Permissions with **ADD** *and* **COPY**

Both the ADD and COPY instructions copy the files, retaining permissions, with root as the owner, even if the USER instruction is specified. Red Hat recommends using a RUN instruction after the copy to change the owner and avoid “permission denied” errors.

5.2.4. Layering Image

Each new **Dockerfile** instruction results in a new image layer being created. If there are too many instructions in a **Dockerfile** the image created by the **Dockerfile** will contain too many layers. It is best to consolidate instructions into a useful form while still maintaining readability.

Listing 99. **Dockerfile** *Unconsolidated* **RUN** *Instruction Example*

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

Eliminate Multiple Layers and Preserve Readability

Each instruction creates a separate layer in the image. It is possible to merge instructions on a single line with **&&** to minimize the number of layers. However, merging layers on a single line can cause issues with readability. In order to improve readability, use the **** escape code to insert line breaks.

The goal of a well-defined Dockerfile is to eliminate unnecessary image layers while maintaining readability of the file.

Listing 100. **Dockerfile** *Consolidated* **RUN** *Instruction*

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

5.2.5. Building Images with Podman

The **podman build** command processes the **Dockerfile** and builds a new image.

Listing 101. Sample **podman build**

```
$ podman build -t NAME:TAG DIR
```

The **DIR** argument is the directory containing the **Dockerfile** for building the image. You can specify "**.**" for the current working directory. The **NAME:TAG** parameter allows specifying a name and a tag for the newly created image. If no **TAG** is provided, the image is tagged as *latest*.

5.3. Demonstration - Building an Image with a Dockerfile

Example 7. Demo - Using a Dockerfile to Build an Image

1. Create Dockerfile with content

Listing 102. Creation of Dockerfile

```
[student@workstation webserver]$ vim Dockerfile

## Dockerfile to create an Apache Container Image
## Set to expose port 80 for web traffic
## Set to run Apache on launch
## Uses Universal base image for RHEL7 as RHEL8 not in REPO

FROM ubi7/ubi:7.7

MAINTAINER Travis Michette <tmichett@redhat.com>

LABEL description="Custom webserver demo for DO180"

RUN yum install -y httpd vim && yum clean all

RUN echo "This is a demo of Apache for DO180 from a Dockerfile" > /var/www/html/index.html

EXPOSE 80

## NGINX not available for classroom environment CMD ["nginx", "-g", "daemon off;"]
ENTRYPOINT ["httpd", "-D", "FOREGROUND"]
```

2. Build the Container image with **podman**.

Listing 103. Source Description

```
[student@workstation webserver]$ sudo podman build --layers=false -t do180/httpd_ch5_demo .

STEP 1: FROM ubi7/ubi:7.7
STEP 2: MAINTAINER Travis Michette <tmichett@redhat.com>
STEP 3: LABEL description="Custom webserver demo for DO180"
STEP 4: RUN yum install -y httpd vim && yum clean all
Loaded plugins: ovl, product-id, search-disabled-repos, subscription-manager
This system is not receiving updates. You can use subscription-manager on the host to register and assign subscriptions.

... output omitted ...

Writing manifest to image destination
Storing signatures
--> ab34b1f7ef6bd22b84dc1e6a07c1be66b7771c23938ed5a7eebf48efd6d00506
```

Eliminating Intermediate Layers

It is important to eliminate any intermediate layers not needed for the image. The **--layers=false** command causes intermediate layers to be deleted.

Repositories and Files Must be Available

It is important that files and repositories used in the dockerfile be available.

+ .Trainig Repository

```
[student@workstation webserver]$ tree
.
├── Dockerfile
└── training.repo

0 directories, 2 files
[student@workstation webserver]$ cat training.repo
[rhel_dvd]
baseurl = http://content.example.com/rhel7.6/x86_64/dvd
enabled = true
gpgcheck = false
name = Remote classroom copy of dvd
```

3. Verify newly built image exists

Listing 104. Using **podman images** *to look for image*

```
[student@workstation webserver]$ sudo podman images
REPOSITORY                                  TAG      IMAGE ID       CREATED         SIZE
localhost/do180/httpd_ch5_demo              latest   ab34b1f7ef6b   2 minutes ago   307MB
```

4. Test newly built image

Listing 105. Using **podman run** *to launch image*

```
[student@workstation webserver]$ sudo podman run --name HTTPD-CH5-DEMO -d -p 9080:80 do180/httpd_ch5_demo
9d62ecd59e69711f8e49e7590d5b709cda3c6a0b5303fec332ea879d72f2f78e
```

Listing 106. Testing HTTPD webserver using **curl**

```
[student@workstation webserver]$ curl localhost:9080
This is a demo of Apache for DO180 from a Dockerfile
```

5. Stop container and Cleanup Image

Listing 107. Stopping and Removing the Container and Image

```
[student@workstation webserver]$ sudo podman stop HTTPD-CH5-DEMO
9d62ecd59e69711f8e49e7590d5b709cda3c6a0b5303fec332ea879d72f2f78e


[student@workstation webserver]$ sudo podman rm HTTPD-CH5-DEMO
9d62ecd59e69711f8e49e7590d5b709cda3c6a0b5303fec332ea879d72f2f78e

[student@workstation webserver]$ sudo podman rmi httpd_ch5_demo
ab34b1f7ef6bd22b84dc1e6a07c1be66b7771c23938ed5a7eebf48efd6d00506
```

References

Dockerfile Reference Guide: https://docs.docker.com/engine/reference/builder/

Creating base images: https://docs.docker.com/engine/userguide/eng-image/baseimages/

6. Deploying Containerized Applications on OpenShift

6.1. Describing Kubernetes and OpenShift Architecture

Goals

- Describe the architecture of a Kubernetes cluster running on OCP
- List main resource types provided by Kubernetes and OCP
- Identify the network characteristics of containers, Kubernetes, and OCP
- List mechanisms to make a pod externally available

6.1.1. Kubernetes and OpenShift

Kubernetes is an orchestration service simplifying deployment, management, and scaling of containerized applications. Servers can act as both a server and node, but generally the roles are segregated for increased stability.

Table 5. Kubernetes Terminology

Term	Definition
Node	A server that hosts applications in a Kubernetes cluster.
Master Node	A node server that manages the **control plane** in a Kubernetes cluster. Master nodes provide basic cluster services such as APIs or controllers.
Worker Node	Also named **Compute Node**, worker nodes execute workloads for the cluster. Application pods are scheduled onto worker nodes.
Resource	Resources are any kind of component definition managed by 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). *Generally controlled by a YAML file and are:* • **Pod** • **Service** • **Route**

Term	Definition
Controller	A controller is a Kubernetes process that watches resources and makes changes attempting to move the current state towards the desired state. *Watches resources and maintains 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. *Assists with resource organization*.
Namespace	A scope for Kubernetes resources and processes, so that resources with the same name can be used in different boundaries. *Known as projects in OpenShift. Bundles resources and processes to stay organized*.

Red Hat OpenShift Container Platform (RHOCP) is a set of modular components and services built on top of RHEL CoreOS and Kubernetes. An OCP cluster is a Kubernetes cluster that can be managed with a CLI or web console.

Table 6. OpenShift Terminology

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

The image below is an illustration of the OCP Platform stack. The pale green portions are the components that OCP adds to the base Kubernetes architecture.

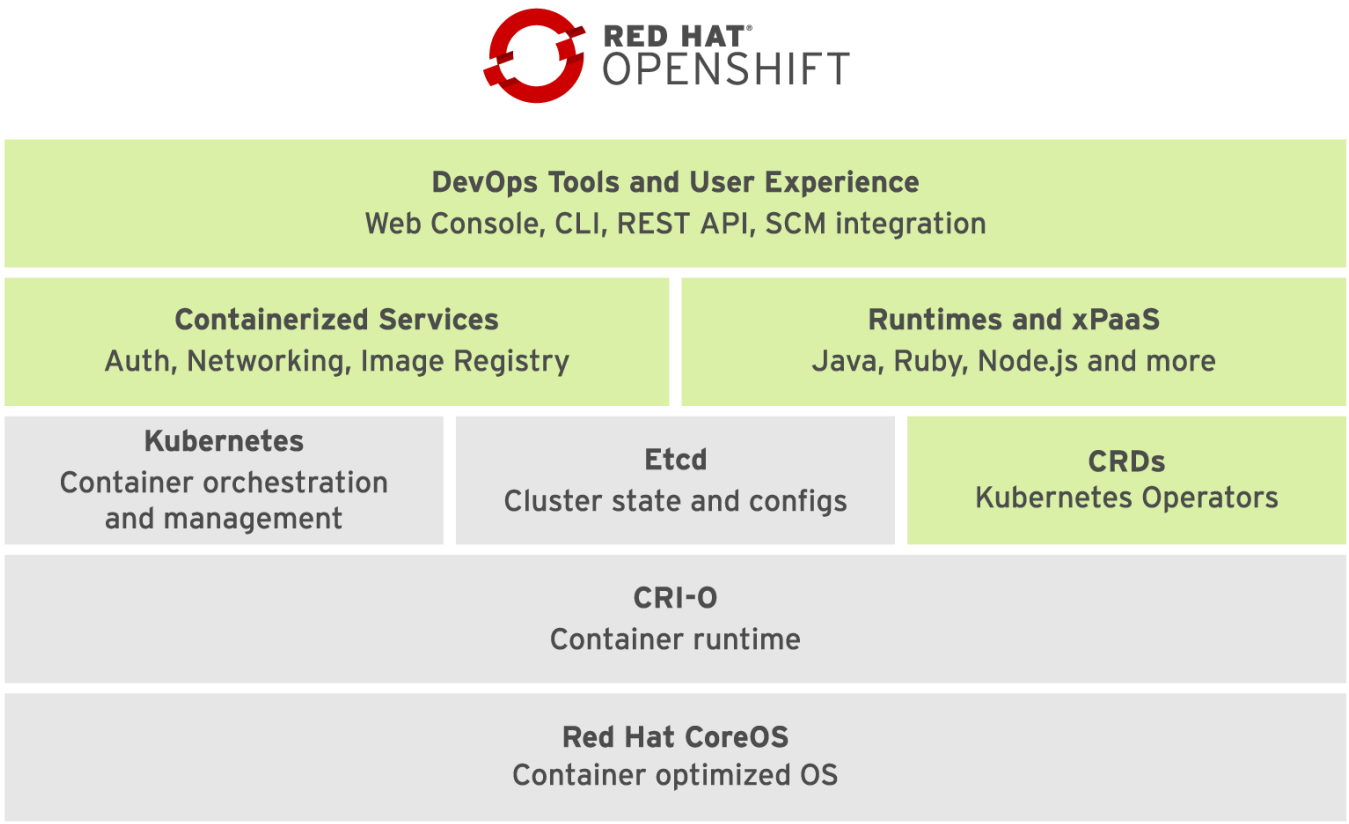

Figure 10. OCP Component Stack

OCP Infrastructure Components

- **BaseOS** - CoreOS is the base OS for containers
- **CRI-O** - Implementation of Kubernetes CRI (Container Runtime Interface) allowing OCI compatible runtimes.
- **Kubernetes** - Manages a cluster of hosts used to run containers. The resource and orchestration component.
- **Etcd** - Distributed key-value store. Maintains cluster state and configuration resources.
- **CRDs** - Custom Resource Definitions stored in **etcd** and managed by Kubernetes. Form the state and configuration of resources managed by OpenShift.
- **Containerized Services** - Fulfill PaaS infrastructure functions (authentication, networking, image registry etc.). These are responsible for most OCP internal services
- **Runtimes and xPaaS** - Base container images ready for use. All of these are preconfigured for particular runtime languages or databases.
- **DevOps Tools and User Experience** - OCP provides the **oc** CLI management tool as well as a webUI with a graphical management console. These use REST APIs which can be integrated with IDEs and CI platforms.

Containers and CoreOS

It is important to note that all pieces of the OCP platform uses containers. As of OCPv4, CoreOS is used as the container optimized OS for which all components can be supported.

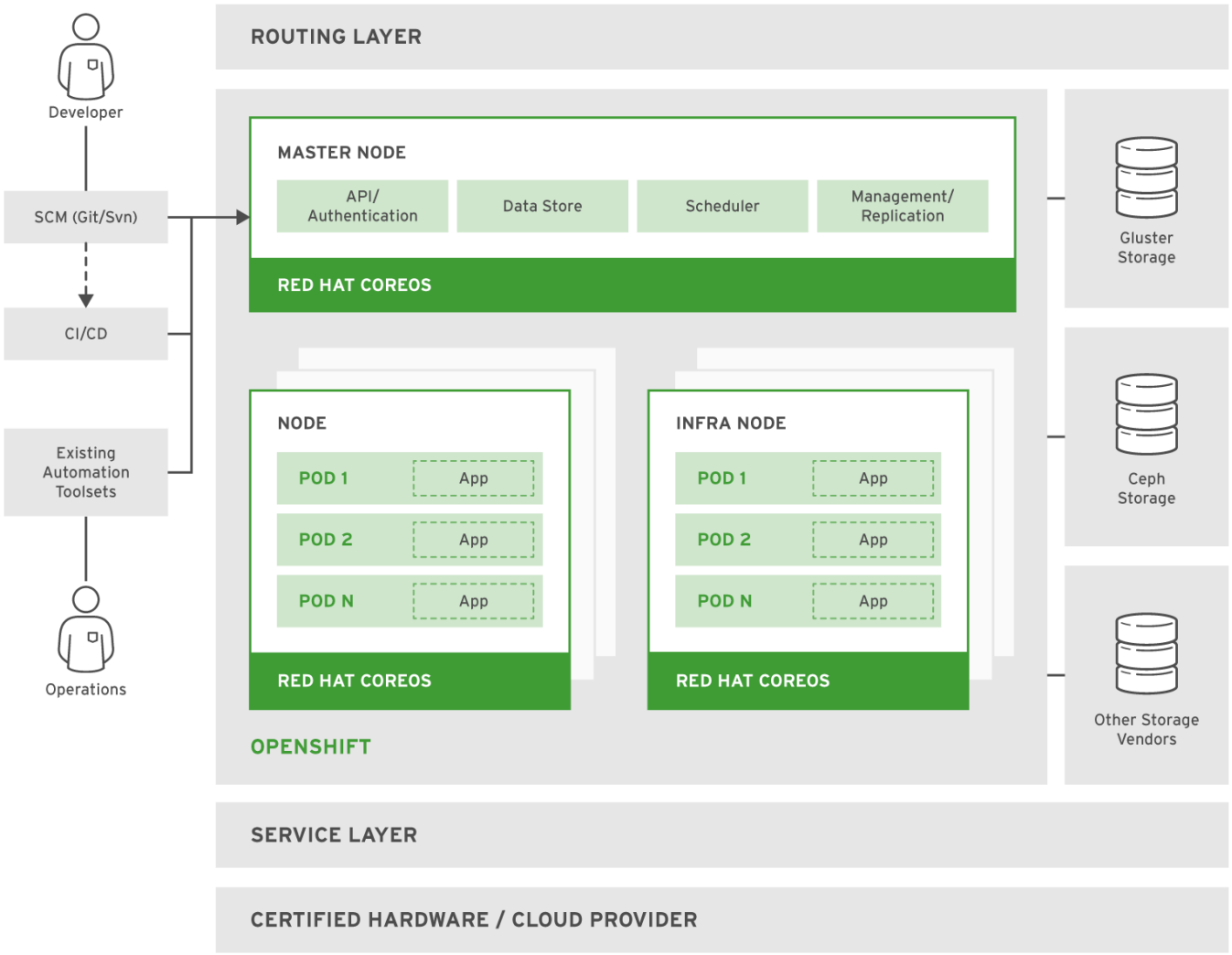

Figure 11. DO447 Classroom Layout

6.1.2. New Features in RHOCP 4

- CoreOS is OS for all nodes
- New installer (based on CoreOS enhancements)
- Self-managing platform that can automatically apply updates and recover without disruption
- New Life-cycle management

- Operator SDK to build/test/package Operators

6.1.3. Describing Kubernetes Resource Types

Kubernetes (and therefore OCP) have six main resource types that can be created and configured using a YAML or JSON file as well as OpenShift management tools.

- **Pods (po)** - Group of containers storing resources (IP addresses, persistent storage volumes, etc).
- **Services (svc)** - Single IP/port combination providing access to a pool of pods.
- **Replication Controllers (rc)** - Defines how pods are replicated. Provides high availability for pods and containers.
- **Persistent Volumes (pv)** = Defines storage areas used by Kubernetes pods. This provides persistent storage to a container like **podman -v** option.
- **Persisten Volume Claims (pvc)** - Represents a request for storage by a pod. Links PVs to containers.
- **ConfigMaps (cm) and Secrets** - How important central configuration is stored. Secrets are used for storing passwords and encoding information.

6.1.4. OpenShift Resource Types

OCP brings a few more main resource types to the Kubernetes underlying architecture.

- **Deployment config (dc)** - Represents set of containers included in a pod and is the config used by the **BC**.
- **Build Config (bc)** - Defines how apps are built/executed in OpenShift projects. The **BC** works with the **DC** to provide CI/CD workflows. This is what is used by OCP's S2I feature.
- **Routes** - Provides a DNS host name (ingress point) for applications and microservices. This exposes a Kubernetes service to the outside network.

6.1.5. Networking

Each container in a Kubernetes cluster has an IP address assigned from an internal network that is only accessible from the node running the container. These IP addresses and constantly assigned and released since containers are ephemeral.

Kubernetes provides an SDN (software-defined network) that spawns internal networks from multiples nodes and allows communication to containers from any pod inside any host. The SDN only works inside the same Kubernetes cluster.

External access to containers can be provided and Kubernetes can specify **NodePort** although this is not recommended by Red Hat and does not scale well. OpenShift makes external access to containers simple and scalable by defining the **Route** resource which defines external-facing DNS names and ports to a service. An OCP route is the desired way to provide access to container services.

Exposing Services Externally

The **oc expose <svc>** command can be used to create a route for a service in order to direct traffic.

Listing 108. Creating a Route and Exposing a Service

```
# oc expose <svc>
```

References

Kubernetes documentation website: https://kubernetes.io/docs/

OpenShift documentation website: https://docs.openshift.com/

Understanding Operators: https://docs.openshift.com/container-platform/4.2/operators/olm-what-operators-are.html

6.2. Creating Kubernetes Resources

Goal: Be able to create standard Kubernetes resources

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

The main method interacting with OCP is using the **oc** command, which is the command-line tool provided with OpenShift.

Listing 109. **oc** *Command Syntax*

```
# oc <command>
```

Using the **oc** *Command*

Before interacting with and managing an OCP cluster, it is often necessary to login. The **oc login** command can provide authentication into the OCP environment.

Listing 110. **oc** *Login*

```
# oc login <Cluster_URL>
```

6.2.2. Describing Pod Resource Definition Syntax

OCP runs containers inside of Kubernetes pods. Each OCP pod needs a resource definition file provided as a JSON or YAML text file or can be generated from defaults using the **oc new-app** command or OCP Web console.

Environment Variables for Containers

Kubernetes transforms all **name** and **value** pairs to environment variables for OCP.

6.2.3. Describing Service Resource Definition Syntax

Kubernetes provides a virtual network allowing pods from different workers to connect. However, there is no easy method for pods to discover IP addresess from other pods. A **service** is an essential resource to OCP applications as they allow containers in one pod to interact with containers in anoter pod. Each time a pod is restarted or a new pod is started, they get different IP addresses. A **service** provides a stable IP address for pods to use no matter how many times a pod gets restarted.

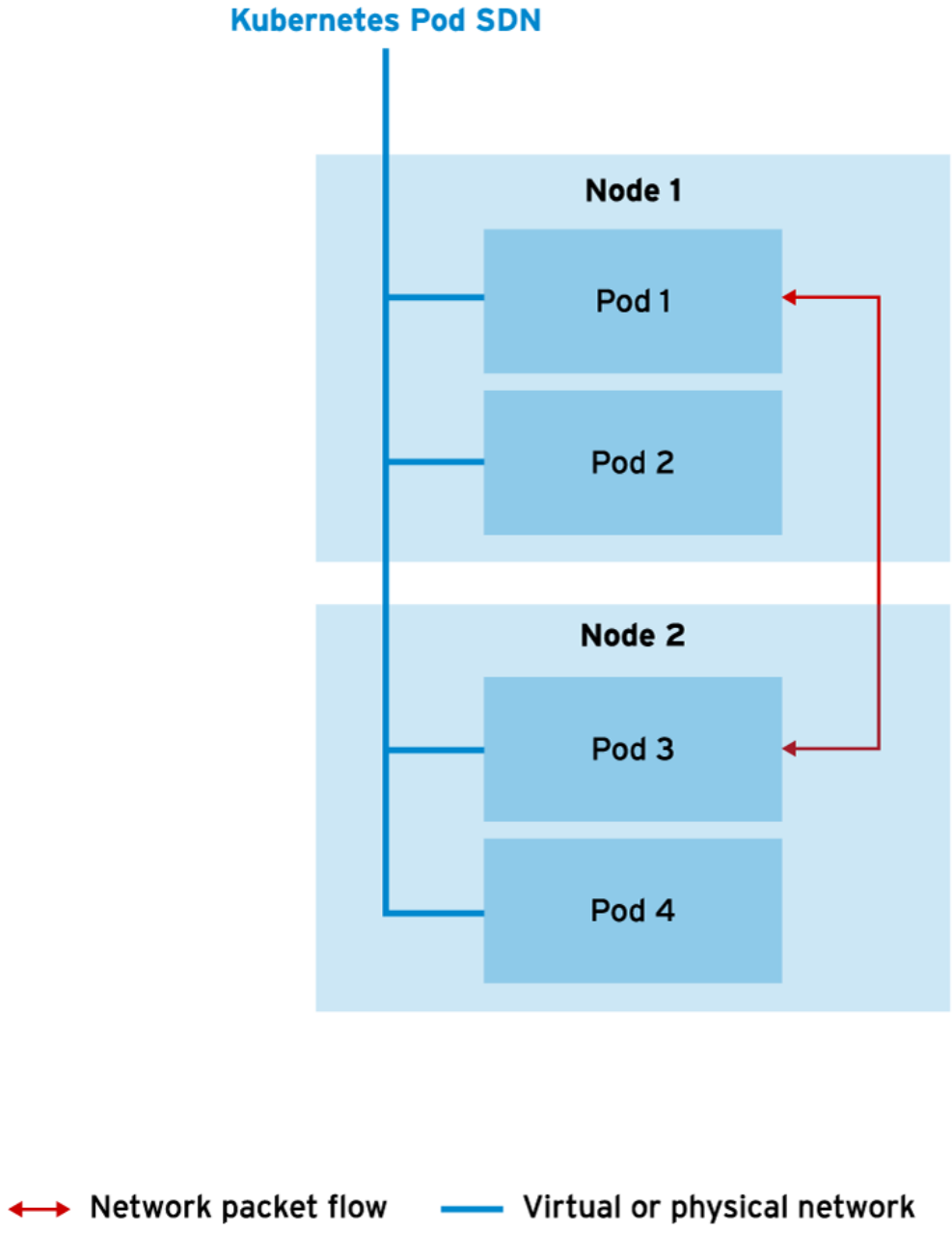

Figure 12. Basic Kubernetes Networking

Need for a **service**

A **service** is used because it should be assumed that pods will go down and be brought back onlune by a replication controller.

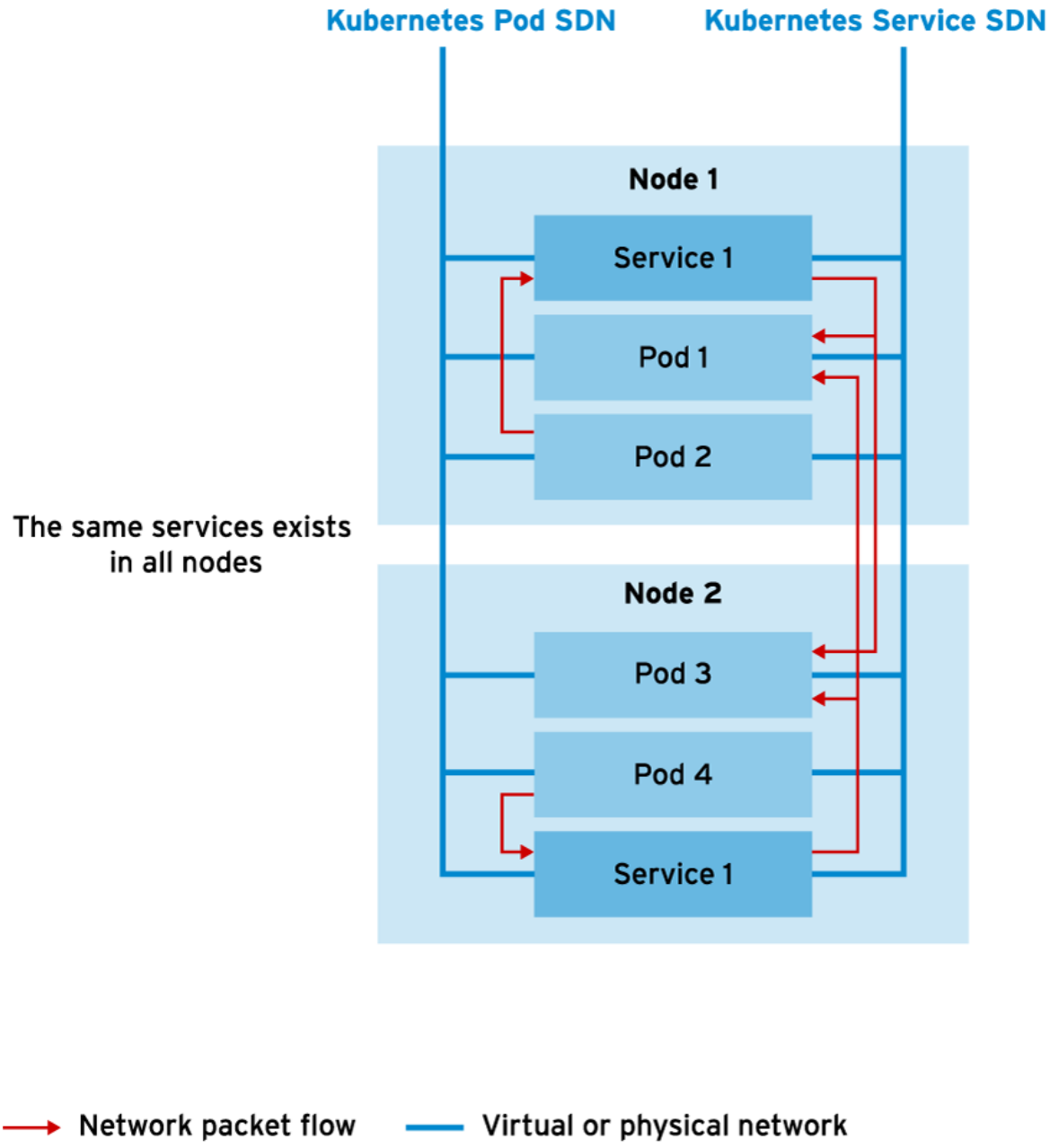

Figure 13. Kubernetes **services** *Networking*

Most real-world application run on multiple pods. The set of running pods behind a service is managed by a **DeploymentConfig**

resource. These resources embed a **ReplicationController** managing how many replicas are created from a pod.

Listing 111. Minimal Service Definition

```
{
  "kind": "Service", ①
  "apiVersion": "v1",
  "metadata": {
      "name": "quotedb" ②
  },
  "spec": {
      "ports": [ ③
          {
              "port": 3306,
              "targetPort": 3306
          }
      ],
      "selector": {
          "name": "mysqldb" ④
      }
    }
}
```

① Defines a service

② Provides name for service

③ Maps network ports

④ Selects pods for pointing a service to. Note that multiple pods can be selected.

service *IP Address*

Each service is assigned a unique IP address provided from the OCP SDN. Each pod matching the **selector** is added to the service as an endpoint.

6.2.4. Discovering Services

Applications find service IP addresses and ports using environment variables. Some environment variables are automatically defined and injected into containers for all pods inside the same project.

Automatic Environment Variables

- **SVC_NAME_SERVICE_HOST** - Service IP Address
- **SVC_NAME_SERVICE_PORT** - Service TCP Port number

SVC_NAME *Variable Information*

The **SVC_NAME** part of the variable is changed to comply with DNS naming restrictions: letters are capitalized and underscores (_) are replaced by dashes (-).

The OCP internal DNS server can also be used to discover services from a pod.

SVC_NAME.PROJECT_NAME.svc.cluster.local

There are two ways for an application to access services from outside an OCP Cluster.

1. **NodePort** type - Older Kubernetes approach. It is possible to use **oc edit svc** to edit service attributes and specify a **NodePort** and provide the port values.
2. **OpenShift Routes** - Preferred approach for OCP to expose services with a unique URL. The **oc expose** command is used to create a route in order to expose a service.

The figure below shows how **NodePort** services allow external access to Kubernetes services by using port forwarding.

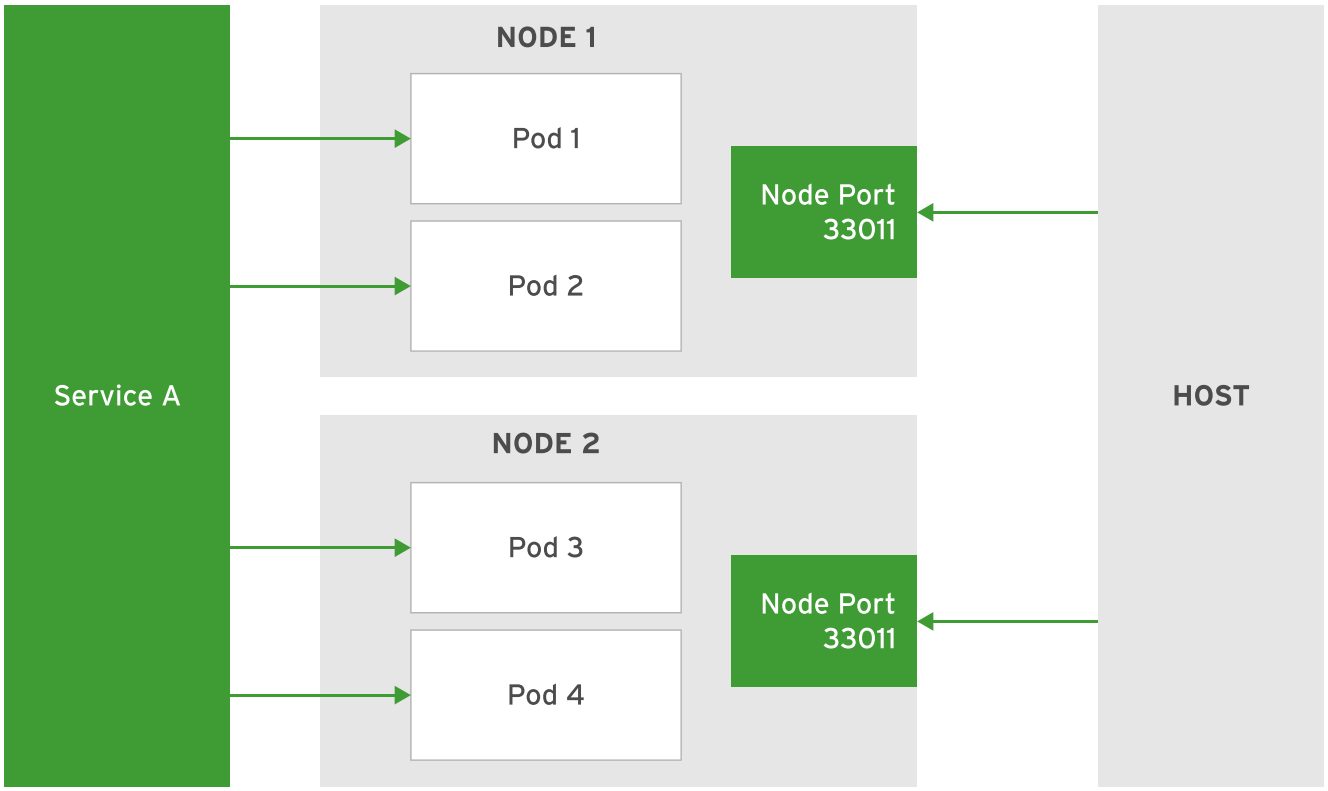

Figure 14. Alternatives for Kubernetes Service External Access

OCP provides the **oc port-forward** command to forward local ports to a port on a pod and can be done on-the-fly. It is important to note that with **oc port-forward** that there is no load balancing and this is a different from using a service.

Port-Forward Notes (Different from a **service***)*

- Port-forward mapping exists on on workstation running the **oc** client
- Port-forward maps a connection to a single pod

NodePort *Approach Should be Avoided*

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.

6.2.5. Creating New Applications

All types of applications can be described by a single resource definition file. The file should contain:

- pod definitions
- service definitions
- replication controllers
- DeploymentConfigs
- PersistentVolumeClaims
- anything else needed that is managed by OCP

The **oc new-app** command can be used with **-o json** or **-o yaml** options to create a skeleton resource definition file in JSON or YAML format. This file defines the application and all needed resources. The file can also be customized and used to create an application with the **oc create -f <filename>** command.

The **oc new-app** command can create pods for OCP in different ways including:

- existing docker images
- from Dockerfiles
- from raw source code using S2I

The image below is a graphical representation of using the **oc new-app** command to create an application from a container image.

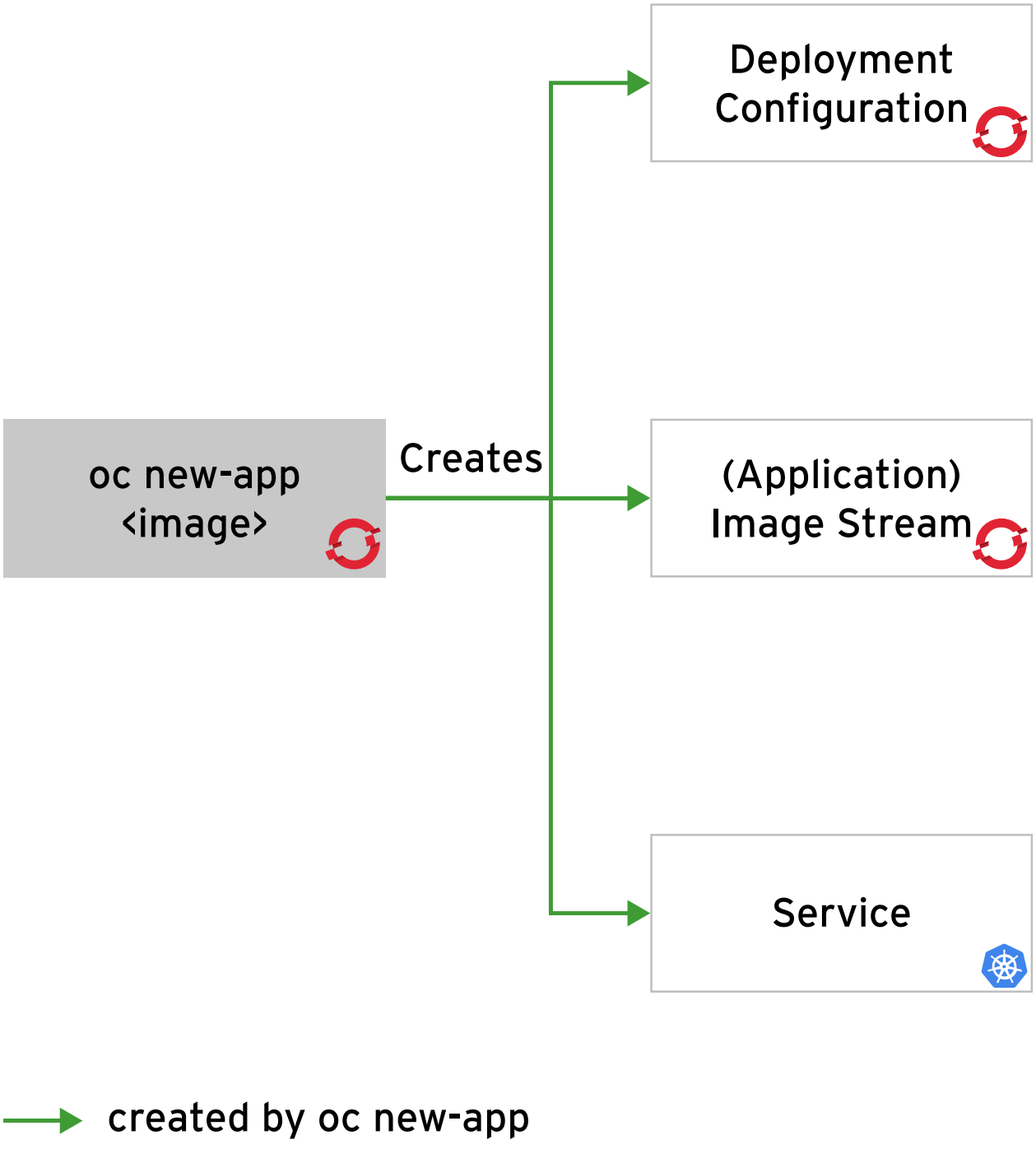

Figure 15. Resources for New Application

Getting **oc new-app** *Help*

The **oc new-app -h** command can be used to see different options available when creating a new application.

6.2.6. Managing OpenShift Resources at the Command Line

The **oc get** command is the main method for retrieving information about cluster resource. It is typically used in this format **oc get RESOURCE_TYPE**. This will display a summary of all resources available of the specified type. Another useful command is the **oc get pods** command which will show all pods and their status.

Listing 112. Getting a Pod Listing

```
[root@ocpbuilder OCP]# oc get pods
NAME                READY   STATUS    RESTARTS   AGE
dns-default-6f7xf   2/2     Running   0          5d15h
dns-default-7slj5   2/2     Running   0          5d14h
dns-default-lfhnq   2/2     Running   0          5d15h
dns-default-xx67v   2/2     Running   0          5d14h
dns-default-zdzv8   2/2     Running   0          5d15h
```

Listing 113. Getting a Node Listing

```
[root@ocpbuilder OCP]# oc get nodes
NAME                         STATUS   ROLES    AGE     VERSION
ocp4-8s4r4-master-0          Ready    master   5d17h   v1.17.1
ocp4-8s4r4-master-1          Ready    master   5d17h   v1.17.1
ocp4-8s4r4-master-2          Ready    master   5d17h   v1.17.1
ocp4-8s4r4-worker-0-4srf8    Ready    worker   5d15h   v1.17.1
ocp4-8s4r4-worker-0-9xfff    Ready    worker   5d15h   v1.17.1
```

6.2.6.1. oc get all

The **oc get all** command retrieves a summary of the most important OCP cluster components

```
# oc get all
```

6.2.6.2. oc describe RESOURCE_TYPE RESOURCE_NAME

The **oc describe** command can retrieve additional information about resource. The **oc describe** command provides detailed information on a specific resource.

Listing 114. Using **oc describe**

```
[root@ocpbuilder OCP]# oc describe service/dns-default
Name:              dns-default
Namespace:         openshift-dns
Labels:            dns.operator.openshift.io/owning-dns=default
Annotations:       <none>
Selector:          dns.operator.openshift.io/daemonset-dns=default
Type:              ClusterIP
IP:                172.30.0.10
Port:              dns  53/UDP
TargetPort:        dns/UDP
Endpoints:         10.128.0.2:5353,10.128.2.5:5353,10.129.0.4:5353 + 2 more...
Port:              dns-tcp  53/TCP
TargetPort:        dns-tcp/TCP
Endpoints:         10.128.0.2:5353,10.128.2.5:5353,10.129.0.4:5353 + 2 more...
Port:              metrics  9153/TCP
TargetPort:        metrics/TCP
Endpoints:         10.128.0.2:9153,10.128.2.5:9153,10.129.0.4:9153 + 2 more...
Session Affinity:  None
Events:            <none>
```

6.2.6.3. oc export

The **oc export** command can export a resource definition. The **export** command prints out the object representation in YAML format, however it can be changed with the **-o** option. This can export resources to JSON/YAML formats.

6.2.6.4. oc create

The **oc create** command creates a resource from a resource definition file. It is often paired with **oc export** for editing resource definitions.

6.2.6.5. oc edit

The **oc edit** command allows a user to edit resources of a resource definition. This directly edits a resource.

6.2.6.6. oc delete RESOURCE_TYPE name

The **oc delete** command removes a resource from the OCP cluster. When a project is deleted all resources and applications within the project get deleted.

6.2.6.7. oc exec CONTAINER_ID options command

The **oc exec** command allows commands to be executed inside a container. This command is similar to **podman exec** and allows both interactive and noninteractive commands to be run.

6.2.7. Labeling resources

Labels can be defined allowing an administrator to establish groups of resources within a project. A label is part of the **metadata** section of a resource and is defined as a key/value pair.

*The **oc** Command and Labels*

Most of the **oc** subcommands support the **-l** option to process resources from a label specification.

Label Definitions with Templates

Labels are applied to all objects below it when defined at the top of a template.

6.3. Demonstration - Creating a Kubernetes Resource

Example 8. DEMO - Deploying a Webserver on OpenShift

1. Login to OCP Cluster

Listing 115. Accessing Credential File and Logging In

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

[student@workstation webserver]$ oc login -u ${RHT_OCP4_DEV_USER} -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful.

You don't have any projects. You can try to create a new project, by running

    oc new-project <projectname>
```

2. Create a new project

*Listing 116. Use **oc new-project** to create a new OCP project*

```
[student@workstation Chapter6]$ oc new-project ${RHT_OCP4_DEV_USER}-mysql-ocp-demo
Now using project "travis-mysql-ocp-demo" on server "https://api.ocp4.michettetech.com:6443".

You can add applications to this project with the 'new-app' command. For example, try:

    oc new-app django-psql-example

to build a new example application in Python. Or use kubectl to deploy a simple Kubernetes application:

    kubectl create deployment hello-node --image=gcr.io/hello-minikube-zero-install/hello-node
```

3. Create a new MySQL application

*Listing 117. Use **oc new-app** to create a new application*

```
[student@workstation Chapter6]$ oc new-app \
--docker-image=registry.access.redhat.com/rhscl/mysql-57-rhel7:latest \
--name=mysql-ocp \
-e MYSQL_USER=demouser -e MYSQL_PASSWORD=redhat -e MYSQL_DATABASE=demodb \
-e MYSQL_ROOT_PASSWORD=r00tpa55
--> Found Docker image 60726b3 (9 months old) from registry.access.redhat.com for "registry.access.redhat.com/rhscl/mysql-57-
rhel7:latest"

... output omitted ...
```

4. Verify running pods

Listing 118. Use **oc get pods -o=wide** *to List Pods and names*

```
[student@workstation Chapter6]$ oc get pods -o=wide
NAME                READY   STATUS      RESTARTS   AGE     IP            NODE                          NOMINATED NODE   READINESS
GATES
mysql-ocp-1-67h2z   1/1     Running     0          5m14s   10.131.0.16   ocp4-7h47x-worker-0-g8dql   <none>           <none>
mysql-ocp-1-deploy  0/1     Completed   0          5m17s   10.131.0.15   ocp4-7h47x-worker-0-g8dql   <none>           <none>
```

Getting Node Names

It is important to note the POD name from the **oc get pods** command as this will be used for setting up port forward to the POD. In this case, the name is **mysql-ocp-1-67h2z**

5. Expose the Service

Listing 119. Use **oc expose service** *to Expose the* **mysql-ocp** *service*

```
[student@workstation Chapter6]$ oc expose service mysql-ocp
route.route.openshift.io/mysql-ocp exposed
```

6. Obtain the Route

Listing 120. Use **oc get routes** *to obtain the routes*

```
[student@workstation Chapter6]$ oc get routes
NAME        HOST/PORT                                                PATH   SERVICES    PORT       TERMINATION   WILDCARD
mysql-ocp   mysql-ocp-travis-mysql-ocp-demo.apps.ocp4.michettetech.com         mysql-ocp   3306-tcp                 None
```

7. Use Port forwarding in one terminal window and test in another

Listing 121. Use **oc port-forward** *to Port Forward and Test*

```
[student@workstation Chapter6]$ oc port-forward mysql-ocp-1-67h2z 3306:3306
Forwarding from 127.0.0.1:3306 -> 3306
Forwarding from [::1]:3306 -> 3306
```

Listing 122. Testing of Port Forwarding

```
[student@workstation ~]$ mysql -u demouser -predhat --protocol tcp -h localhost
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MySQL connection id is 3
Server version: 5.7.24 MySQL Community Server (GPL)

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MySQL [(none)]> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| demodb             |
+--------------------+
2 rows in set (0.01 sec)
```

8. Cleanup OCP

Listing 123. Delete OCP project with **oc delete project** *Command*

```
[student@workstation Chapter6]$ oc delete project ${RHT_OCP4_DEV_USER}-mysql-ocp-demo
project.project.openshift.io "travis-mysql-ocp-demo" deleted
```

References

Additional information about pods and services is available in the Pods and Services section of the OpenShift Container Platform documentation: **Architecture** - https://access.redhat.com/documentation/en-us/openshift_container_platform/4.2/html-single/architecture/index

Additional information about creating images is available in the OpenShift Container Platform documentation: **Creating Images** - https://access.redhat.com/documentation/en-us/openshift_container_platform/4.2/html/images/index

Labels and label selectors details are available in Working with Kubernetes Objects section for the Kubernetes documentation: **Labels and Selectors** - https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/

6.4. Creating Routes

Goal: Expose a service using an OpenShift route

6.4.1. Working with Routes

It is important to note that services allow for network access between pods in OCP and routes allow for network access to pods from resources outside of OCP.

Routes are created using the **oc expose** command. A route points to a service allowing access of a pod in a static way.

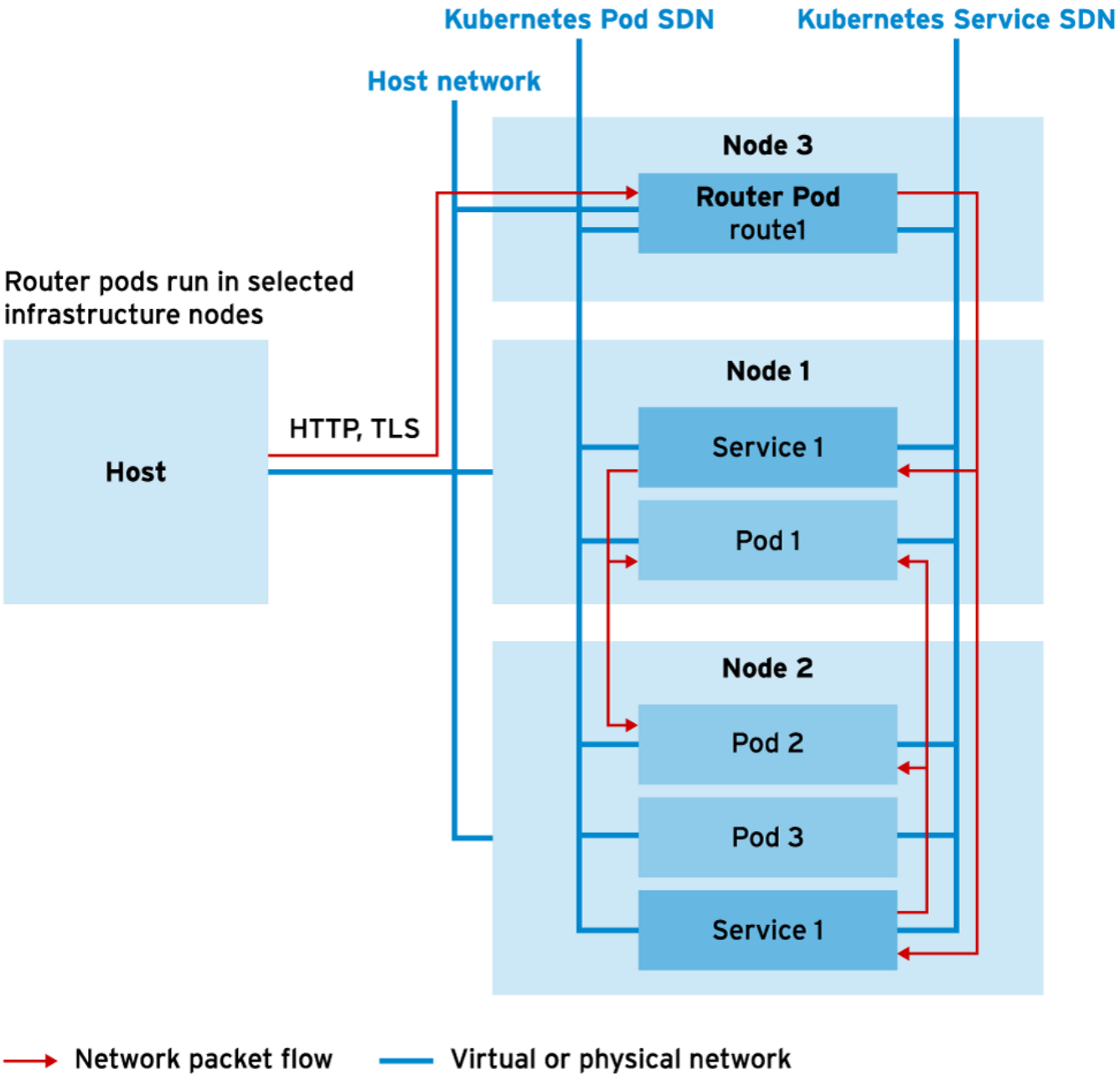

Figure 16. Openshift Routes and Kubernetes Services

Route - Connects a public-facing IP address and DNS hostname to an internal-facing service IP. The router service uses *HAProxy* as the default implementation.

Difference between **service** *and* **route**

A **service** is mean to be consumed internally while a **route** is meant to be consumed externally. A **route** links directly to the service resource name.

6.4.2. Creating Routes

The **oc create** command can be used to create route resources by providing a JSON or YAML file with the proper route resource definitions. The **oc new-app** command doesn't create a route when a pod is built from a container image. It will create a **service** but not a route.

As mentioned earlier, the other way of creating a route is to use **oc expose service** command and passing the service name. It is also possible to name the **route** using the **--name** option.

Listing 124. Exposing a Service and Naming a Route

```
$ oc expose service <service_name> --name <route_name>
```

Default Route Parameteres

By default, routes created with **oc expose** generate a DNS name in this form:

route-name-project-name.default-domain

- **route-name** - Name assigned to route. This is the name provided with the **--name** option.
- **project-name** - Name of project containing resource
- **default-domain** - Name configured as part of OCP and corresponds to the wildcard DNS domain

6.4.2.1. Leveraging the Default Routing Service

The default routing service is implemented with **HAProxy**. Router pods, containers, and the configuration can be inspected by selecting the correct namespace.

Listing 125. Inspecting Router Apps

```
$ oc get pod --all-namespaces -l app=router
```

By default, OCP deploys routers in the **openshift-ingress** project. The **oc describe pod** command can be used to get routing configuration details.

Listing 126. Inspecting the Router Configuration Details

```
$ oc describe pod router-default-UUID
```

TIP Header

The subdomain, or default domain gets the value from **ROUTER_CANONICAL_HOSTNAME** entry. It is also defined by the **subdomain** keyword in the **routingConfig** section of the OCP config file **master-config.yaml**.

6.5. Demonstration - Creating Routes

Example 9. DEMO - Creating a Route to Application

1. Create a project to demo the route

Listing 127. Source Description

```
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-route-demo
Now using project "rhn-gps-tmichett-route-demo" on server "https://api.ocp-na2.prod.nextcle.com:6443".
```

2. Create a new app for OCP

Listing 128. Create a new app with **oc new-app**

```
[student@workstation ~]$ oc new-app \
php:7.1~https://github.com/${RHT_OCP4_GITHUB_USER}/DO180-apps \
--context-dir php-helloworld --name php-demo

... output omitted ...

'oc expose svc/php-demo'
Run 'oc status' to view your app
```

3. Review information about the service

Listing 129. Using **oc describe** *to describe the service*

```
[student@workstation ~]$ oc describe svc
Name:              php-demo
Namespace:         rhn-gps-tmichett-route-demo
Labels:            app=php-demo
Annotations:       openshift.io/generated-by: OpenShiftNewApp
Selector:          app=php-demo,deploymentconfig=php-demo
Type:              ClusterIP
IP:                172.30.39.175
Port:              8080-tcp  8080/TCP
TargetPort:        8080/TCP
Endpoints:         <none>
Port:              8443-tcp  8443/TCP
TargetPort:        8443/TCP
Endpoints:         <none>
Session Affinity:  None
Events:            <none>
```

4. Expose the Service

Listing 130. Use **oc expose** *command to expose the service*

```
[student@workstation ~]$ oc expose svc/php-demo --name ch6-route-demo
route.route.openshift.io/ch6-route-demo exposed
```

5. Describe the Route

Listing 131. Use **oc describe route**

```
[student@workstation ~]$ oc describe route
Name:           ch6-route-demo
Namespace:      rhn-gps-tmichett-route-demo
Created:        16 seconds ago
Labels:         app=php-demo
Annotations:        openshift.io/host.generated=true
Requested Host:     ch6-route-demo-rhn-gps-tmichett-route-demo.apps.ocp-na2.prod.nextcle.com
              exposed on router default (host apps.ocp-na2.prod.nextcle.com) 16 seconds ago
Path:           <none>
TLS Termination:    <none>
Insecure Policy:    <none>
Endpoint Port:      8080-tcp

Service:    php-demo
Weight:     100 (100%)
Endpoints:  <none>
```

6. Verify the route has been exposed and that you can connect to it

Listing 132. Verifying Exposed Route

```
[student@workstation ~]$ curl ch6-route-demo-rhn-gps-tmichett-route-demo.apps.ocp-na2.prod.nextcle.com
Hello, World! php version is 7.1.30
```

There is a small time delay

It may take a few minutes for everything to build and be exposed.

7. Cleanup the project

Listing 133. Use **oc delete** *to remove the project*

```
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-route-demo
project.project.openshift.io "rhn-gps-tmichett-route-demo" deleted
```

References

Additional information about the architecture of routes in OpenShift is available in the *Architecture and Developer Guide* sections of the **OpenShift Container Platform documentation** - https://access.redhat.com/documentation/en-us/openshift_container_platform/

6.6. Creating Applications with Source-to-Image

Goal: Deploy an application using Source-to-Image (S2I) facility of OCP.

6.6.1. The Source-to-Image (S2I) Process

Source-to-Image (S2I) is a tool allowing you to build container images from application source code. The S2I tool takes source code from a Git repository and injects the code into a base container based on the language and framework desired for the source code being used.

The figure below shows the **oc new-app** creating a BuildConfig for deploying an application with S2I.

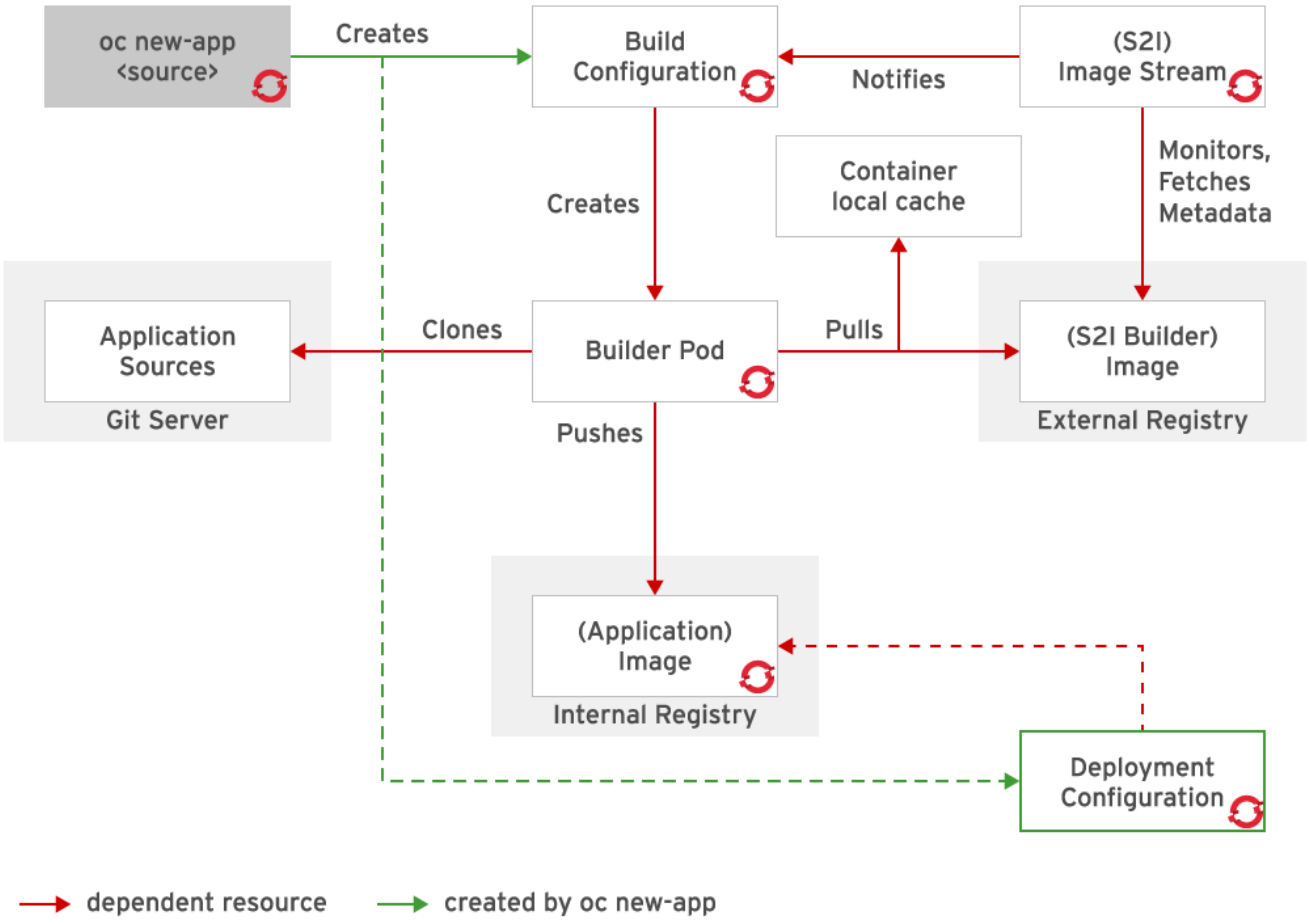

Figure 17. Deployment Configuration and Resource Dependence

S2I is the primary strategy for building applications in OCP.

Reasons to use S2I

- **User efficiency** - Developers can work using standard programming tools and don't need to understand system administration

- **Patching** - Allows rebuilding of applications consistently if base image needs a patch
- **Speed** - Assembly process performs large number of operations without creating new layers for each step
- **Ecosystem** - Encourages sharing and re-use of base images and scripts across multiple applications

6.6.2. Describing Image Streams

An image stream is a way of collecting multiple versions of an image under a single alias. The **image stream resource** is a configuration that names container images associated with an **image stream tag** as aliases for container images.

An S2I build is performed to the entire image stream.

6.6.3. Building an Application with S2I and the CLI

Building applications with S2I can be done with the **oc new-app** command.

Listing 134. Building an Application from Source and Image Stream

```
$ oc new-app -i php http://my.git.server.com/my-app --name=myapp
```

Build configurations (**bc**) are responsible for defining input parameters and triggers to transform code into runnable images. The **BuildConfig (bc)** is the second resource in a definition file identified by ***"kind": "BuildConfig"*** resource definition file.

DeploymentConfig (dc) is the third resource in a deployment configuration responsible for customizing the deployment process in OCP.

DeploymentConfig Objects

- User customizable strategies to transition existing deployment to new deployments
- Rollbacks to a previous deployment
- Manual replication scaling

Monitoring Builds and Obtaining Logs

After a new application is created, the build process can be viewed by using the **oc get builds** command. You can see logs with the **oc logs build/appname** command.

A new build can be triggered with the **oc start-build <build_config_name>** command. This will initialize a new build.

6.6.4. Relationship Between Build and Deployment Configurations

The **BuildConfig** pod is responsible for creating images in OCP and pushing them to the internal container registry. The **DeploymentConfig** pod is responsible for deploying pods to OCP. The outcome of **DeploymentConfig** pod execution is the creation of pods with images deployed in the internal container registry.

The **BuildConfig** creates an image and pushes it to the container registry. The **DeploymentConfig** reacts to images that **BuildConfig** changes for the registry and deploys the pods to OCP defining various triggers and replicas.

References

Source-to-Image (S2I) Build: https://access.redhat.com/documentation/en-us/openshift_container_platform/4.2/html/builds/build-strategies#build-strategy- s2i_build-strategies

S2I GitHub repository: https://github.com/openshift/source-to-image

6.7. Creating Applications with the OpenShift Web Console

Goals

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

6.7.1. Accessing the OpenShift Web Console

The OCP Web Console allows users to execute the same tasks as the OCP CLI (**oc**) command. It is accessed by the following URL: https://consoleopenshift-console.{wildcard DNS domain for the RHOCP cluster}/

6.7.1.1. Managing Projects

Projects can be managed from the OCP Home page. The **Projects Status** page shows all applications created within a project space.

Note on SSL Certificates

It is necessary to trust two certificates for managing an OCP environment. There is one certificate for the OCP Console and there is another certificate for the REST API

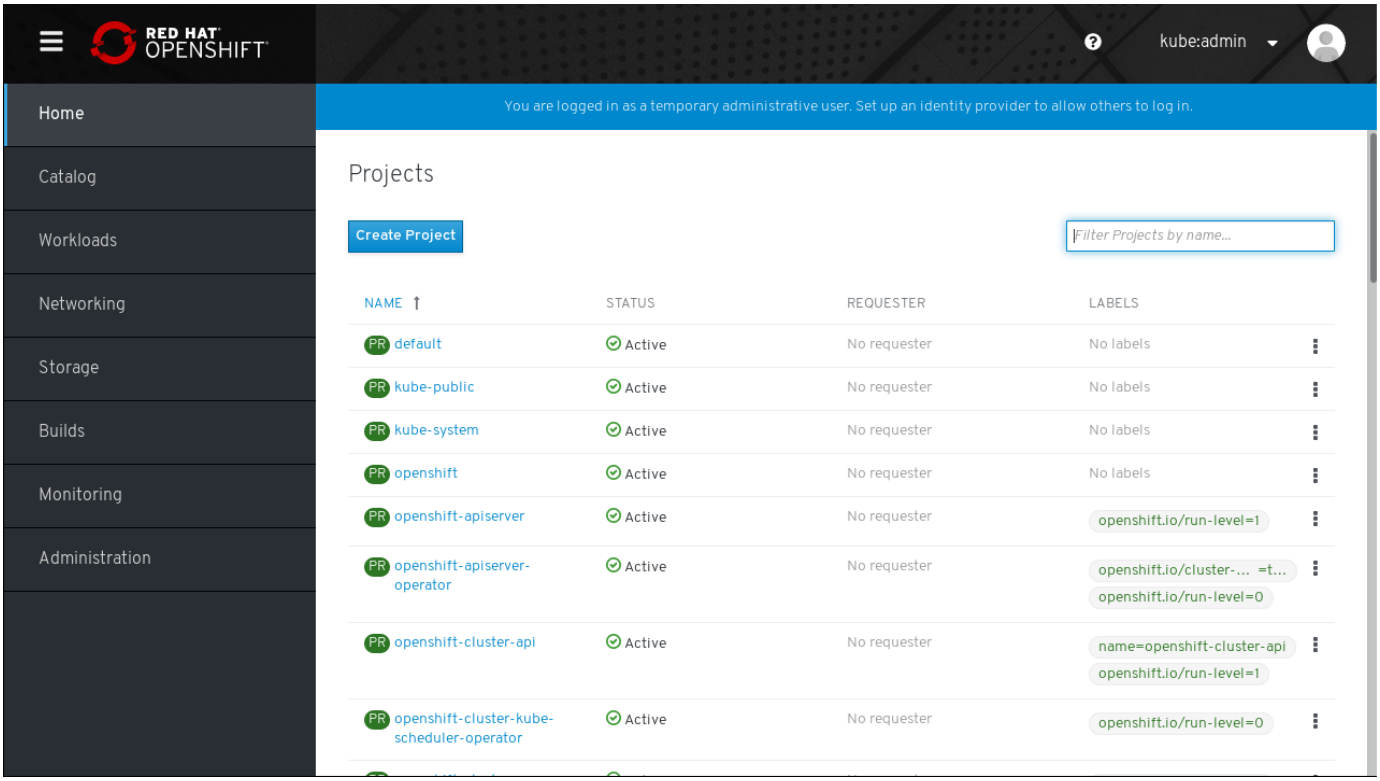

Figure 18. OpenShift Web Console

6.7.1.2. Navigating the Web Console

OCP Web Console menus are located on the left side and expand for the sub-menus providing related management functions.

- **Catalog** - Navigate multiple image streams
- **Workloads** - Access to resources and pods in a DeploymentConfig
- **Networking** - Used to manage Services and Routes
- **Storage** - Create and manage persistent volumes
- **Builds**
  - **Build Configs** - Displays list of project build configurations
  - **Builds** - Provides list of recent build processes and allows access to logs
  - **Image Streams** - Provides list of image streams defined in a project
- **Monitoring** - Provides access to manage OCP alerts
- **Adminstration** - Control and manages resource quotas as well as other cluster/project settings.

6.7.2. Creating New Applications

Provides a click-through experience for creating an application. Applications can be deployed using the **Developer Catalog** or using various S2I templates.

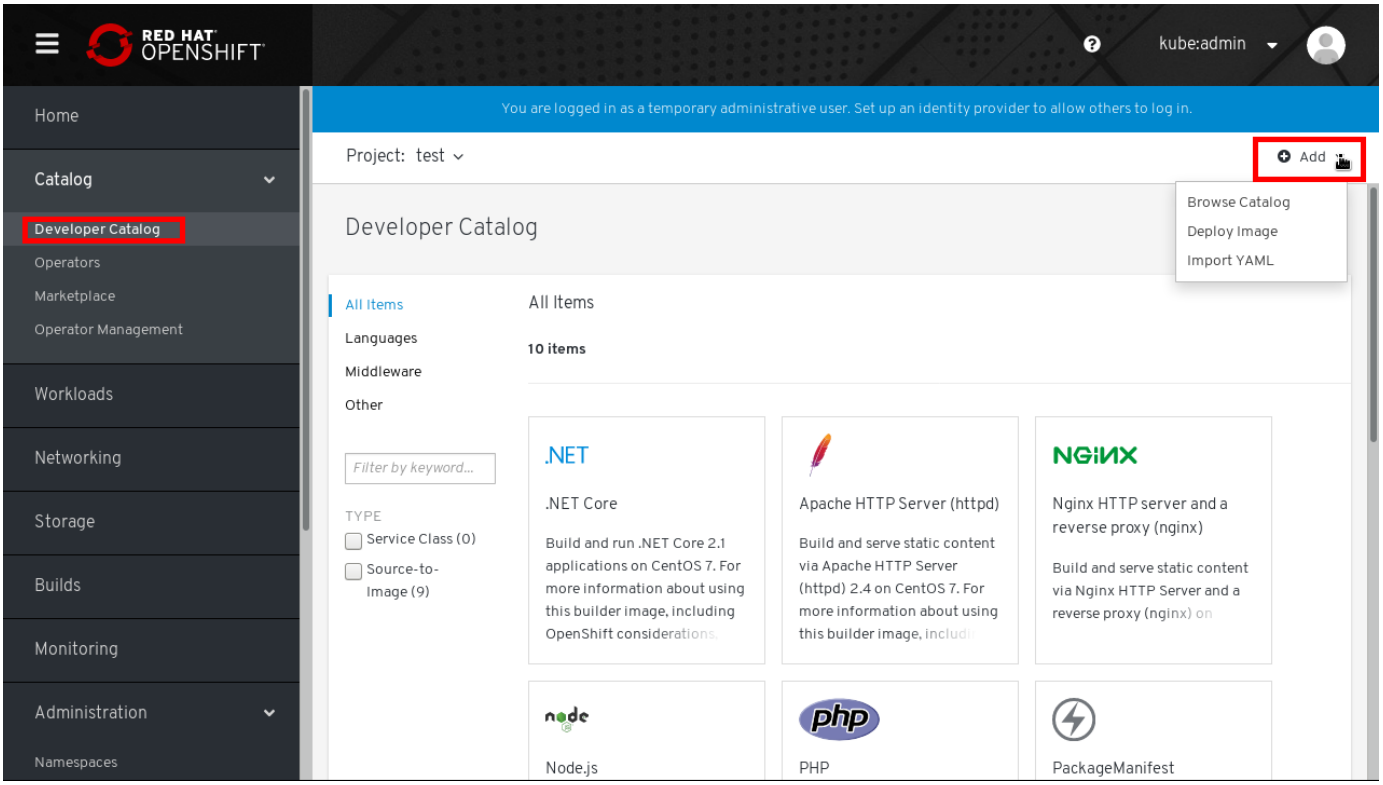

Figure 19. OpenShift Developer Catalog

6.7.2.1. Managing Application Builds

The **BuildConfigs** sub-menu of the **Builds** menu allows you to do the following:

- View build configuration parameters
- View and edit environment variables
- View a list of recent application builds and access logs from the build process

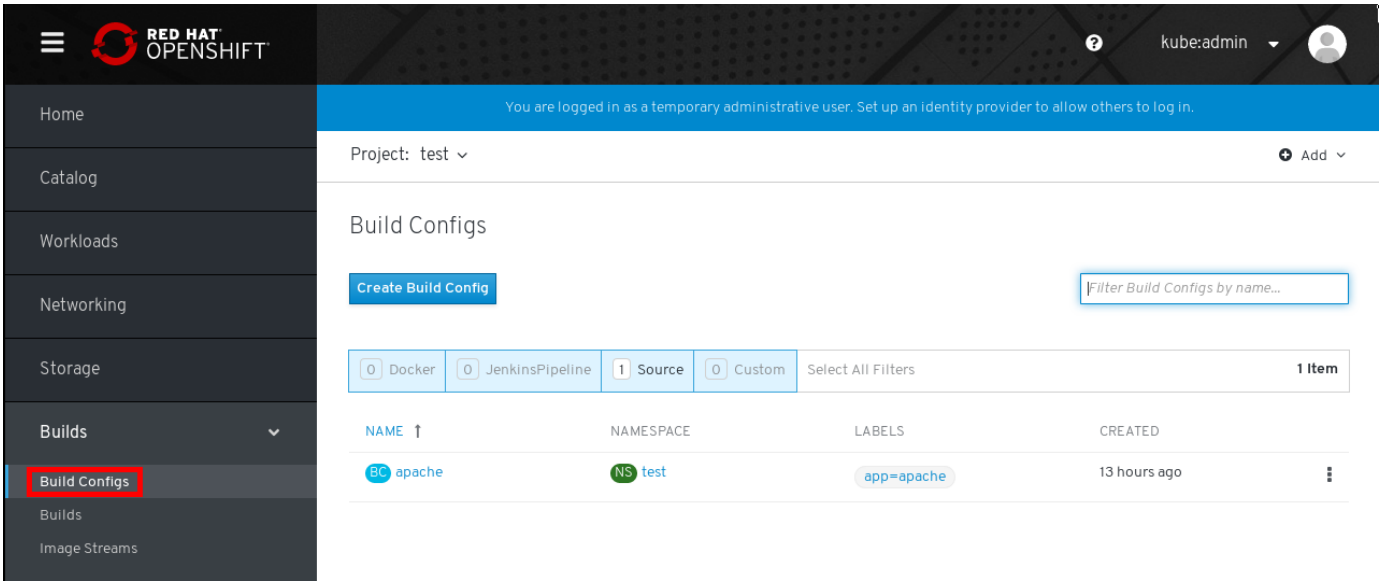

Figure 20. OpenShift Build Configurations

6.7.3. Managing Deployed Applications

The **Workloads** menu provides access to deployment configurations for the project. From here you can do the following:

- View deployment configuration parameters
- Change number of application pods for scaling
- View and edit environment variables
- View a list of application pods and access logs for that pod

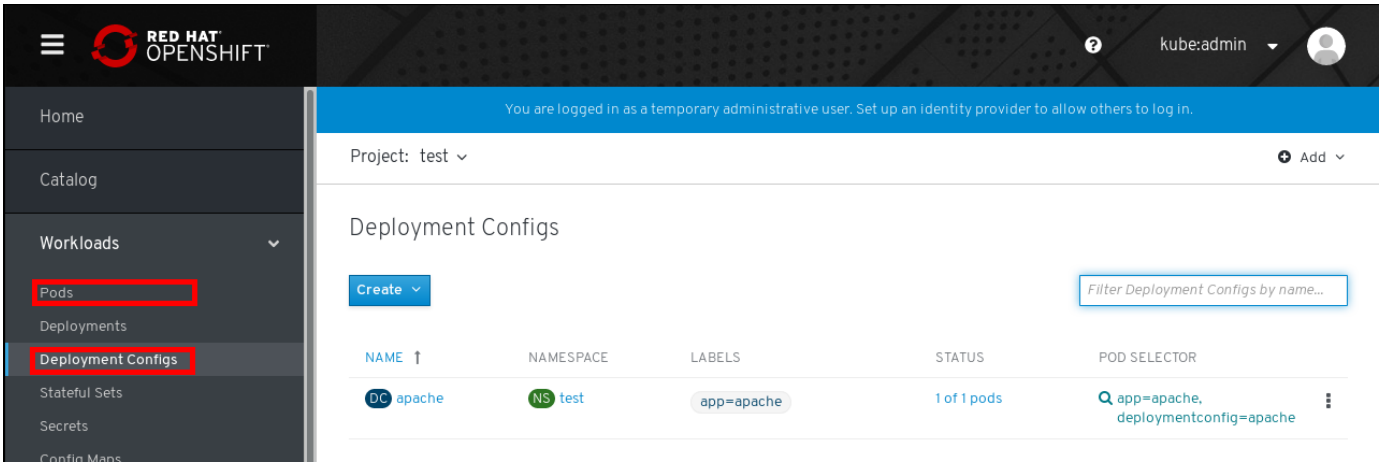

Figure 21. OpenShift Workloads Menu / Deployment Configurations

6.7.4. Other Web Console Features

- Manage resources - project quotas, user membership, secrets, and other advanced resources
- Create persistent volume claims
- Monitor builds, deployments, pods, and system events
- Create CI/CD pipeline with Jenkins :pygments-style: tango :source-highlighter: pygments :toc: :toclevels: 7 :sectnums: :sectnumlevels: 6 :numbered: :chapter-label: :icons: font :imagesdir: images/

7. Deploying Multi-Container Applications

7.1. Considerations for Multi-Container Applications

Goals

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

7.1.1. Leveraging Multi-Container Applications

Kubernetes and OCP provide tools to facilitate container orchestration. This eliminates complication and the need to manually manage containers. Container orchestration becomes even more important for multi-container applications as restarts of these applications can often break functionality.

7.1.2. Discovering Services in a Multi-Container Application

Podman uses Container Network Interface (CNI) to create software-defined-networks (SDN) between the containers and the host. CNI will assign new IP addresses when a container starts. Each container exposes all ports to other containers in the same SDN so services are readily available and accessible.

Important Header

Containers with dynamic IP addresses become difficult to manage when working with multicontainer applications because each container needs to be able to communicate with other containers to use services.

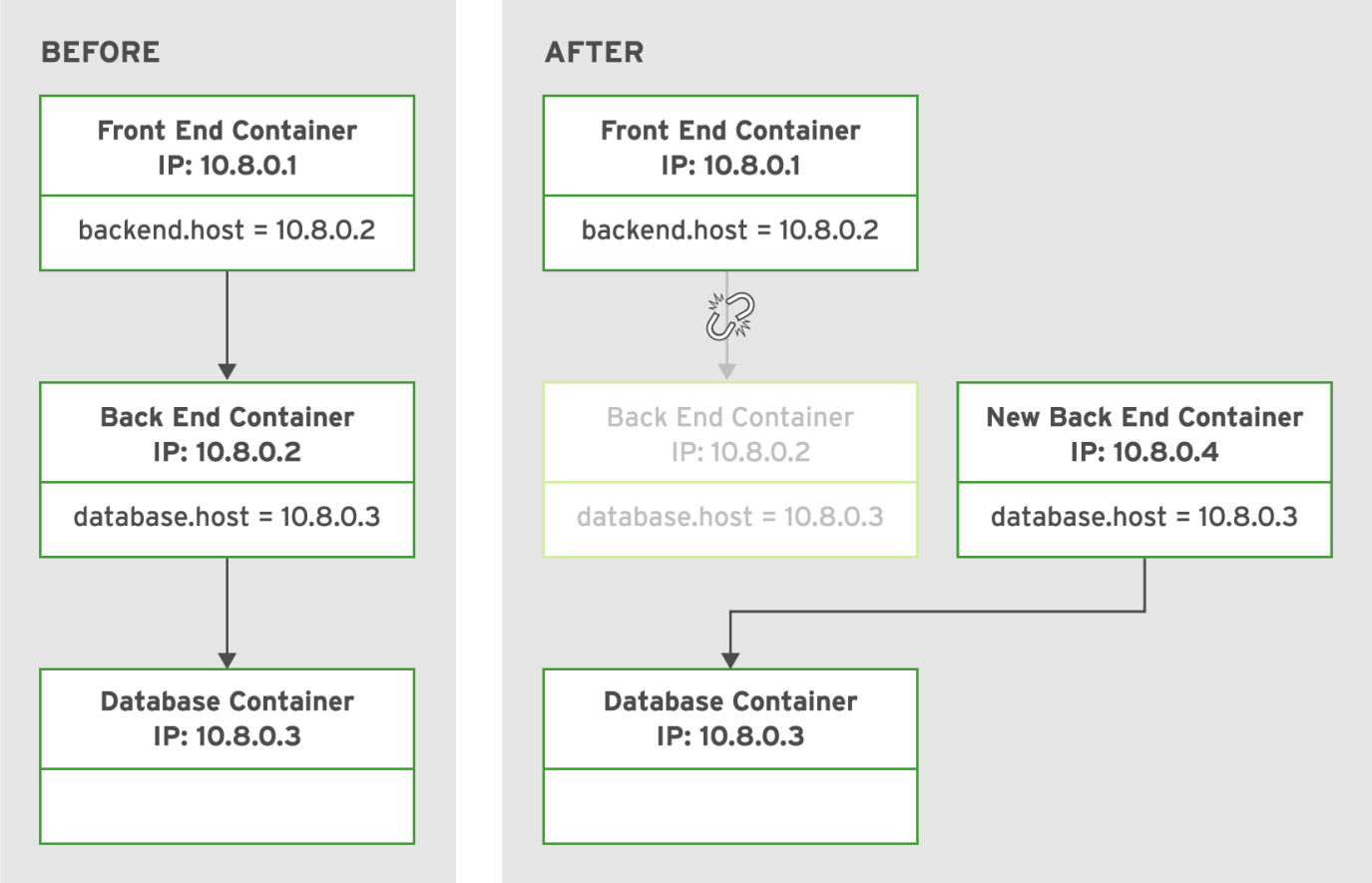

Figure 22. Three-tiered Application Links

In the above diagram, a three container application has a backend container getting reset. This backend container receives a new IP address breaking communication for the application. Kubernetes and OCP provide mechanisms for service discoverability for dynamic network changes.

7.1.3. Comparing Podman and Kubernetes

Pods get attached to a Kubernetes namespace, in OCP, this is called a **project**. Kubernetes assign resources to services defined in the namespace and generates the corresponding environment variables.

Kubernetes Environment Variable Conventions

- Uppercase
- Snakecase - environmnent variables created by a service with multiple words separated with _
- Service name first
- Protocol type

service *Benefits*

A **service** allows for pointing to a service instead of directly to a container or IP address.

7.1.4. Describing the To Do List Application

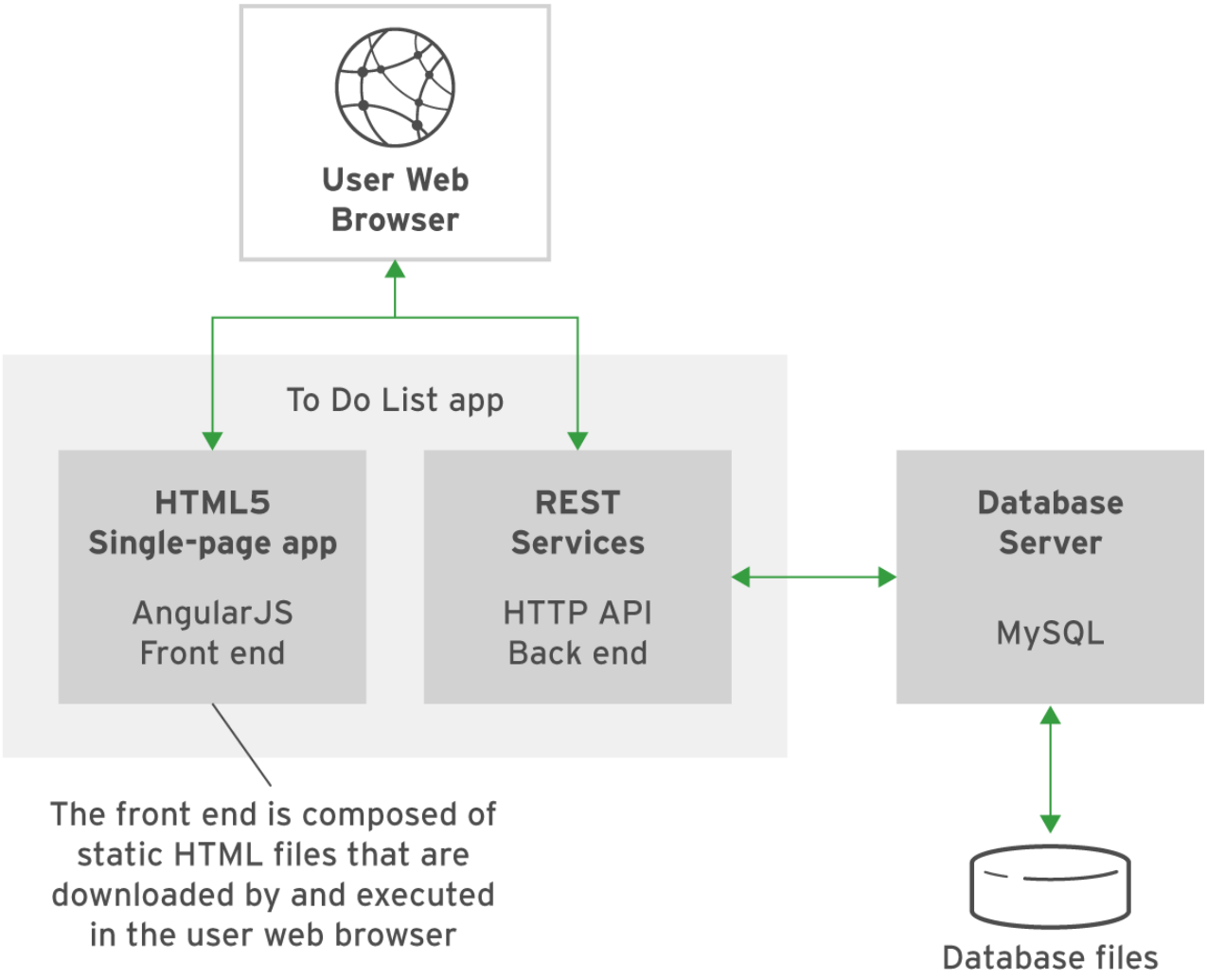

Figure 23. Logical Architecture of To Do List Application

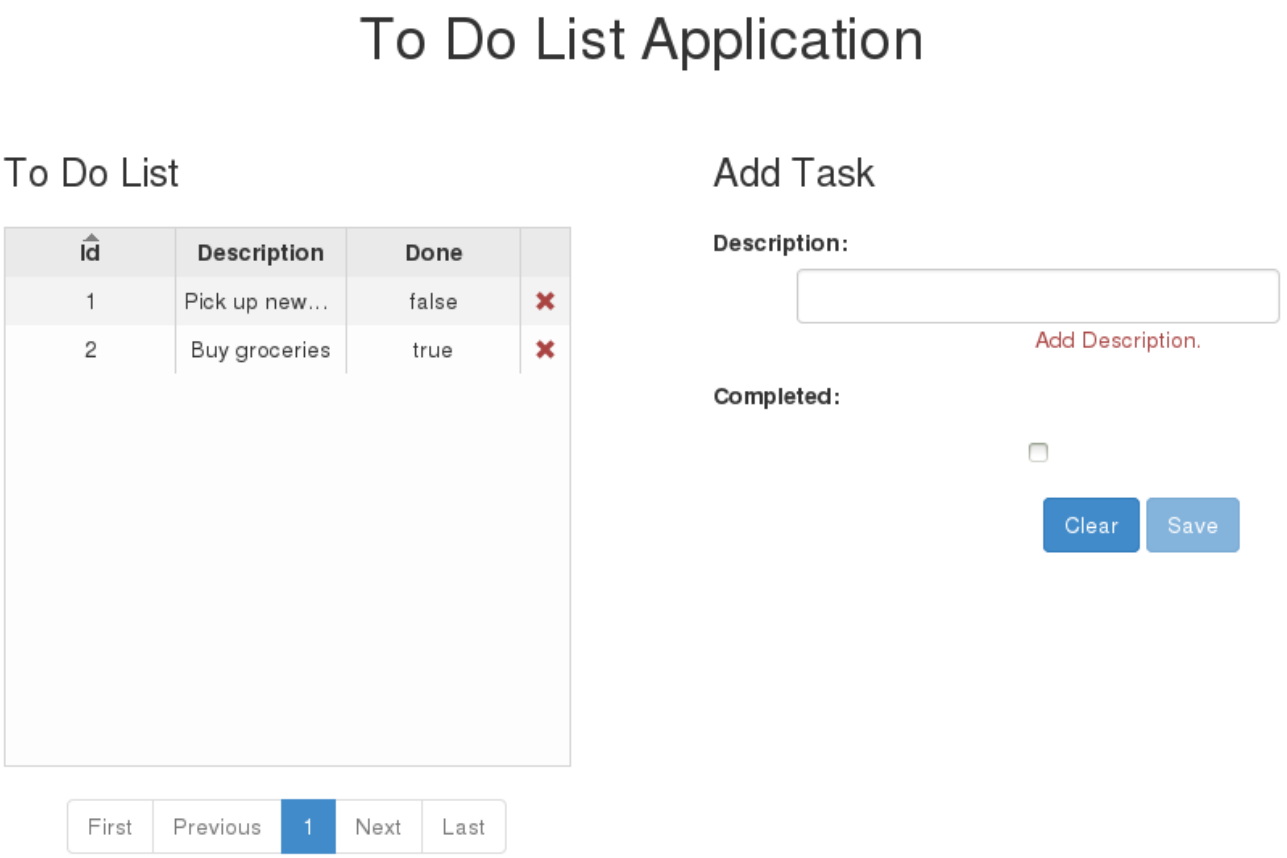

Figure 24. To Do List Application

Red Hat DO180 Training Repository

https://github.com/RedHatTraining/DO180-apps.git.

7.2. Deploying a Multi-Container Application on OpenShift

Goal: Be able to deploy a multicontainer application on OpenShift using a template.

7.2.1. Examining the Skeleton of a Template

Web applications might require a **BuildConfig**, **DeploymentConfig**, **Service**, and **Route** resource to run an OCP project. OCP **templates** provide a way to simplify creation of application resources. A template will create resources in a repeatable fashion.

There are several default templates created by the OpenShift installer in the **openshift** namespace. The **oc get templates** command with the **-n openshift** option can list preinstalled templates.

```
[student@workstation ~]$ oc get templates -n openshift
```

You can use **openshift get template <name> -n openshift -o yaml** to display the definition of a template file.

Listing 135. YAML Source File of Mysql-Persistent Template

```
[student@workstation ~]$ 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: DeploymentConfig metadata:
annotations: ...annotations omitted...
name: ${DATABASE_SERVICE_NAME} spec: ...spec 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...
```

It is also possible to create a template using the **oc create** command and specifying the corresponding YAML file with the template definition.

Listing 136. Creating a **todo-template**

```
[student@workstation deploy-multicontainer]$ oc create -f todo-template.yaml template.template.openshift.io/todonodejs-persistent created
```

The creation of the template above will create a template under the current project. It is also possible to specify the name of the project where the template should be created with the **-n** option and the project name.

Listing 137. Source Description

```
[student@workstation deploy-multicontainer]$ oc create -f todo-template.yaml \ > -n openshift
```

openshift *Namespace and Templates*

Any template created under the **openshift** namespace (OpenShift project) is available in the web console under the dialog box accessible in the **Catalog → Developer Catalog** menu item. Moreover, any template created under the current project is accessible from that project.

7.2.1.1. Parameters

OCP templates can define parameters which can be assigned values. There are two methods of using listing available parameters from a template.

- **oc describe**
- **oc process** with the **--parameters** option

7.2.2. Processing a Template Using the CLI

The **oc process** commands allows processing of a template and can return output in various formats. The process of the template file is either in JSON or YAML and the output that is returned is typically JSON, although it is possible to use **-o yaml** tp have the output as YAML.

Listing 138. Processing Template with **oc process** *and Output as JSON*

```
$ oc process -f <filename>
```

Listing 139. Processing Template with **oc process** *and Output as YAML*

```
$ oc process -o yaml -f <filename>
```

Redirecting **oc process** *Output to a File*

The **oc process** command returns a list of resources to standard output. This output can be redirected to a file.

```
$ oc process -o yaml -f filename > myapp.yaml
```

Templates generate resources with configurable attributes based on parameters. The **-p** option followed by **<name⇐<value>** pair. This can be done as a two-step process by creating a new template file with the new parameters and then using this new file to create the applicaiton.

1. Create new template based on **oc process**

Listing 140. Creating a new processed template

```
$ 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
```

2. Create the application with **oc create**

Listing 141. Creating Application from Processed Template

```
$ oc create -f mysqlProcessed.yaml
```

Single Step Processing and Creation of Application

It is also possible to generate an application from a template without first saving the intermediate template file using **oc process** and **oc create** via a UNIX pipe.

Listing 142. Generating an Application with Custom Template Parameters

```
$ oc process -f mysql.yaml -p MYSQL_USER=dev \
> -p MYSQL_PASSWORD=$P4SSD -p MYSQL_DATABASE=bank \ > -p VOLUME_CAPACITY=10Gi | oc create -f -
```

7.2.3. Configuring Persistent Storage for OpenShift Applications

OCP manages persistent storage as a set of pooled, cluster-wide resources. In order to present storage to a cluster, a PersistentVolume must be created and defined.

Listing 143. Listing **PersistenVolume** *Objects*

```
[admin@host ~]$ oc get pv
```

The **oc get pv** command can list PersistentVolume objects in a cluster. To see the YAML definition, use the **oc get** command with the **-o yaml** option.

Listing 144. PersistentVolume Definition

```
[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
```

The **oc create** command can add additional **PersistentVolumes** to a cluster

Listing 145. Creating and Adding New Volumes

```
[admin@host ~]$ oc create -f pv1001.yaml
```

7.2.3.1. Requesting Persistent Volumes

When applications need storage, a **PersistentVolumeClaim (PVC)** object must be requested. The easiest way to request a PVC claim is using a YAML file and the **oc create** command. It is possible to also list PVC requests with the **oc get pvc** command.

Listing 146. Requesting a PVC

```
[admin@host ~]$ oc create -f pvc.yaml
```

Listing 147. Listing PVC Requests

```
[admin@host ~]$ oc get pvc
```

7.2.3.2. Configuring Persistent Storage with Templates

Templates requiring a persistent storage volume should have a suffix of **-persistent**. An easy way to search for templates in the **openshift** project is below.

Listing 148. Listing Persistent Templates

```
[student@workstation ~]$ oc get templates -n openshift | grep persistent
```

Important Header

In a template definition under DeploymentConfig, a **PersistentVolumeClaim** must exist in order for the template to request a persistent volume.

References

Developer information about templates can be found in the Using Templates section of the OpenShift Container Platform documentation: **Developer Guide** - https://access.redhat.com/documentation/en-us/openshift_container_platform/4.2/html-single/images/index#using-templates

8. Troubleshooting Containerized Applications

8.1. Troubleshooting S2I Builds and Deployments

Goals

- Troubleshoot an application build and deployment steps on OpenShift
- Analyze OpenShift logs to identify problems during the build and deploy process

8.1.1. Introduction to the S2I Process

Source-to-Image (S2I) allows images to automatically be created based on the programming language of the application source code in OCP. The S2I process has two major steps.

Source-to-Image Steps

- **Build Step** - Compiles source code, downloads library dependencies, and packages application as a container image. The **BuildConfig (BC)** is responsible for the build step.
- **Deployment Step** - Starts a pod and makes sure application is available for OCP. The **DeploymentConfig (DC)** is responsible for the deployment step.

Unique S2I Processes

For each S2I process, every application uses its own **BC** and **DC** Pods start in order to complete the build and the deployment process aborts if the build fails.

Each S2I process is started in a separate pod. After a successful build, the application starts on a separate pod. It is possible to look at the **Builds** section of the WebUI and get detailed information about the builds, including logs.

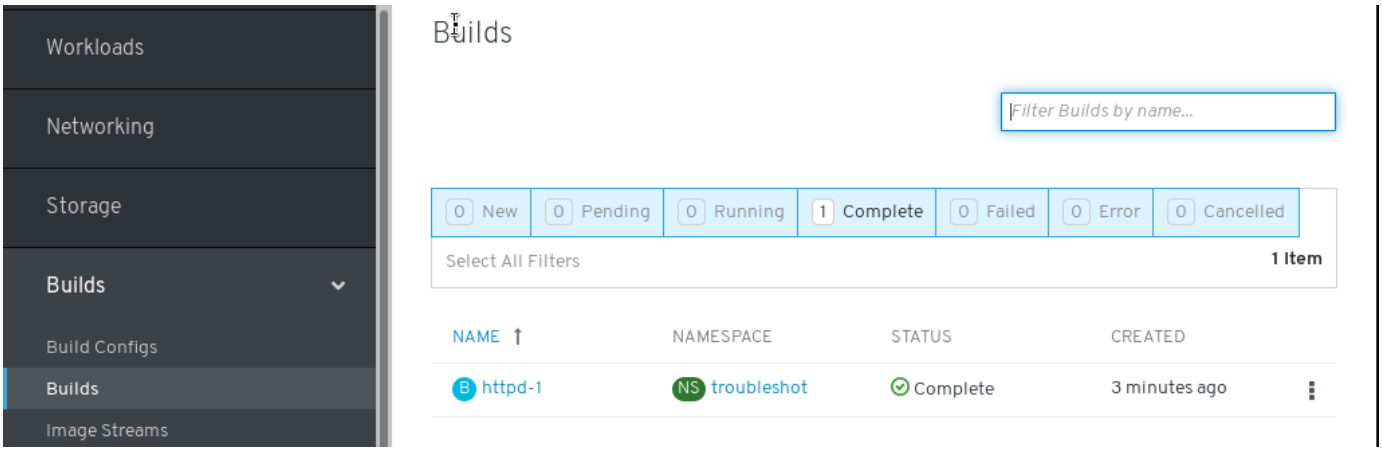

Figure 25. Project Build Instances

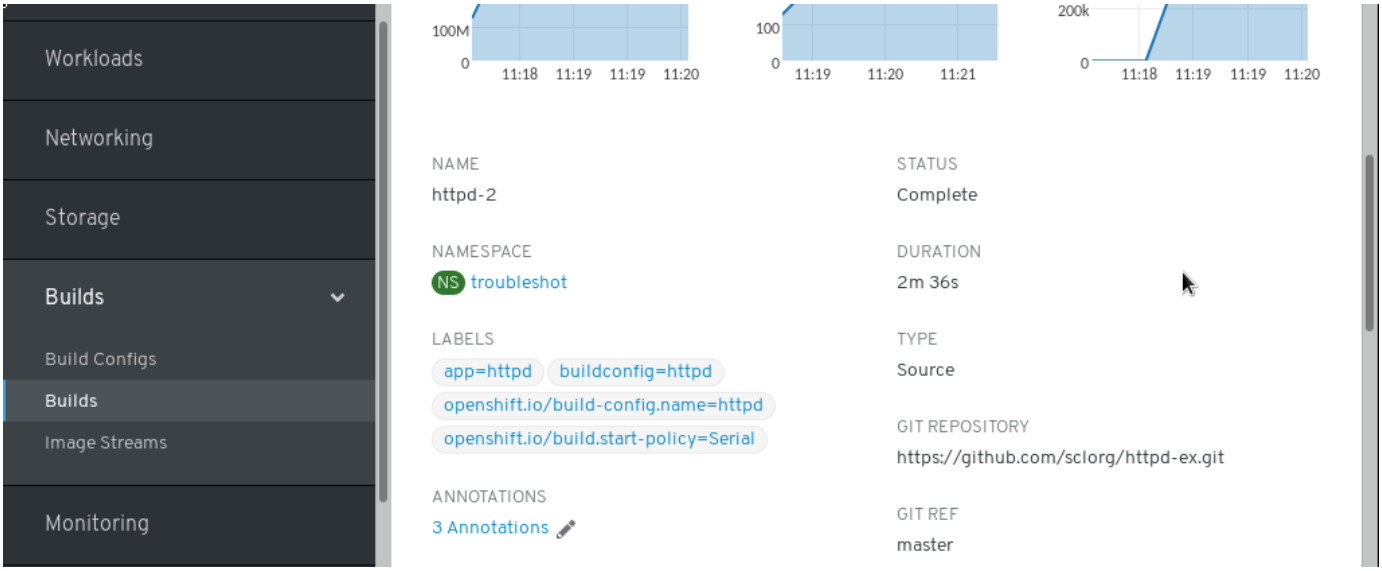

Figure 26. Detailed Build Instance View

It is also possible to build and retrieve logs from the command line as well using the **oc logs** command and the **oc start-build** commands.

Listing 149. Retrieving Build Logs

```
$ oc logs bc/<application-name>
```

Listing 150. Starting a Build

```
$ oc start-build <application-name>
```

Listing 151. Retrieving Deployment Logs

```
$ oc logs dc/<application-name>
```

8.1.2. Describing Common Problems

The **oc logs** command provides information about the build, deploy, and run processes of an application during pod execution. Logs can often indicate missing values or options that must be included/enabled, as well as any incorrect parameters or environment incompatibilities.

8.1.2.1. Troubleshooting Permission Issues

Permission issues are often one of the most common problems. Additionally, RHEL7 enforces SELinux policies which further restrict filesystem resources, network ports, and processes. Sometimes, containers also require runnign with a specific UID or other policy.

The **oc adm policy** command can be used to relax OCP project security and test for permission errors.

Listing 152. Modifying OCP Policy for SELinux and UID

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

8.1.2.2. Troubleshooting Invalid Parameters

Shared parameters for multi-container applications are also a common issue. It is a good practice to centralize and store shared parameters in **ConfigMaps**. These **ConfigMaps** can be injected by the **Deployment Config** into containers as environment variables. By using this method, a **ConfigMap** can be injected into different containers ensuring the environment variables are available and defined the same.

8.1.2.3. Troubleshooting Volume Mount Errors

Sometimes it is possible that a persistent volume can't be mounted even though a claim is released. To resolve these issues, delete the PV claim and then recreate the PV.

Listing 153. Fixing a Persistent Volume

```
oc delete pv <pv_name>
oc create -f <pv_resource_file>
```

8.1.2.4. Troubleshooting Obsolete Images

OCP pulls images from the source unless it exists locally. In order to manage images and ensure that the most current image is used, it is a good practice to perform image cleanup. The **podman rmi** command can remove images and be scheduled. Additionally, you can use **oc adm prune** as an automated way to remove obsolete images and resources.

References

More information about troubleshooting images is available in the Images section of the OpenShift Container Platform documentation accessible at: **Creating Images** - https://docs.openshift.com/container-platform/4.2/openshift_images/create- images.html

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** - https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod- configmap/#define-container-environment-variables-using-configmap-data

8.2. Troubleshooting Containerized Applications

Goals

- Implement techniques for troubleshooting and debugging containerized applications
- Use the port-forwarding feature of OCP client tool
- View container logs
- View OCP cluster events

8.2.1. Forwarding Ports for Troubleshooting

podman can use port forwarding with the **-P** command with the **run** subcommand. However, this only provides port-forwarding during runtime.

Listing 154. Sample Port Forwarding

```
$ sudo podman run --name db -p 30306:3306 mysql
```

OpenShift provides the ability to use port-fowarding with the **oc port-forward** command which forwards a local port to a pod port.

Port Forwarding Differences

- Port-Forwarding maps only exist on workstation where **oc** client runs
- Port-Forwarding only maps a port to a single pod, so load-balanced services on multiple pods are not tested

Listing 155. Port Forwarding with **oc port-forward**

```
$ oc port-forward db 30306 3306
```

podman *vs OpenShift Port Forwarding*

The **podman run -p** can only be used for port-forwarding when the container gets started. The **oc port-forward** command can be used anytime within the container lifecycle.

8.2.2. Enabling Remote Debugging with Port Forwarding

Another feature for port forwarding is to enable remote debugging.

8.2.3. Accessing Container Logs

Both **podman** and OpenShift provide the ability to view logs in running containers and pods. However, these utilities require that applications are configured to send all logging output to standard output.

Listing 156. Using **podman** *to get Container Logs*

```
$ podman logs <containerName>
```

Listing 157. Using **oc logs** *to get Container Logs*

```
$ oc logs <podName> [-c <containerName>]
```

8.2.4. OpenShift Events

OpenShift events are high-level logging and not as log-level as logs making troubleshooting a little less difficult. To read these logs, use the **oc get events** command.

```
$ oc get events
```

Events from the **oc get events** command above aren't filtered and span the entire OCP cluster. The **oc describe** command can be used to retrieve events related to a specific pod.

Listing 158. Using **oc describe** *to Obtain Pod Specific Events*

```
$ oc describe pod <podname>
```

8.2.5. Accessing Running Containers

Both **podman** and OpenShift provide the **exec** subcommand which allows creation of new processes inside of a running container.

Listing 159. Using **podman exec**

```
$ sudo podman exec [options] container command [arguments]
```

Listing 160. Using **oc exec**

```
$ oc exec [options] pod [-c container] -- command [arguments]
```

It is possible to provide the **it** option to the **exec** subcommand which will provide a single interactive command or can be used to start an interactive shell.

```
$ oc exec -it <container> /bin/bash
```

8.2.6. Overriding Container Binaries

Most container images don't contain all troubleshooting commands and binaries needed by administrators to inspect and manage the containers. It is possible for administrators to temporarily access missing commands by mounting host binaries folders as volumes inside the command. This must be done at the launching of the container.

Listing 161. Overriding the Container Image **/bin** *Folder*

```
$ sudo podman run -it -v /bin:/bin image /bin/bash
```

8.2.7. Transferring Files To and Out of Containers

It can also be necessary to retrieve or transfer files in/out of running containers. The **podman cp** command is capable of transferring and copying files into and out of a running container.

Listing 162. Using **podman cp** *to Copy a File into a Container*

```
$ sudo podman cp standalone.conf todoapi:/opt/jboss/standalone/conf/
```

Listing 163. Using **podman cp** *to Copy a File from a Container*

```
$ sudo podman cp todoapi:/opt/jboss/standalone/conf/standalone.conf .
```

It is also possible to use **podman exec** with UNIX pipes to pass files into and out of containers. This is more useful when you have other utilities that create data such as **mysqldump**

OpenShift Copy Equivalent

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** - https://access.redhat.com/documentation/en-us/openshift_container_platform/4.2/html-single/architecture/index/

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** - https://access.redhat.com/documentation/en-us/openshift_container_platform/4.2/html-single/applications/index/