Chapter 3.  Run Applications as Containers and Pods

[**Create Linux Containers and Kubernetes Pods**](https://rha.ole.redhat.com/rha/app/courses/do180-4.14/pages/ch03/c2358540-87d5-48de-b49e-6f23bdcd629c)

[**Guided Exercise: Create Linux Containers and Kubernetes Pods**](https://rha.ole.redhat.com/rha/app/courses/do180-4.14/pages/ch03s02/c2358540-87d5-48de-b49e-6f23bdcd629c)

[**Find and Inspect Container Images**](https://rha.ole.redhat.com/rha/app/courses/do180-4.14/pages/ch03s03/c2358540-87d5-48de-b49e-6f23bdcd629c)

[**Guided Exercise: Find and Inspect Container Images**](https://rha.ole.redhat.com/rha/app/courses/do180-4.14/pages/ch03s04/c2358540-87d5-48de-b49e-6f23bdcd629c)

[**Troubleshoot Containers and Pods**](https://rha.ole.redhat.com/rha/app/courses/do180-4.14/pages/ch03s05/c2358540-87d5-48de-b49e-6f23bdcd629c)

[**Guided Exercise: Troubleshoot Containers and Pods**](https://rha.ole.redhat.com/rha/app/courses/do180-4.14/pages/ch03s06/c2358540-87d5-48de-b49e-6f23bdcd629c)

[**Lab: Run Applications as Containers and Pods**](https://rha.ole.redhat.com/rha/app/courses/do180-4.14/pages/ch03s07/c2358540-87d5-48de-b49e-6f23bdcd629c)

[**Quiz: Run Applications as Containers and Pods**](https://rha.ole.redhat.com/rha/app/courses/do180-4.14/pages/ch03s08/c2358540-87d5-48de-b49e-6f23bdcd629c)

[**Summary**](https://rha.ole.redhat.com/rha/app/courses/do180-4.14/pages/ch03s09/c2358540-87d5-48de-b49e-6f23bdcd629c)

**Abstract**

|  |  |
| --- | --- |
| **Goal** | Run and troubleshoot containerized applications as unmanaged Kubernetes pods. |
| **Objectives** | * Run containers inside pods and identify the host OS processes and namespaces that the containers use. * Find containerized applications in container registries and get information about the runtime parameters of supported and community container images. * Troubleshoot a pod by starting additional processes on its containers, changing their ephemeral file systems, and opening short-lived network tunnels. |
| **Sections** | * Create Linux Containers and Kubernetes Pods (and Guided Exercise) * Find and Inspect Container Images (and Guided Exercise) * Troubleshoot Containers and Pods (and Guided Exercise) |
| **Lab** | * Run Applications as Containers and Pods |

Create Linux Containers and Kubernetes Pods

Objectives

* Run containers inside pods and identify the host OS processes and namespaces that the containers use.

Creating Containers and Pods

Kubernetes and OpenShift offer many ways to create containers in pods. You can use one such way, the run command, with the kubectl or oc CLI to create and deploy an application in a pod from a container image. A *container image* contains immutable data that defines an application and its libraries.

**Note**

Container images are discussed in more detail elsewhere in the course.

The run command uses the following syntax:

**oc run *RESOURCE/NAME* --image *IMAGE* [options]**

For example, the following command deploys an Apache HTTPD application in a pod named web-server that uses the registry.access.redhat.com/ubi8/httpd-24 container image.

[user@host ~]$ **kubectl run web-server --image registry.access.redhat.com/ubi8/httpd-24**

You can use several options and flags with the run command. The --command option executes a custom command and its arguments in a container, rather than the default command that is defined in the container image. You must follow the --command option with a double dash (--) to separate the custom command and its arguments from the run command options. The following syntax is used with the --command option:

**oc run *RESOURCE/NAME* --image *IMAGE* --command -- *cmd* *arg1* ... *argN***

You can also use the double dash option to provide custom arguments to a default command in the container image.

**kubectl run *RESOURCE/NAME* --image *IMAGE* -- *arg1* *arg2* ... *argN***

To start an interactive session with a container in a pod, include the -it options before the pod name. The -i option tells Kubernetes to keep open the standard input (stdin) on the container in the pod. The -t option tells Kubernetes to open a TTY session for the container in the pod. You can use the -it options to start an interactive, remote shell in a container. From the remote shell, you can then execute additional commands in the container.

The following example starts an interactive remote shell, /bin/bash, in the default container in the my-app pod.

[user@host ~]$ **oc run -it my-app --image registry.access.redhat.com/ubi9/ubi \**

**--command -- /bin/bash**

If you don't see a command prompt, try pressing enter.

bash-5.1$

**Note**

Unless you include the --namespace or -n options, the run command creates containers in pods in the current selected project.

You can also define a restart policy for containers in a pod by including the --restart option. A pod restart policy determines how the cluster should respond when containers in that pod exit. The --restart option has the following accepted values: Always, OnFailure, and Never.

**Always**

If the restart policy is set to Always, then the cluster continuously tries to restart a successfully exited container, for up to five minutes. The default pod restart policy is Always. If the --restart option is omitted, then the pod is configured with the Always policy.

**OnFailure**

Setting the pod restart policy to OnFailure tells the cluster to restart only failed containers in the pod, for up to five minutes.

**Never**

If the restart policy is set to Never, then the cluster does not try to restart exited or failed containers in a pod. Instead, the pods immediately fail and exit.

The following example command executes the date command in the container of the pod named my-app, redirects the date command output to the terminal, and defines Never as the pod restart policy.

[user@host ~]$ **oc run -it my-app \**

**--image registry.access.redhat.com/ubi9/ubi \**

**--restart Never --command -- date**

Mon Feb 20 22:36:55 UTC 2023

To automatically delete a pod after it exits, include the --rm option with the run command.

[user@host ~]$ **kubectl run -it my-app --rm \**

**--image registry.access.redhat.com/ubi9/ubi \**

**--restart Never --command -- date**

Mon Feb 20 22:38:50 UTC 2023

pod "date" deleted

For some containerized applications, you might need to specify environment variables for the application to work. To specify an environment variable and its value, include the --env= option with the run command.

[user@host ~]$ **oc run mysql \**

**--image registry.redhat.io/rhel9/mysql-80 \**

**--env MYSQL\_ROOT\_PASSWORD=myP@$$123**

pod/mysql created

User and Group IDs Assignment

When a project is created, OpenShift adds annotations to the project that determine the user ID (UID) range and supplemental group ID (GID) for pods and their containers in the project. You can retrieve the annotations with the oc describe project *project-name* command.

[user@host ~]$ **oc describe project my-app**

Name: my-app

*...output omitted...*

Annotations: openshift.io/description=

openshift.io/display-name=

openshift.io/requester=developer

openshift.io/sa.scc.mcs=s0:c27,c4

**openshift.io/sa.scc.supplemental-groups=1000710000/10000**

**openshift.io/sa.scc.uid-range=1000710000/10000**

*...output omitted...*

With OpenShift default security policies, regular cluster users cannot choose the USER or UIDs for their containers. When a regular cluster user creates a pod, OpenShift ignores the USER instruction in the container image. Instead, OpenShift assigns to the user in the container a UID and a supplemental GID from the identified range in the project annotations. The GID of the user is always 0, which means that the user belongs to the root group. Any files and directories that the container processes might write to must have read and write permissions by GID=0 and have the root group as the owner. Although the user in the container belongs to the root group, the user is an unprivileged account.

In contrast, when a cluster administrator creates a pod, the USER instruction in the container image is processed. For example, if the USER instruction for the container image is set to 0, then the user in the container is the root privileged account, with a 0 value for the UID. Executing a container as a privileged account is a security risk. A privileged account in a container has unrestricted access to the container's host system. Unrestricted access means that the container could modify or delete system files, install software, or otherwise compromise its host. Red Hat therefore recommends that you run containers as rootless, or as an unprivileged user with only the necessary privileges for the container to run.

Red Hat also recommends that you run containers from different applications with unique user IDs. Running containers from different applications with the same UID, even an unprivileged one, is a security risk. If the UID for two containers is the same, then the processes in one container could access the resources and files of the other container. By assigning a distinct range of UIDs and GIDs for each project, OpenShift ensures that applications in different projects do not run as the same UID or GID.

Pod Security

The Kubernetes Pod Security Admission controller issues a warning when a pod is created without a defined security context. Security contexts grant or deny OS-level privileges to pods. OpenShift uses the Security Context Constraints controller to provide safe defaults for pod security. You can ignore pod security warnings in these course exercises. Security Context Constraints (SCC) are discussed in more detail in course DO280: *Red Hat OpenShift Administration II: Operating a Production Kubernetes Cluster*.

Execute Commands in Running Containers

To execute a command in a running container in a pod, you can use the exec command with the kubectl or oc CLI. The exec command uses the following syntax:

**oc exec *RESOURCE/NAME* -- *COMMAND* [args...] [options]**

The output of the executed command is sent to your terminal. In the following example, the exec command executes the date command in the my-app pod.

[user@host ~]$ **oc exec my-app -- date**

Tue Feb 21 20:43:53 UTC 2023

The specified command is executed in the first container of a pod. For multicontainer pods, include the -c or --container= options to specify which container is used to execute the command. The following example executes the date command in a container named ruby-container in the my-app pod.

[user@host ~]$ **kubectl exec my-app -c ruby-container -- date**

Tue Feb 21 20:46:50 UTC 2023

The exec command also accepts the -i and -t options to create an interactive session with a container in a pod. In the following example, Kubernetes sends stdin to the bash shell in the ruby-container container from the my-app pod, and sends stdout and stderr from the bash shell back to the terminal.

[user@host ~]$ **oc exec my-app -c ruby-container -it -- bash -il**

[1000780000@ruby-container /]$

In the previous example, a raw terminal is opened in the ruby-container container. From this interactive session, you can execute additional commands in the container. To terminate the interactive session, you must execute the exit command in the raw terminal.

[user@host ~]$ **kubectl exec my-app -c ruby-container -it -- bash -il**

[1000780000@ruby-container /]$ date

Tue Feb 21 21:16:00 UTC 2023

[1000780000@ruby-container] **exit**

Container Logs

Container logs are the standard output (stdout) and standard error (stderr) output of a container. You can retrieve logs with the logs pod *pod-name* command that the kubectl and oc CLIs provide. The command includes the following options:

**-l or --selector=''**

Filter objects based on the specified key:value label constraint.

**--tail=**

Specify the number of lines of recent log files to display; the default value is -1 with no selectors, which displays all log lines.

**-c or --container=**

Print the logs of a particular container in a multicontainer pod.

**-f or --follow**

Follow, or stream, logs for a container.

**-p or --previous=true**

Print the logs of a previous container instance in the pod, if it exists. This option is helpful for troubleshooting a pod that failed to start, because it prints the logs of the last attempt.

The following example restricts oc logs command output to the 10 most recent log files:

[user@host ~]$ **oc logs postgresql-1-jw89j --tail=10**

done

server stopped

Starting server...

2023-01-04 22:00:16.945 UTC [1] LOG: starting PostgreSQL 12.11 on x86\_64-redhat-linux-gnu, compiled by gcc (GCC) 8.5.0 20210514 (Red Hat 8.5.0-10), 64-bit

2023-01-04 22:00:16.946 UTC [1] LOG: listening on IPv4 address "0.0.0.0", port 5432

2023-01-04 22:00:16.946 UTC [1] LOG: listening on IPv6 address "::", port 5432

2023-01-04 22:00:16.953 UTC [1] LOG: listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"

2023-01-04 22:00:16.960 UTC [1] LOG: listening on Unix socket "/tmp/.s.PGSQL.5432"

2023-01-04 22:00:16.968 UTC [1] LOG: redirecting log output to logging collector process

2023-01-04 22:00:16.968 UTC [1] HINT: Future log output will appear in directory "log".

You can also use the attach *pod-name* -c *container-name* -it command to connect to and start an interactive session on a running container in a pod. The -c *container-name* option is required for multicontainer pods. If the container name is omitted, then Kubernetes uses the kubectl.kubernetes.io/default-container annotation on the pod to select the container. Otherwise, the first container in the pod is chosen. You can use the interactive session to retrieve application log files and to troubleshoot application issues.

[user@host ~]$ **oc attach my-app -it**

If you don't see a command prompt, try pressing enter.

bash-4.4$

You can also retrieve logs from the web console by clicking the **Logs** tab of any pod.

|  |
| --- |
| A screenshot of a computer  Description automatically generated |

If you have more than one container, then you can change between them to list the logs of each one.

|  |
| --- |
| A screenshot of a computer  Description automatically generated |

Deleting Resources

You can delete Kubernetes resources, such as pod resources, with the delete command. The delete command can delete resources by resource type and name, resource type and label, standard input (stdin), and with JSON- or YAML-formatted files. The command accepts only one argument type at a time.

For example, you can supply the resource type and name as a command argument.

[user@host ~]$ **oc delete pod php-app**

You can also delete pods from the web console by clicking **Actions** and then **Delete Pod** in the pod's principal menu.

|  |
| --- |
| A screenshot of a computer  Description automatically generated |

To select resources based on labels, you can include the -l option and the key:value label as a command argument.

[user@host ~]$ **kubectl delete pod -l app=my-app**

pod "php-app" deleted

pod "mysql-db" deleted

You can also provide the resource type and a JSON- or YAML-formatted file that specifies the name of the resource. To use a file, you must include the -f option and provide the full path to the JSON- or YAML-formatted file.

[user@host ~]$ **oc delete pod -f ~/php-app.json**

pod "php-app" deleted

You can also use stdin and a JSON- or YAML-formatted file that includes the resource type and resource name with the delete command.

[user@host ~]$ **cat ~/php-app.json | kubectl delete -f -**

pod "php-app" deleted

Pods support graceful termination, which means that pods try to terminate their processes first before Kubernetes forcibly terminates the pods. To change the time period before a pod is forcibly terminated, you can include the --grace-period flag and a time period in seconds in your delete command. For example, to change the grace period to 10 seconds, use the following command:

[user@host ~]$ **oc delete pod php-app --grace-period=10**

To shut down the pod immediately, set the grace period to 1 second. You can also use the --now flag to set the grace period to 1 second.

[user@host ~]$ **oc delete pod php-app --now**

You can also forcibly delete a pod with the --force option. If you forcibly delete a pod, Kubernetes does not wait for a confirmation that the pod's processes ended, which can leave the pod's processes running until its node detects the deletion. Therefore, forcibly deleting a pod could result in inconsistency or data loss. Forcibly delete pods only if you are sure that the pod's processes are terminated.

[user@host ~]$ **kubectl delete pod php-app --force**

To delete all pods in a project, you can include the --all option.

[user@host ~]$ **kubectl delete pods --all**

pod "php-app" deleted

pod "mysql-db" deleted

Likewise, you can delete a project and its resources with the oc delete project *project-name* command.

[user@host ~]$ **oc delete project my-app**

project.project.openshift.io "my-app" deleted

The CRI-O Container Engine

A container engine is required to run containers. Worker and control plane nodes in an OpenShift Container Platform cluster use the CRI-O container engine to run containers. Unlike tools such as Podman or Docker, the CRI-O container engine is a runtime that is designed and optimized specifically for running containers in a Kubernetes cluster. Because CRI-O meets the Kubernetes Container Runtime Interface (CRI) standards, the container engine can integrate with other Kubernetes and OpenShift tools, such as networking and storage plug-ins.

**Note**

For more information about the Kubernetes Container Runtime Interface (CRI) standards, refer to the *CRI-API* repository at <https://github.com/kubernetes/cri-api>.

CRI-O provides a command-line interface to manage containers with the crictl command. The crictl command includes several subcommands to help you to manage containers. The following subcommands are commonly used with the crictl command:

**crictl pods**

Lists all pods on a node.

**crictl image**

Lists all images on a node.

**crictl inspect**

Retrieve the status of one or more containers.

**crictl exec**

Run a command in a running container.

**crictl logs**

Retrieve the logs of a container.

**crictl ps**

List running containers on a node.

To manage containers with the crictl command, you must first identify the node that is hosting your containers.

[user@host ~]$ **kubectl get pods -o wide**

NAME READY STATUS RESTARTS AGE IP NODE

postgresql-1-8lzf2 1/1 Running 0 20m 10.8.0.64 **master01**

postgresql-1-deploy 0/1 Completed 0 21m 10.8.0.63 master01

[user@host ~]$ **oc get pod postgresql-1-8lzf2 -o jsonpath='{.spec.nodeName}{"\n"}'**

master01

Next, you must connect to the identified node as a cluster administrator. Cluster administrators can use SSH to connect to a node or create a debug pod for the node. Regular users cannot connect to or create debug pods for cluster nodes.

As a cluster administrator, you can create a debug pod for a node with the oc debug node/*node-name* command. OpenShift creates the pod/*node-name*-debug pod in your currently selected project and automatically connects you to the pod. You must then enable access host binaries, such as the crictl command, with the chroot /host command. This command mounts the host's root file system in the /host directory within the debug pod shell. By changing the root directory to the /host directory, you can run binaries contained in the host's executable path.

[user@host ~]$ **oc debug node/master01**

Starting pod/master01-debug ...

To use host binaries, run **chroot /host**

Pod IP: 192.168.50.10

If you don't see a command prompt, try pressing enter.

sh-4.4# **chroot /host**

After enabling host binaries, you can use the crictl command to manage the containers on the node. For example, you can use the crictl ps and crictl inspect commands to retrieve the process ID (PID) of a running container. You can then use the PID to retrieve or enter the namespaces within a container, which is useful for troubleshooting application issues.

To find the PID of a running container, you must first determine the container's ID. You can use the crictl ps command with the --name option to filter the command output to a specific container.

sh-5.1# **crictl ps --name postgresql**

CONTAINER IMAGE CREATED STATE NAME ATTEMPT POD ID POD

27943ae4f3024 image...7104 5...ago Running postgresql 0 5768...f015 postgresql-1...

The default output of the crictl ps command is a table. You can find the short container ID under the CONTAINER column. You can also use the -o or --output options to specify the format of the crictl ps command as JSON or YAML and then parse the output. The parsed output displays the full container ID.

sh-5.1# **crictl ps --name postgresql -o json | jq .containers[0].id**

"2794...29a4"

After identifying the container ID, you can use the crictl inspect command and the container ID to retrieve the PID of the running container. By default, the crictl inspect command displays verbose output. You can use the -o or --output options to format the command output as JSON, YAML, a table, or as a Go template. If you specify the JSON format, you can then parse the output with the jq command. Likewise, you can use the grep command to limit the command output.

sh-5.1# **crictl inspect -o json 27943ae4f3024 | jq .info.pid**

**43453**

sh-5.1# **crictl inspect 27943ae4f3024 | grep pid**

"pid": **43453**,

*...output omitted...*

After determining the PID of a running container, you can use the lsns -p *PID* command to list the system namespaces of a container.

sh-5.1# **lsns -p 43453**

NS TYPE NPROCS PID USER COMMAND

4026531835 cgroup 530 1 root /usr/lib/systemd/systemd --switched-root --system --deserialize 17

4026531837 user 530 1 root /usr/lib/systemd/systemd --switched-root --system --deserialize 17

4026537853 uts 8 43453 1000690000 postgres

4026537854 ipc 8 43453 1000690000 postgres

4026537856 net 8 43453 1000690000 postgres

4026538013 mnt 8 43453 1000690000 postgres

4026538014 pid 8 43453 1000690000 postgres

You can also use the PID of a running container with the nsenter command to enter a specific namespace of a running container. For example, you can use the nsenter command to execute a command within a specified namespace on a running container. The following example executes the ps -ef command within the process namespace of a running container.

sh-5.1# **nsenter -t 43453 -p -r ps -ef**

UID PID PPID C STIME TTY TIME CMD

1000690+ 1 0 0 18:49 ? 00:00:00 postgres

1000690+ 58 1 0 18:49 ? 00:00:00 postgres: logger

1000690+ 60 1 0 18:49 ? 00:00:00 postgres: checkpointer

1000690+ 61 1 0 18:49 ? 00:00:00 postgres: background writer

1000690+ 62 1 0 18:49 ? 00:00:00 postgres: walwriter

1000690+ 63 1 0 18:49 ? 00:00:00 postgres: autovacuum launcher

1000690+ 64 1 0 18:49 ? 00:00:00 postgres: stats collector

1000690+ 65 1 0 18:49 ? 00:00:00 postgres: logical replication launcher

root 7414 0 0 20:14 ? 00:00:00 ps -ef

The -t option specifies the PID of the running container as the target PID for the nsenter command. The -p option directs the nsenter command to enter the process or pid namespace. The -r option sets the top-level directory of the process namespace as the root directory, thus enabling commands to execute in the context of the namespace.

You can also use the -a option to execute a command in all of the container's namespaces.

sh-5.1# **nsenter -t 43453 -a ps -ef**

UID PID PPID C STIME TTY TIME CMD

1000690+ 1 0 0 18:49 ? 00:00:00 postgres

1000690+ 58 1 0 18:49 ? 00:00:00 postgres: logger

1000690+ 60 1 0 18:49 ? 00:00:00 postgres: checkpointer

1000690+ 61 1 0 18:49 ? 00:00:00 postgres: background writer

1000690+ 62 1 0 18:49 ? 00:00:00 postgres: walwriter

1000690+ 63 1 0 18:49 ? 00:00:00 postgres: autovacuum launcher

1000690+ 64 1 0 18:49 ? 00:00:00 postgres: stats collector

1000690+ 65 1 0 18:49 ? 00:00:00 postgres: logical replication launcher

root 10058 0 0 20:45 ? 00:00:00 ps -ef

Guided Exercise: Create Linux Containers and Kubernetes Pods

Run a base OS container in a pod and compare the environment inside the container with its host node.

**Outcomes**

* Create a pod with a single container, and identify the pod and its container within the container engine of an OpenShift node.
* View the logs of a running container.
* Retrieve information inside a container, such as the operating system (OS) release and running processes.
* Identify the process ID (PID) and namespaces for a container.
* Identify the User ID (UID) and supplemental group ID (GID) ranges of a project.
* Compare the namespaces of containers in one pod versus in another pod.
* Inspect a pod with multiple containers, and identify the purpose of each container.

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

This command ensures that all resources are available for this exercise.

[student@workstation ~]$ **lab start pods-containers**

**Instructions**

1. Log in to the OpenShift cluster and create the pods-containers project. Determine the UID and GID ranges for pods in the pods-containers project.
   1. Log in to the OpenShift cluster as the developer user with the oc command.
   2. [student@workstation ~]$ **oc login -u developer -p developer \**
   3. **https://api.ocp4.example.com:6443**
   4. Login successful

*...output omitted...*

* 1. Create the pods-containers project.
  2. [student@workstation ~]$ **oc new-project pods-containers**
  3. Now using project "pods-containers" on server "https://api.ocp4.example.com:6443".

*...output omitted...*

* 1. Identify the UID and GID ranges for pods in the pods-containers project.
  2. [student@workstation ~]$ **oc describe project pods-containers**
  3. Name: pods-containers
  4. Created: 28 seconds ago
  5. Labels: kubernetes.io/metadata.name=pods-containers
  6. pod-security.kubernetes.io/audit=restricted
  7. pod-security.kubernetes.io/audit-version=v1.24
  8. pod-security.kubernetes.io/warn=restricted
  9. pod-security.kubernetes.io/warn-version=v1.24
  10. Annotations: openshift.io/description=
  11. openshift.io/display-name=
  12. openshift.io/requester=developer
  13. openshift.io/sa.scc.mcs=s0:c28,c22
  14. **openshift.io/sa.scc.supplemental-groups=1000800000/10000**
  15. **openshift.io/sa.scc.uid-range=1000800000/10000**
  16. Display Name: <none>
  17. Description: <none>
  18. Status: Active
  19. Node Selector: <none>
  20. Quota: <none>

Resource limits: <none>

Your UID and GID range values might differ from the previous output.

1. As the developer user, create a pod called ubi9-user from a UBI9 base container image. The image is available in the registry.ocp4.example.com:8443/ubi9/ubi container registry. Set the restart policy to Never and start an interactive session. Configure the pod to execute the whoami and id commands to determine the UIDs, supplemental groups, and GIDs of the container user in the pod. Delete the pod afterward.

After the ubi-user pod is deleted, log in as the admin user and then re-create the ubi9-user pod. Retrieve the UIDs and GIDs of the container user. Compare the values to the values of the ubi9-user pod that the developer user created.

Afterward, delete the ubi9-user pod.

* 1. Use the oc run command to create the ubi9-user pod. Configure the pod to execute the whoami and id commands through an interactive bash shell session.
  2. [student@workstation ~]$ **oc run -it ubi9-user --restart 'Never' \**
  3. **--image registry.ocp4.example.com:8443/ubi9/ubi \**
  4. **-- /bin/bash -c "whoami && id"**
  5. 1000800000

uid=1000800000(1000800000) gid=0(root) groups=0(root),1000800000

Your values might differ from the previous output.

Notice that the user in the container has the same UID that is identified in the pods-containers project. However, the GID of the user in the container is 0, which means that the user belongs to the root group. Any files and directories that the container processes might write to must have read and write permissions by GID=0 and have the root group as the owner.

Although the user in the container belongs to the root group, a UID value over 1000 means that the user is an unprivileged account. When a regular OpenShift user, such as the developer user, creates a pod, the containers within the pod run as unprivileged accounts.

* 1. Delete the pod.
  2. [student@workstation ~]$ **oc delete pod ubi9-user**

pod "ubi9-user" deleted

* 1. Log in as the admin user with the redhatocp password.
  2. [student@workstation ~]$ **oc login -u admin -p redhatocp**
  3. Login successful.
  4. You have access to 71 projects, the list has been suppressed. You can list all projects with 'oc projects'

Using project "pods-containers".

* 1. Re-create the ubi9-user pod as the admin user. Configure the pod to execute the whoami and id commands through an interactive bash shell session. Compare the values of the UID and GID for the container user to the values of the ubi9-user pod that the developer user created.

**Note**

It is safe to ignore pod security warnings for exercises in this course. OpenShift uses the Security Context Constraints controller to provide safe defaults for pod security.

[student@workstation ~]$ **oc run -it ubi9-user --restart 'Never' \**

**--image registry.ocp4.example.com:8443/ubi9/ubi \**

**-- /bin/bash -c "whoami && id"**

Warning: would violate PodSecurity "restricted:v1.24": allowPrivilegeEscalation != false (container "ubi9-user" must set securityContext.allowPrivilegeEscalation=false), unrestricted capabilities (container "ubi9-user" must set securityContext.capabilities.drop=["ALL"]), runAsNonRoot != true (pod or container "ubi9-user" must set securityContext.runAsNonRoot=true), seccompProfile (pod or container "ubi9-user" must set securityContext.seccompProfile.type to "RuntimeDefault" or "Localhost")

root

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

Notice that the value of the UID is 0, which differs from the UID range value of the pod-containers project. The user in the container is the privileged account root user and belongs to the root group. When a cluster administrator creates a pod, the containers within the pod run as a privileged account by default.

* 1. Delete the ubi9-user pod.
  2. [student@workstation ~]$ **oc delete pod ubi9-user**

pod "ubi9-user" deleted

1. As the developer user, use the oc run command to create a ubi9-date pod from a UBI9 base container image. The image is available in the registry.ocp4.example.com:8443/ubi9/ubi container registry. Set the restart policy to Never, and configure the pod to execute the date command. Retrieve the logs of the ubi9-date pod to confirm that the date command executed. Delete the pod afterward.
   1. Log in as the developer user with the developer password.
   2. [student@workstation ~]$ **oc login -u developer -p developer**
   3. Login successful.
   4. You have one project on this server: "pods-containers"

Using project "pods-containers".

* 1. Create a pod called ubi9-date that executes the date command.
  2. [student@workstation ~]$ **oc run ubi9-date --restart 'Never' \**
  3. **--image registry.ocp4.example.com:8443/ubi9/ubi -- date**

pod/ubi9-date created

* 1. Wait a few moments for the creation of the pod. Then, retrieve the logs of the ubi9-date pod.
  2. [student@workstation ~]$ **oc logs ubi9-date**

Mon Nov 28 15:02:55 UTC 2022

* 1. Delete the ubi9-date pod.
  2. [student@workstation ~]$ **oc delete pod ubi9-date**

pod "ubi9-date" deleted

1. Use the oc run ubi9-command -it command to create a ubi9-command pod with the registry.ocp4.example.com:8443/ubi9/ubi container image. Add the /bin/bash in the oc run command to start an interactive shell. Exit the pods and view the logs for the ubi9-command pod with the oc logs command. Then, connect to the ubi9-command pod with the oc attach command, and issue the following command:

**while true; do echo $(date); sleep 2; done**

This command executes the date and sleep commands to generate output to the console every two seconds. Use the oc logs command to retrieve the logs of the ubi9 pod, and confirm that the logs display the executed date and sleep commands.

* 1. Create a pod called ubi9-command and start an interactive shell.
  2. [student@workstation ~]$ **oc run ubi9-command -it \**
  3. **--image registry.ocp4.example.com:8443/ubi9/ubi -- /bin/bash**
  4. If you don't see a command prompt, try pressing enter.

bash-5.1$

* 1. Exit the shell session.
  2. bash-5.1$ **exit**
  3. exit

Session ended, resume using 'oc attach ubi9-command -c ubi9-command -i -t' command when the pod is running

* 1. Use the oc logs command to view the logs of the ubi9-command pod.
  2. [student@workstation ~]$ **oc logs ubi9-command**

bash-5.1$ [student@workstation ~]$

The pod's command prompt is returned. The oc logs command displays the pod's current stdout and stderr output in the console. Because you disconnected from the interactive session, the pod's current stdout is the command prompt, and not the commands that you executed previously.

* 1. Use the oc attach command to connect to the ubi9-command pod again. In the shell, execute the while true; do echo $(date); sleep 2; done command to continuously generate stdout output.
  2. [student@workstation ~]$ **oc attach ubi9-command -it**

If you don't see a command prompt, try pressing enter.

bash-5.1$ **while true; do echo $(date); sleep 2; done**

Mon Nov 28 15:15:16 UTC 2022

Mon Nov 28 15:15:18 UTC 2022

Mon Nov 28 15:15:20 UTC 2022

Mon Nov 28 15:15:22 UTC 2022

*...output omitted...*

* 1. Open another terminal window and view the logs for the ubi9-command pod with the oc logs command. Limit the log output to the last 10 entries with the --tail option. Confirm that the logs display the results of the command that you executed in the container.
  2. [student@workstation ~]$ **oc logs ubi9-command --tail=10**
  3. Mon Nov 28 15:15:16 UTC 2022
  4. Mon Nov 28 15:15:18 UTC 2022
  5. Mon Nov 28 15:15:20 UTC 2022
  6. Mon Nov 28 15:15:22 UTC 2022
  7. Mon Nov 28 15:15:24 UTC 2022
  8. Mon Nov 28 15:15:26 UTC 2022
  9. Mon Nov 28 15:15:28 UTC 2022
  10. Mon Nov 28 15:15:30 UTC 2022
  11. Mon Nov 28 15:15:32 UTC 2022

Mon Nov 28 15:15:34 UTC 2022

1. Identify the name for the container in the ubi9-command pod. Identify the process ID (PID) for the container in the ubi9-command pod by using a debug pod for the pod's host node. Use the crictl command to identify the PID of the container in the ubi9-command pod. Then, retrieve the PID of the container in the debug pod.
   1. Identify the container name in the ubi9-command pod with the oc get command. Specify the JSON format for the command output. Parse the JSON output with the jq command to retrieve the value of the .status.containerStatuses[].name object.
   2. [student@workstation ~]$ **oc get pod ubi9-command -o json | \**
   3. **jq .status.containerStatuses[].name**

"ubi9-command"

The ubi9-command pod has a single container of the same name.

* 1. Find the host node for the ubi9-command pod. Start a debug pod for the host with the oc debug command.
  2. [student@workstation ~]$ **oc get pods ubi9-command -o wide**
  3. NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES

ubi9-command 1/1 Running 2 (16m ago) 27m 10.8.0.26 master01 <none> <none>

[student@workstation ~]$ **oc debug node/master01**

Error from server (Forbidden): nodes "master01" is forbidden: User "developer" cannot get resource "nodes" in API group "" at the cluster scope

The debug pod fails because the developer user does not have the required permission to debug a host node.

* 1. Log in as the admin user with the redhatocp password. Start a debug pod for the host with the oc debug command. After connecting to the debug pod, run the chroot /host command to use host binaries, such as the crictl command-line tool.
  2. [student@workstation ~]$ **oc login -u admin -p redhatocp**
  3. Login successful.

*...output omitted...*

[student@workstation ~]$ **oc debug node/master01**

Starting pod/master01-debug ...

To use host binaries, run `chroot /host`

Pod IP: 192.168.50.10

If you don't see a command prompt, try pressing enter

sh-4.4# **chroot /host**

* 1. Use the crictl ps command to retrieve the ubi9-command container ID. Specify the ubi9-command container with the --name option and use the JSON output format. Parse the JSON output with the jq -r command to get the RAW JSON output. Export the container ID as the $CID environment variable.

**Note**

When using jq without the -r flag, the container ID is wrapped in double quotes, which does not work with crictl commands. If the -r flag is not used, then you can add | tr -d '"' to the end of the command to trim the double quotes.

sh-5.1# **crictl ps --name ubi9-command -o json | jq -r .containers[0].id**

81adbc6222d79ed9ba195af4e9d36309c18bb71bc04b2e8b5612be632220e0d6

sh-5.1# **CID=$(crictl ps --name ubi9-command -o json | jq -r .containers[0].id)**

sh-5.1# **echo $CID**

81adbc6222d79ed9ba195af4e9d36309c18bb71bc04b2e8b5612be632220e0d6

Your container ID value might differ from the previous output.

* 1. Use the crictl inspect command to find the PID of the ubi9-command container. The PID value is in the .info.pid object in the crictl inspect output. Export the ubi9-command container PID as the $PID environment variable.
  2. sh-5.1# **crictl inspect $CID | grep pid**
  3. "pid": 365297,
  4. "pids": {
  5. "type": "pid"
  6. *...output omitted...*
  7. }

*...output omitted...*

sh-5.1# **PID=*365297***

Your PID values might differ from the previous output.

1. Use the lsns command to list the system namespaces of the ubi9-command container. Confirm that the running processes in the container are isolated to different system namespaces.
   1. View the system namespaces of the ubi9-command container with the lsns command. Specify the PID with the -p option and use the $PID environment variable. In the resulting table, the NS column contains the namespace values for the container.
   2. sh-5.1# **lsns -p $PID**
   3. NS TYPE NPROCS PID USER COMMAND
   4. 4026531835 cgroup 540 1 root /usr/lib/systemd/systemd --switched-root --system --deserialize 16
   5. 4026531837 user 540 1 root /usr/lib/systemd/systemd --switched-root --system --deserialize 16
   6. 4026536117 uts 1 153168 1000800000 /bin/bash
   7. 4026536118 ipc 1 153168 1000800000 /bin/bash
   8. 4026536120 net 1 153168 1000800000 /bin/bash
   9. 4026537680 mnt 1 153168 1000800000 /bin/bash

4026537823 pid 1 153168 1000800000 /bin/bash

Your namespace values might differ from the previous output.

1. Use the host debug pod to retrieve and compare the operating system (OS) and the GNU C Library (glibc) package version of the ubi9-command container and the host node.
   1. Retrieve the OS for the host node with the cat /etc/redhat-release command.
   2. sh-5.1# **cat /etc/redhat-release**

Red Hat Enterprise Linux CoreOS release 4.14

* 1. Use the crictl exec command and the $CID container ID variable to retrieve the OS of the ubi9-command container. Use the -it options to create an interactive terminal to execute the cat /etc/redhat-release command.
  2. sh-5.1# **crictl exec -it $CID cat /etc/redhat-release**

Red Hat Enterprise Linux release 9.1 (Plow)

The ubi9-command container has a different OS from the host node.

* 1. Use the ldd --version command to retrieve the glibc package version of the host node.
  2. sh-5.1$ **ldd --version**
  3. ldd (GNU libc) 2.28
  4. Copyright (C) 2018 Free Software Foundation, Inc.

*...output omitted...*

* 1. Use the crictl exec command and the $CID container ID variable to retrieve the glibc package version of the ubi9-command container. Use the -it options to create an interactive terminal to execute the ldd --version command.
  2. sh-5.1# **crictl exec -it $CID ldd --version**
  3. ldd (GNU libc) 2.34
  4. Copyright (C) 2021 Free Software Foundation, Inc.

*...output omitted...*

The ubi9-command container has a different version of the glibc package from its host.

1. Exit the master01-debug pod and the ubi9-command pod.
   1. Exit the master01-debug pod. You must issue the exit command to end the host binary access. Execute the exit command again to exit and remove the master01-debug pod.
   2. sh-5.1# **exit**

exit

sh-4.4# **exit**

exit

Removing debug pod ...

Temporary namespace openshift-debug-bg7kn was removed.

* 1. Return to the terminal window that is connected to the ubi9-command pod. Press **Ctrl**+**C** and then execute the exit command. Confirm that the pod is still running.
  2. *...output omitted...*
  3. **^C**
  4. bash-5.1$ **exit**
  5. exit

Session ended, resume using 'oc attach ubi9-command -c ubi9-command -i -t' command when the pod is running

[student@workstation ~]$ **oc get pods**

NAME READY STATUS RESTARTS AGE

ubi9-command 1/1 Running 2 (6s ago) 35m

**Finish**

On the workstation machine, use the lab command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

[student@workstation ~]$ **lab finish pods-containers**

Find and Inspect Container Images

Objectives

* Find containerized applications in container registries and get information about the runtime parameters of supported and community container images.

Container Image Overview

A container is an isolated runtime environment where applications are executed as isolated processes. The isolation of the runtime environment ensures that applications do not interfere with other containers or system processes.

A container image contains a packaged version of your application, with all the necessary dependencies for the application to run. Images can exist without containers. However, containers depend on images, because containers use container images to build a runtime environment to execute applications.

Containers can be split into two similar but distinct concepts: *container images* and *container instances*. A *container image* contains immutable data that defines an application and its libraries. You can use container images to create *container instances*, which are running processes that are isolated by a set of kernel namespaces.

You can use each container image many times to create many distinct container instances. These replicas can be split across multiple hosts. The application within a container is independent of the host environment.

Container Image Registries

Image registries are services that offer container images to download. Image creators and maintainers can store and distribute container images in a controlled manner to public or private audiences. Some examples of image registries include Quay.io, Red Hat Registry, Docker Hub, and Amazon ECR.

Red Hat Registry

Red Hat distributes container images by using two registries: registry.access.redhat.com (where no authentication is required), and registry.redhat.io (where authentication is required). The Red Hat Ecosystem Catalog, <https://catalog.redhat.com/>, provides centralized searching utility for both registries. You can search the Red Hat Ecosystem Catalog for technical details about container images. The catalog hosts a large set of container images, including from major open source projects, such as Apache, MySQL, and Jenkins.

Because the Red Hat Ecosystem Catalog is also searched for software products other than container images, you must navigate to <https://catalog.redhat.com/software/containers/explore> to specifically search for container images.

|  |
| --- |
| A screenshot of a computer  Description automatically generated |

Figure 3.4: Red Hat Ecosystem Catalog

The details page of a container image gives relevant information, such as technical data, the installed packages within the image, or a security scan. You can navigate through these options by using the tabs on the website. You can also change the image version by selecting a specific tag.

|  |
| --- |
| A screenshot of a computer  Description automatically generated |

Figure 3.5: The Red Hat Universal Base Image 9

The Red Hat internal security team vets all images in the container catalog. Red Hat rebuilds all components to avoid known security vulnerabilities.

Red Hat container images provide the following benefits:

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

**Note**

You must log in to the registry.redhat.io registry with a customer portal account or a Red Hat Developer account to use the stored container images in the registry.

Quay.io

Although the Red Hat Registry stores only images from Red Hat and certified providers, you can store your own images with Quay.io, another public image registry that Red Hat sponsors. Although storing public images in Quay is free of charge, some options are available only for paying customers. Quay also offers an on-premise version of the product, which you can use to set up an image registry in your own servers.

Quay.io introduces features such as server-side image building, fine-grained access controls, and automatic scanning of images for known vulnerabilities.

Quay.io offers live images that creators regularly update. Quay.io users can create their namespaces, with fine-grained access control, and publish their created images to that namespace. Container Catalog users rarely or never push new images, but consume trusted images from the Red Hat team.

|  |
| --- |
| A screenshot of a website  Description automatically generated |

Figure 3.6: The Quay.io welcome page

Private Registries

Image creators or maintainers might want to make their images publicly available. However, other image creators might prefer to keep their images private, for the following reasons:

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

In some cases, private images are preferred. Private registries give image creators control over image placement, distribution, and usage. Private images are more secure than images in public registries.

Public Registries

Other public registries, such as Docker Hub and Amazon ECR, are also available for storing, sharing, and consuming container images. These registries can include official images that the registry owners or the registry community users create and maintain. For example, Docker Hub hosts a Docker Official Image of a WordPress container image. Although the docker.io/library/wordpress container image is a Docker Official Image, the container image is not supported by WordPress, Docker, or Red Hat. Instead, the Docker Community, a global group of Docker Hub users, supports and maintains the container image. Support for this container image depends on the availability and skills of the Docker Community users.

Consuming container images from public registries brings risks. For example, a container image might include malicious code or vulnerabilities, which can compromise the host system that executes the container image. A host system can also be compromised by public container images, because the images are often configured with the privileged root user. Additionally, the software in a container image might not be correctly licensed, or might violate licensing terms.

Before you use a container image from a public registry, review and verify the container image. Also ensure that you have the correct permissions to use the software in the container image.

Container Image Identifiers

Several objects provide identifying information about a container image.

**Registry**

It is a content server, such as registry.access.redhat.com, that is used to store and share container images. A registry consists of one or more repositories that contain tagged container images.

**Name**

It identifies the container image repository; it is a string that is composed of letters, numbers, and some special characters. This component refers to the name of the directory, or the container repository, within the container registry where the container image is.

For example, consider the fully qualified domain name (FQDN) of the registry.access.redhat.com/ubi9/httpd-24:1-233 container image. The container image is in the ubi9/httpd-24 repository in the registry.access.redhat.com container registry.

**ID/Hash**

It is the SHA (Secure Hash Algorithm) code to pull or verify an image. The SHA image ID cannot change, and always references the same container image content. The ID/hash is the true, unique identifier of an image. For example, the sha256:4186a1ead13fc30796f951694c494e7630b82c320b81e20c020b3b07c888985b image ID always refers to the registry.access.redhat.com/ubi9/httpd-24:1-233 container image.

**Tag**

It is a label for a container image in a repository, to distinguish from other images, for version control. The tag comes after the image repository name and is delimited by a colon (:).

When an image tag is omitted, the floating tag, latest, is used as the default tag. A floating tag is an alias to another tag. In contrast, a fixed tag points to a specific container build. For the registry.access.redhat.com/ubi9/httpd-24:1-233.1669634588 container image, 1-233.1669634588 is the fixed tag for the image, and at the time of writing, corresponds to the floating latest flag.

Container Image Components

A container image is composed of multiple components.

**Layers**

Container images are created from instructions. Each instruction adds a layer to the container image. Each layer consists of the differences between it and the following layer. The layers are then stacked to create a read-only container image.

**Metadata**

Metadata includes the instructions and documentation for a container image.

Container Image Instructions and Metadata

Container image layers consist of instructions, or steps, and metadata for building the image. You can override instructions during container creation to adjust the container image according to your needs. Some instructions can affect the running container, and other instructions are for informational purposes only.

The following instructions affect the state of a running container:

**ENV**

Defines the available environment variables in the container. A container image might include multiple ENV instructions. Any container can recognize additional environment variables that are not listed in its metadata.

**ARG**

It defines build-time variables, typically to make a customizable container build. Developers commonly configure the ENV instructions by using the ARG instruction. It is useful for preserving the build-time variables for run time.

**USER**

Defines the active user in the container. Later instructions run as this user. It is a good practice to define a user other than root for security purposes. OpenShift does not honor the user in a container image, for regular cluster users. Only cluster administrators can run containers (pods) with their chosen *user ID (UIDs)* and *group IDs (GIDs)*.

**ENTRYPOINT**

It defines the executable to run when the container is started.

**CMD**

It defines the command to execute when the container is started. This command is passed to the executable that the ENTRYPOINT instruction defines. Base images define a default ENTRYPOINT executable, which is usually a shell executable, such as Bash.

**WORKDIR**

It sets the current working directory within the container. Later instructions execute within this directory.

Metadata is used for documentation purposes, and does not affect the state of a running container. You can also override the metadata values during container creation.

The following metadata is for information only, and does not affect the state of the running container:

**EXPOSE**

It indicates the network port that the application binds to within the container. This metadata does not automatically bind the port on the host, and is used only for documentation purposes.

**VOLUME**

It defines where to store data outside the container. The value shows the path where your container runtime mounts the directory inside the container. More than one path can be defined to create multiple volumes.

**LABEL**

Adds a key-value pair to the metadata of the image for organization and image selection.

Container engines are not required to honor metadata in a container image, such as USER or EXPOSE. A container engine can also recognize additional environment variables that are not listed in the container image metadata.

Base Images

A *base image* is the image that your resulting container image is built on. Your chosen base image determines the Linux distribution, and any of the following components:

* Package manager
* Init system
* File system layout
* Preinstalled dependencies and runtimes

The base image can also influence factors such as image size, vendor support, and processor compatibility.

Red Hat provides enterprise-grade container images that are engineered to be the base operating system layer for your containerized applications. These container images are intended as a common starting point for containers, and are known as *universal base images* (UBI). Red Hat UBI container images are *Open Container Initiative (OCI)* compliant images that contain portions of Red Hat Enterprise Linux (RHEL). UBI container images include a subset of RHEL content. They provide a set of prebuilt runtime languages, such as Python and Node.js, and associated DNF repositories that you can use to add application dependencies. UBI-based images can be distributed without cost or restriction. They can be deployed to both Red Hat and non-Red Hat platforms, and be pushed to your chosen container registry.

A Red Hat subscription is not required to use or distribute UBI-based images. However, Red Hat provides full support only for containers that are built on UBI if the containers are deployed to a Red Hat platform, such as a Red Hat OpenShift Container Platform (RHOCP) cluster or RHEL.

Red Hat provides four UBI variants: standard, init, minimal, and micro. All UBI variants and UBI-based images use Red Hat Enterprise Linux (RHEL) at their core and are available from the Red Hat Container Catalog. The main differences are as follows:

**Standard**

This image is the primary UBI, which includes DNF, systemd, and utilities such as gzip and tar.

**Init**

This image simplifies running multiple applications within a single container by managing them with systemd.

**Minimal**

This image is smaller than the init image and provides nice-to-have features. This image uses the microdnf minimal package manager instead of the full-sized version of DNF.

**Micro**

This image is the smallest available UBI, and includes only the minimum packages. For example, this image does not include a package manager.

Inspecting and Managing Container Images

Various tools can inspect and manage container images, including the oc image command and Skopeo.

Skopeo

Skopeo is another tool to inspect and manage remote container images. With Skopeo, you can copy and sync container images from different container registries and repositories. You can also copy an image from a remote repository and save it to a local disk. If you have the appropriate repository permissions, then you can also delete an image from container registry. You also can use Skopeo to inspect the configuration and contents of a container image, and to list the available tags for a container image. Unlike other container image tools, Skopeo can execute without a privileged account, such as root. Skopeo does not require a running daemon to execute various operations.

Skopeo is executed with the skopeo command-line utility, which you can install with various package managers, such as DNF, Brew, and APT. The skopeo utility might already be installed on some Linux-based distributions. You can install the skopeo utility on Fedora, CentOS Stream 8 and later, and Red Hat Enterprise Linux 8 and later systems by using the DNF package manager.

[user@host ~]$ **sudo dnf -y install skopeo**

The skopeo utility is currently not available as a packaged binary for Windows-based systems. However, the skopeo utility is available as a container image from the quay.io/skopeo/stable container repository. For more information about the Skopeo container image, refer to the skopeoimage overview guide in the Skopeo repository (<https://github.com/containers/skopeo/blob/main/contrib/skopeoimage/README.md>).

You can also build skopeo from source code in a container, or build it locally without using a container. Refer to the installation guide in the Skopeo repository (<https://github.com/containers/skopeo/blob/main/install.md#container-images>) for more information about installing or building Skopeo from source code.

The skopeo utility provides commands to help you to manage and inspect container images and container image registries. For container registries that require authentication, you must first log in to the registry before you can execute additional skopeo commands.

[user@host ~]$ **skopeo login quay.io**

**Note**

OpenShift clusters are typically configured with registry credentials. When a pod is created from a container image in a remote repository, OpenShift authenticates to the container registry with the configured registry credentials, and then pulls, or copies, the image. Because OpenShift automatically uses the registry credentials, you typically do not need to manually authenticate to a container registry when you create a pod. By contrast, the oc image command and the skopeo utility require you first to log in to a container registry.

After you log in to a container registry (if required), you can execute additional skopeo commands against container images in a repository. When you execute a skopeo command, you must specify the transport and the repository name. A *transport* is the mechanism to transfer or move container images between locations. Two common transports are docker and dir. The docker transport is used for container registries, and the dir transport is used for local directories.

The oc image command and other tools default to the docker transport, and so you do not need to specify the transport when executing commands. However, the skopeo utility does not define a default transport; you must specify the transport with the container image name. Most skopeo commands use the skopeo *command* *[command options]* *transport*://*IMAGE-NAME* format. For example, the following skopeo list-tags command lists all available tags in a registry.access.redhat.com/ubi9/httpd-24 container repository by using the docker transport:

[user@host ~]$ **skopeo list-tags docker://registry.access.redhat.com/ubi9/httpd-24**

{

"Repository": "registry.access.redhat.com/ubi9/httpd-24",

"Tags": [

"1-229",

"1-217.1666632462",

"1-201",

"1-194.1655192191-source",

"1-229-source",

"1-194-source",

"1-201-source",

"1-217.1664813224",

"1-217.1664813224-source",

"1-210-source",

"1-233.1669634588",

"1",

"1-217.1666632462-source",

"1-233",

"1-194.1655192191",

"1-217.1665076049-source",

"1-217",

"1-233.1669634588-source",

"1-210",

"1-217.1665076049",

"1-217-source",

"1-233-source",

"1-194",

"latest"

]

}

The skopeo utility includes other useful commands for container image management.

**skopeo inspect**

View low-level information for an image name, such as environment variables and available tags. Use the skopeo inspect *[command options]* *transport*://*IMAGE-NAME* command format. You can include the --config flag to view the configuration, metadata, and history of a container repository. The following example retrieves the configuration information for the registry.access.redhat.com/ubi9/httpd-24 container repository:

[user@host ~]$ **skopeo inspect --config docker://registry.access.redhat.com/ubi9/httpd-24**

*...output omitted...*

"config": {

"User": "1001",

"ExposedPorts": {

"8080/tcp": {},

"8443/tcp": {}

},

"Env": [

*...output omitted...*

"HTTPD\_MAIN\_CONF\_PATH=/etc/httpd/conf",

"HTTPD\_MAIN\_CONF\_MODULES\_D\_PATH=/etc/httpd/conf.modules.d",

"HTTPD\_MAIN\_CONF\_D\_PATH=/etc/httpd/conf.d",

"HTTPD\_TLS\_CERT\_PATH=/etc/httpd/tls",

"HTTPD\_VAR\_RUN=/var/run/httpd",

"HTTPD\_DATA\_PATH=/var/www",

"HTTPD\_DATA\_ORIG\_PATH=/var/www",

"HTTPD\_LOG\_PATH=/var/log/httpd"

],

"Entrypoint": [

"container-entrypoint"

],

"Cmd": [

"/usr/bin/run-httpd"

],

"WorkingDir": "/opt/app-root/src",

*...output omitted...*

}

*...output omitted...*

**skopeo copy**

Copy an image from one location or repository to another. Use the skopeo copy *transport*://*SOURCE-IMAGE* *transport*://*DESTINATION-IMAGE* format. For example, the following command copies the quay.io/skopeo/stable:latest container image to the skopeo repository in the registry.example.com container registry:

[user@host ~]$ **skopeo copy docker://quay.io/skopeo/stable:latest \**

**docker://registry.example.com/skopeo:latest**

**skopeo delete**

Delete a container image from a repository. You must use the skopeo delete *[command options]* *transport*://*IMAGE-NAME* format. The following command deletes the skopeo:latest image from the registry.example.com container registry:

[user@host ~]$ **skopeo delete docker://registry.example.com/skopeo:latest**

**skopeo sync**

Synchronize one or more images from one location to another. Use this command to copy all container images from a source to a destination. The command uses the skopeo sync *[command options]* --src *transport* --dest *transport* *SOURCE* *DESTINATION* format. The following command synchronizes the registry.access.redhat.com/ubi8/httpd-24 container repository to the registry.example.com/httpd-24 container repository:

[user@host ~]$ **skopeo sync --src docker --dest docker \**

**registry.access.redhat.com/ubi8/httpd-24 registry.example.com/httpd-24**

Registry Credentials

Some registries require users to authenticate. For example, Red Hat containers that are based on RHEL typically require authenticated access:

[user@host ~]$ **skopeo inspect docker://registry.redhat.io/rhel8/httpd-24**

FATA[0000] Error parsing image name "docker://registry.redhat.io/rhel8/httpd-24": **unable to retrieve auth token: invalid username/password: unauthorized: Please login to the Red Hat Registry using your Customer Portal credentials.** Further instructions can be found here: https://access.redhat.com/RegistryAuthentication

You might choose a different image that does not require authentication, such as the UBI 8 image:

[user@host ~]$ **skopeo inspect docker://registry.access.redhat.com/ubi8:latest**

{

"Name": "registry.access.redhat.com/ubi8",

"Digest": "sha256:70fc...1173",

"RepoTags": [

"8.7-1054-source",

"8.6-990-source",

"8.6-754",

"8.4-203.1622660121-source",

*...output omitted...*.

Alternatively, you must execute the skopeo login command for the registry before you can access the RHEL 8 image.

[user@host ~]$ **skopeo login registry.redhat.io**

Username: **YOUR\_USER**

Password: **YOUR\_PASSWORD**

Login Succeeded!

[user@host ~]$ **skopeo list-tags docker://registry.redhat.io/rhel8/httpd-24**

{

"Repository": "registry.redhat.io/rhel8/httpd-24",

"Tags": [

"1-166.1645816922",

"1-209",

"1-160-source",

"1-112",

*...output omitted...*

Skopeo stores the credentials in the ${XDG\_RUNTIME\_DIR}/containers/auth.json file, where the ${XDG\_RUNTIME\_DIR} refers to a directory that is specific to the current user. The credentials are encoded in the base64 format:

[user@host ~]$ **cat ${XDG\_RUNTIME\_DIR}/containers/auth.json**

{

"auths": {

"registry.redhat.io": {

"auth": **"dXNlcjpodW50ZXIy"**

}

}

}

[user@host ~]$ **echo -n dXNlcjpodW50ZXIy | base64 -d**

user:hunter2

**Note**

For security reasons, the skopeo login command does not show your password in the interactive session. Although you do not see what you are typing, Skopeo registers every key stroke. After typing your full password in the interactive session, press **Enter** to start the login.

The oc image Command

The OpenShift command-line interface provides the oc image command. You can use this command to inspect, configure, and retrieve information about container images.

The oc image info command inspects and retrieves information about a container image. You can use the oc image info command to identify the ID/hash SHA and to list the image layers of a container image. You can also review container image metadata, such as environment variables, network ports, and commands. If a container image repository provides a container image in multiple architectures, such as amd64 or arm64, then you must include the --filter-by-os tag. For example, you can execute the following command to retrieve information about the registry.access.redhat.com/ubi9/httpd-24:1-233 container image that is based on the amd64 architecture:

[user@host ~]$ **oc image info registry.access.redhat.com/ubi9/httpd-24:1-233 \**

**--filter-by-os amd64**

Name: registry.access.redhat.com/ubi9/httpd-24:1-233

Digest: sha256:4186...985b

*...output omitted...*

Image Size: 130.8MB in 3 layers

Layers: 79.12MB sha256:d74e...1cad

17.32MB sha256:dac0...a283

34.39MB sha256:47d8...5550

OS: linux

Arch: amd64

Entrypoint: container-entrypoint

Command: /usr/bin/run-httpd

Working Dir: /opt/app-root/src

User: 1001

Exposes Ports: 8080/tcp, 8443/tcp

Environment: container=oci

*...output omitted...*

HTTPD\_CONTAINER\_SCRIPTS\_PATH=/usr/share/container-scripts/httpd/

HTTPD\_APP\_ROOT=/opt/app-root

HTTPD\_CONFIGURATION\_PATH=/opt/app-root/etc/httpd.d

HTTPD\_MAIN\_CONF\_PATH=/etc/httpd/conf

HTTPD\_MAIN\_CONF\_MODULES\_D\_PATH=/etc/httpd/conf.modules.d

HTTPD\_MAIN\_CONF\_D\_PATH=/etc/httpd/conf.d

HTTPD\_TLS\_CERT\_PATH=/etc/httpd/tls

HTTPD\_VAR\_RUN=/var/run/httpd

HTTPD\_DATA\_PATH=/var/www

HTTPD\_DATA\_ORIG\_PATH=/var/www

HTTPD\_LOG\_PATH=/var/log/httpd

*...output omitted...*

The oc image command provides more options to manage container images.

**oc image append**

Use this command to add layers to container images, and then push the container image to a registry.

**oc image extract**

You can use this command to extract or copy files from a container image to a local disk. Use this command to access the contents of a container image without first running the image as a container. A running container engine is not required.

**oc image mirror**

Copy or mirror container images from one container registry or repository to another. For example, you can use this command to mirror container images between public and private registries. You can also use this command to copy a container image from a registry to a disk. The command mirrors the HTTP structure of a container registry to a directory on a disk. The directory on the disk can then be served as a container registry.

Running Containers as Root

Running containers as the root user is a security risk, because an attacker could exploit the application, access the container, and exploit further vulnerabilities to escape from the containerized environment into the host system. Attackers might escape the containerized environment by exploiting bugs and vulnerabilities that are typically in the kernel or container runtime.

Traditionally, when an attacker gains access to the container file system by using an exploit, the root user inside the container corresponds to the root user outside the container. If an attacker escapes the container isolation, then they have elevated privileges on the host system, which potentially causes more damage.

Containers that do not run as the root user have limitations that might prove unsuitable for use in your application, such as the following limitations:

**Non-trivial Containerization**

Some applications might require the root user. Depending on the application architecture, some applications might not be suitable for non-root containers, or might require a deeper understanding to containerize.

For example, applications such as HTTPd and Nginx start a bootstrap process and then create a process with a non-privileged user, which interacts with external users. Such applications are non-trivial to containerize for rootless use.

Red Hat provides containerized versions of HTTPd and Nginx that do not require root privileges for production usage. You can find the containers in the Red Hat container registry (https://catalog.redhat.com/software/containers/explore).

**Required Use of Privileged Utilities**

Non-root containers cannot bind to privileged ports, such as the 80 or 443 ports. Red Hat advises against using privileged ports, but to use port forwarding instead.

Similarly, non-root containers cannot use the ping utility by default, because it requires elevated privileges to establish raw sockets.

Guided Exercise: Find and Inspect Container Images

Use a supported MySQL container image to run server and client pods in Kubernetes; also test two community images and compare their runtime requirements.

**Outcomes**

* Locate and run container images from a container registry.
* Inspect remote container images and container logs.
* Set environment variables and override entry points for a container.
* Access files and directories within a container.

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

This command ensures that all resources are available for this exercise.

[student@workstation ~]$ **lab start pods-images**

**Instructions**

1. Log in to the OpenShift cluster and create the pods-images project.
   1. Log in to the OpenShift cluster as the developer user with the oc command.
   2. [student@workstation ~]$ **oc login -u developer -p developer \**
   3. **https://api.ocp4.example.com:6443**

*...output omitted...*

* 1. Create the pods-images project.
  2. [student@workstation ~]$ **oc new-project pods-images**

*...output omitted...*

1. Authenticate to registry.ocp4.example.com:8443, which is the classroom container registry. This private registry hosts certain copies and tags of community images from Docker and Bitnami, as well as some supported images from Red Hat. Use skopeo to log in as the developer user, and then retrieve a list of available tags for the registry.ocp4.example.com:8443/redhattraining/docker-nginx container repository.
   1. Use the skopeo login command to log in as the developer user with the developer password.
   2. [student@workstation ~]$ **skopeo login registry.ocp4.example.com:8443**
   3. Username: **developer**
   4. Password: **developer**

Login Succeeded!

* 1. The classroom registry contains a copy and specific tags of the docker.io/library/nginx container repository. Use the skopeo list-tags command to retrieve a list of available tags for the registry.ocp4.example.com:8443/redhattraining/docker-nginx container repository.
  2. [student@workstation ~]$ **skopeo list-tags \**
  3. **docker://registry.ocp4.example.com:8443/redhattraining/docker-nginx**
  4. {
  5. "Repository": "registry.ocp4.example.com:8443/redhattraining/docker-nginx",
  6. "Tags": [
  7. "1.23",
  8. "1.23-alpine",
  9. "1.23-perl",
  10. "1.23-alpine-perl"
  11. "latest"
  12. ]

}

1. Create a docker-nginx pod from the registry.ocp4.example.com:8443/redhattraining/docker-nginx:1.23 container image. Investigate any pod failures.
   1. Use the oc run command to create the docker-nginx pod.
   2. [student@workstation ~]$ **oc run docker-nginx \**
   3. **--image registry.ocp4.example.com:8443/redhattraining/docker-nginx:1.23**

pod/docker-nginx created

* 1. After a few moments, verify the status of the docker-nginx pod.
  2. [student@workstation ~]$ **oc get pods**
  3. NAME READY STATUS RESTARTS AGE

docker-nginx 0/1 Error 0 4s

[student@workstation ~]$ **oc get pods**

NAME READY STATUS RESTARTS AGE

docker-nginx 0/1 CrashLoopBackOff 2 (17s ago) 38s

The docker-nginx pod failed to start.

* 1. Investigate the pod failure. Retrieve the logs of the docker-nginx pod to identify a possible cause of the pod failure.
  2. [student@workstation ~]$ **oc logs docker-nginx**
  3. *...output omitted...*
  4. /docker-entrypoint.sh: Configuration complete; ready for start up
  5. 2022/12/02 18:51:45 [warn] 1#1: the "user" directive makes sense only if the master process runs with super-user privileges, ignored in /etc/nginx/nginx.conf:2
  6. nginx: [warn] the "user" directive makes sense only if the master process runs with super-user privileges, ignored in /etc/nginx/nginx.conf:2
  7. 2022/12/02 18:51:45 [emerg] 1#1: **mkdir() "/var/cache/nginx/client\_temp" failed (13: Permission denied)**

**nginx: [emerg] mkdir() "/var/cache/nginx/client\_temp" failed (13: Permission denied)**

The pod failed to start because of permission issues for the nginx directories.

* 1. Create a debug pod for the docker-nginx pod.
  2. [student@workstation ~]$ **oc debug pod/docker-nginx**
  3. Starting pod/docker-nginx-debug ...
  4. Pod IP: 10.8.0.72
  5. If you don't see a command prompt, try pressing enter.

$

* 1. From the debug pod, verify the permissions of the /etc/nginx and /var/cache/nginx directories.
  2. $ **ls -la /etc/ | grep nginx**

drwxr-xr-x. 3 root root 132 Nov 15 13:14 nginx

$ **ls -la /var/cache | grep nginx**

drwxr-xr-x. 2 root root 6 Oct 19 09:32 nginx

Only the root user has permission to the nginx directories. The pod must therefore run as the privileged root user to work.

* 1. Retrieve the user ID (UID) of the docker-nginx user to determine whether the user is a privileged or unprivileged account. Then, exit the debug pod.
  2. $ **whoami**

1000820000

$ **exit**

Removing debug pod ...

Your UID value might differ from the previous output.

A UID over 0 means that the container's user is a non-root account. Recall that OpenShift default security policies prevent regular user accounts, such as the developer user, from running pods and their containers as privileged accounts.

* 1. Confirm that the docker-nginx:1.23 image requires the root privileged account. Use the skopeo inspect --config command to view the configuration for the image.
  2. [student@workstation ~]$ **skopeo inspect --config \**
  3. **docker://registry.ocp4.example.com:8443/redhattraining/docker-nginx:1.23**
  4. *...output omitted...*
  5. "config": {
  6. "ExposedPorts": {
  7. "80/tcp": {}
  8. },
  9. "Env": [
  10. "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
  11. "NGINX\_VERSION=1.23.3",
  12. "NJS\_VERSION=0.7.9",
  13. "PKG\_RELEASE=1~bullseye"
  14. ],
  15. "Entrypoint": [
  16. "/docker-entrypoint.sh"
  17. ],
  18. "Cmd": [
  19. "nginx",
  20. "-g",
  21. "daemon off;"
  22. ],
  23. "Labels": {
  24. "maintainer": "NGINX Docker Maintainers \u003cdocker-maint@nginx.com\u003e"
  25. },
  26. "StopSignal": "SIGQUIT"
  27. },

*...output omitted...*

The image configuration does not define USER metadata, which confirms that the image must run as the root privileged user.

* 1. The docker-nginx:1-23 container image must run as the root privileged user. OpenShift security policies prevent regular cluster users, such as the developer user, from running containers as the root user. Delete the docker-nginix pod.
  2. [student@workstation ~]$ **oc delete pod docker-nginx**

pod "docker-nginx" deleted

1. Create a bitnami-mysql pod, which uses a copy of the Bitnami community MySQL image. The image is available in the registry.ocp4.example.com:8443/redhattraining/bitnami-mysql container repository.
   1. A copy and specific tags of the docker.io/bitnami/mysql container repository are hosted in the classroom registry. Use the skopeo list-tags command to identify available tags for the Bitnami MySQL community image in the registry.ocp4.example.com:8443/redhattraining/bitnami-mysql container repository.
   2. [student@workstation ~]$ **skopeo list-tags \**
   3. **docker://registry.ocp4.example.com:8443/redhattraining/bitnami-mysql**
   4. {
   5. "Repository": "registry.ocp4.example.com:8443/redhattraining/bitnami-mysql",
   6. "Tags": [
   7. "8.0.31",
   8. "8.0.30",
   9. "8.0.29",
   10. "8.0.28",
   11. "latest"
   12. ]

}

* 1. Retrieve the configuration of the bitnami-mysql:8.0.31 container image. Determine whether the image requires a privileged account by inspecting image configuration for USER metadata.
  2. [student@workstation ~]$ **skopeo inspect --config \**
  3. **docker://registry.ocp4.example.com:8443/redhattraining/bitnami-mysql:8.0.31**
  4. *...output omitted...*
  5. "config":
  6. "User": **"1001"**,
  7. "ExposedPorts": {
  8. "3306/tcp": {}
  9. },

*....output omitted...*

The image defines the 1001 UID, which means that the image does not require a privileged account.

* 1. Create the bitnami-mysql pod with the oc run command. Use the registry.ocp4.example.com:8443/redhattraining/bitnami-mysql:8.0.31 container image. Then, wait a few moments and then retrieve the pod's status with the oc get command.
  2. [student@workstation ~]$ **oc run bitnami-mysql \**
  3. **--image registry.ocp4.example.com:8443/redhattraining/bitnami-mysql:8.0.31**

pod/bitnami-mysql created

[student@workstation ~]$ **oc get pods**

NAME READY STATUS RESTARTS AGE

bitnami-mysql 0/1 CrashLoopBackoff 2 (19s ago) 23s

The pod failed to start.

* 1. Examine the logs of the bitnami-mysql pod to determine the cause of the failure.
  2. [student@workstation ~]$ **oc logs bitnami-mysql**
  3. mysql 16:18:00.40
  4. mysql 16:18:00.40 Welcome to the Bitnami mysql container
  5. mysql 16:18:00.40 Subscribe to project updates by watching https://github.com/bitnami/containers
  6. mysql 16:18:00.40 Submit issues and feature requests at https://github.com/bitnami/containers/issues
  7. mysql 16:18:00.40
  8. mysql 16:18:00.41 INFO ==> \*\* Starting MySQL setup \*\*
  9. mysql 16:18:00.42 INFO ==> Validating settings in MYSQL\_\*/MARIADB\_\* env vars

mysql 16:18:00.42 ERROR ==> **The MYSQL\_ROOT\_PASSWORD environment variable is empty or not set.** Set the environment variable ALLOW\_EMPTY\_PASSWORD=yes to allow the container to be started with blank passwords. This is recommended only for development.

The MYSQL\_ROOT\_PASSWORD environment variable must be set for the pod to start.

* 1. Delete and then re-create the bitnami-mysql pod. Specify redhat123 as the value for the MYSQL\_ROOT\_PASSWORD environment variable. After a few moments, verify the status of the pod.
  2. [student@workstation ~]$ **oc delete pod bitnami-mysql**

pod "bitnami-mysql" deleted

[student@workstation ~]$ **oc run bitnami-mysql \**

**--image registry.ocp4.example.com:8443/redhattraining/bitnami-mysql:8.0.31 \**

**--env MYSQL\_ROOT\_PASSWORD=redhat123**

pod/bitnami-mysql created

[student@workstation ~]$ **oc get pods**

NAME READY STATUS RESTARTS AGE

bitnami-mysql 1/1 Running 0 20s

The bitnami-mysql pod successfully started.

* 1. Determine the UID of the container user in the bitnami-mysql pod. Compare this value to the UID in the container image and to the UID range of the pods-images project.
  2. [student@workstation ~]$ **oc exec -it bitnami-mysql -- /bin/bash -c "whoami && id"**
  3. 1000820000

uid=1000820000(1000820000) gid=0(root) groups=0(root),1000820000

[student@workstation ~]$ **oc describe project pods-images**

Name: pods-images

*...output omitted...*

Annotations: openshift.io/description=

*...output omitted...*

openshift.io/sa.scc.supplemental-groups=1000820000/10000

**openshift.io/sa.scc.uid-range=1000820000/10000**

*...output omitted...*

Your values for the UID of the container and the UID range of the project might differ from the previous output.

The container user UID is the same as the specified UID range in the namespace. Notice that the container user UID does not match the 1001 UID of the container image. For a container to use the specified UID of a container image, the pod must be created with a privileged OpenShift user account, such as the admin user.

1. The private classroom registry hosts a copy of a supported MySQL image from Red Hat. Retrieve the list of available tags for the registry.ocp4.example.com:8443/rhel9/mysql-80 container repository. Compare the rhel9/mysql-80 container image release version that is associated with each tag.
   1. Use the skopeo list-tags command to list the available tags for the rhel9/mysql-80 container image.
   2. [student@workstation ~]$ **skopeo list-tags \**
   3. **docker://registry.ocp4.example.com:8443/rhel9/mysql-80**
   4. {
   5. "Repository": "registry.ocp4.example.com:8443/rhel9/mysql-80",
   6. "Tags": [
   7. "1-237",
   8. "1-228",
   9. "1-228-source",
   10. "1-224",
   11. "1-224-source",
   12. "latest",
   13. "1"
   14. ]

}

Several tags are available:

* + - The latest and 1 tags are floating tags, which are aliases to other tags, such as the 1-237 tag.
    - The 1-228 and 1-224 tags are fixed tags, which point to a build of a container.
    - The 1-228-source and 1-224-source tags are source containers, which provide the necessary sources and license terms to rebuild and distribute the images.
  1. Use the skopeo inspect command to compare the rhel9/mysql-80 container image release version and SHA IDs that are associated with the identified tags.

**Note**

To improve readability, the instructions truncate the SHA-256 strings.

On your system, the commands return the full SHA-256 strings.

[student@workstation ~]$ **skopeo inspect \**

**docker://registry.ocp4.example.com:8443/rhel9/mysql-80:latest**

*...output omitted...*

"Name": "registry.ocp4.example.com:8443/rhel9/mysql-80",

"Digest": **"sha256:d282...f38f",**

*...output omitted...*

"Labels":

*...output omitted...*

"name": "rhel9/mysql-80",

"release": **"237",**

*...output omitted...*

You can also format the output of the skopeo inspect command with a Go template. Append the template objects with \n to add new lines between the results.

[student@workstation ~]$ **skopeo inspect --format \**

**"Name: {{.Name}}\n Digest: {{.Digest}}\n Release: {{.Labels.release}}" \**

**docker://registry.ocp4.example.com:8443/rhel9/mysql-80:latest**

Name: registry.ocp4.example.com:8443/rhel9/mysql-80

Digest: **sha256:d282...f38f**

Release: **237**

[student@workstation ~]$ **skopeo inspect --format \**

**"Name: {{.Name}}\n Digest: {{.Digest}}\n Release: {{.Labels.release}}" \**

**docker://registry.ocp4.example.com:8443/rhel9/mysql-80:1**

Name: registry.ocp4.example.com:8443/rhel9/mysql-80

Digest: **sha256:d282...f38f**

Release: **237**

[student@workstation ~]$ **skopeo inspect --format \**

**"Name: {{.Name}}\n Digest: {{.Digest}}\n Release: {{.Labels.release}}" \**

**docker://registry.ocp4.example.com:8443/rhel9/mysql-80:1-237**

Name: registry.ocp4.example.com:8443/rhel9/mysql-80

Digest: **sha256:d282...f38f**

Release: **237**

The latest, 1, and 1-237 tags resolve to the same release versions and SHA IDs. The latest and 1 tags are floating tags for the 1-237 fixed tag.

1. The classroom registry hosts a copy and certain tags of the registry.redhat.io/rhel9/mysql-80 container repository. Use the oc run command to create a rhel9-mysql pod from the registry.ocp4.example.com:8443/rhel9/mysql-80:1-228 container image. Verify the status of the pod and then inspect the container logs for any errors.
   1. Create a rhel9-mysql pod with the registry.ocp4.example.com:8443/rhel9/mysql-80:1-237 container image.
   2. [student@workstation ~]$ **oc run rhel9-mysql \**
   3. **--image registry.ocp4.example.com:8443/rhel9/mysql-80:1-237**

pod/rhel9-mysql created

* 1. After a few moments, retrieve the pod's status with the oc get command.
  2. [student@workstation ~]$ **oc get pods**
  3. NAME READY STATUS RESTARTS AGE
  4. bitnami-mysql 1/1 Running 0 5m16s

rhel9-mysql 0/1 CrashLoopBackoff 2 (29s ago) 49s

The pod failed to start.

* 1. Retrieve the logs for the rhel9-mysql pod to determine why the pod failed.
  2. [student@workstation ~]$ **oc logs rhel9-mysql**
  3. => sourcing 20-validate-variables.sh ...
  4. **You must either specify the following environment variables:**
  5. MYSQL\_USER (regex: '^[a-zA-Z0-9\_]+$')
  6. MYSQL\_PASSWORD (regex: '^[a-zA-Z0-9\_~!@#$%^&\*()-=<>,.?;:|]+$')
  7. MYSQL\_DATABASE (regex: '^[a-zA-Z0-9\_]+$')
  8. Or the following environment variable:
  9. MYSQL\_ROOT\_PASSWORD (regex: '^[a-zA-Z0-9\_~!@\#$%^&\*()-=<>,.?;:|]+$')
  10. Or both.
  11. Optional Settings:
  12. MYSQL\_LOWER\_CASE\_TABLE\_NAMES (default: 0)

*...output omitted...*

The pod failed because the required environment variables were not set for the container.

1. Delete the rhel9-mysql pod. Create another rhel9-mysql pod and specify the necessary environment variables. Retrieve the status of the pod and inspect the container logs to confirm that the new pod is working.
   1. Delete the rhel9-mysql pod with the oc delete command. Wait for the pod to delete before continuing to the next step.
   2. [student@workstation ~]$ **oc delete pod rhel9-mysql**

pod "rhel9-mysql" deleted

* 1. Create another rhel9-mysql pod from the registry.ocp4.example.com:8443/rhel9/mysql-80:1-237 container image. Use the oc run command with the --env option to specify the following environment variables and their values:

| **Variable** | **Value** |
| --- | --- |
| MYSQL\_USER | redhat |
| MYSQL\_PASSWORD | redhat123 |
| MYSQL\_DATABASE | worldx |

* 1. [student@workstation ~]$ **oc run rhel9-mysql \**
  2. **--image registry.ocp4.example.com:8443/rhel9/mysql-80:1-237 \**
  3. **--env MYSQL\_USER=redhat \**
  4. **--env MYSQL\_PASSWORD=redhat123 \**
  5. **--env MYSQL\_DATABASE=worldx**
  6. pod/rhel9-mysql created
  7. After a few moments, retrieve the status of the rhel9-mysql pod with the oc get command. View the container logs to confirm that the database on the rhel9-mysql pod is ready to accept connections.
  8. [student@workstation ~]$ **oc get pods**
  9. NAME READY STATUS RESTARTS AGE
  10. bitnami-mysql 1/1 Running 0 10m

rhel9-mysql 1/1 Running 0 20s

[student@workstation ~]$ **oc logs rhel9-mysql**

*...output omitted...*

2022-11-02T20:14:14.333599Z 0 [System] [MY-011323] [Server] X Plugin ready for connections. Bind-address: '::' port: 33060, socket: /var/lib/mysql/mysqlx.sock

2022-11-02T20:14:14.333641Z 0 [System] [MY-010931] [Server] /usr/libexec/mysqld: **ready for connections.** Version: '8.0.30' socket: '/var/lib/mysql/mysql.sock' port: 3306 Source distribution.

The rhel9-mysql pod is ready to accept connections.

1. Determine the location of the MySQL database files for the rhel9-mysql pod. Confirm that the directory contains the worldx database.
   1. Use the oc image command to inspect the rhel9/mysql-80:1-228 image in the registry.ocp4.example.com:8443 classroom registry.
   2. [student@workstation ~]$ **oc image info \**
   3. **registry.ocp4.example.com:8443/rhel9/mysql-80:1-237**
   4. Name: registry.ocp4.example.com:8443/rhel9/mysql-80:1-237
   5. *...output omitted...*
   6. Command: run-mysqld
   7. Working Dir: /opt/app-root/src
   8. User: 27
   9. Exposes Ports: 3306/tcp
   10. Environment: container=oci
   11. STI\_SCRIPTS\_URL=image:///usr/libexec/s2i
   12. STI\_SCRIPTS\_PATH=/usr/libexec/s2i
   13. APP\_ROOT=/opt/app-root
   14. PATH=/opt/app-root/src/bin:/opt/app-root/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
   15. PLATFORM=el9
   16. MYSQL\_VERSION=8.0
   17. APP\_DATA=/opt/app-root/src

HOME=/var/lib/mysql

The container manifest sets the HOME environment variable for the container user to the /var/lib/mysql directory.

* 1. Use the oc exec command to list the contents of the /var/lib/mysql directory.
  2. [student@workstation ~]$ **oc exec -it rhel9-mysql -- ls -la /var/lib/mysql**
  3. total 12
  4. drwxrwxr-x. 1 mysql root 102 Nov 2 20:41 .
  5. drwxr-xr-x. 1 root root 19 Oct 24 18:47 ..
  6. drwxrwxr-x. 1 mysql root 4096 Nov 2 20:54 data
  7. srwxrwxrwx. 1 mysql 1000820000 0 Nov 2 20:41 mysql.sock
  8. -rw-------. 1 mysql 1000820000 2 Nov 2 20:41 mysql.sock.lock
  9. srwxrwxrwx. 1 mysql 1000820000 0 Nov 2 20:41 mysqlx.sock

-rw-------. 1 mysql 1000820000 2 Nov 2 20:41 mysqlx.sock.lock

A data directory exists in the /var/lib/mysql directory.

* 1. Use the oc exec command again to list the contents of the /var/lib/mysql/data directory.
  2. [student@workstation ~]$ **oc exec -it rhel9-mysql \**
  3. **-- ls -la /var/lib/mysql/data | grep worldx**

drwxr-x---. 2 1000820000 root 6 Nov 2 20:41 **worldx**

The /var/lib/mysql/data directory contains the worldx database with the worldx directory.

1. Determine the IP address of the rhel9-mysql pod. Next, create another MySQL pod, named mysqlclient, to access the rhel9-mysql pod. Confirm that the mysqlclient pod can view the available databases on the rhel9-mysql pod with the mysqlshow command.
   1. Identify the IP address of the rhel9-mysql pod.
   2. [student@workstation ~]$ **oc get pods rhel9-mysql -o json | jq .status.podIP**

"10.8.0.109"

Note the IP address. Your IP address might differ from the previous output.

* 1. Use the oc run command to create a pod named mysqlclient that uses the registry.ocp4.example.com:8443/rhel9/mysql-80:1-237 container image. Set the value of the MYSQL\_ROOT\_PASSWORD environment variable to redhat123, and then confirm that the pod is running.
  2. [student@workstation ~]$ **oc run mysqlclient \**
  3. **--image registry.ocp4.example.com:8443/rhel9/mysql-80:1-237 \**
  4. **--env MYSQL\_ROOT\_PASSWORD=redhat123**

pod/mysqlclient created

[student@workstation ~]$ **oc get pods**

NAME READY STATUS RESTARTS AGE

bitnami-mysql 1/1 Running 0 15m

mysqlclient 1/1 Running 0 19s

rhel9-mysql 1/1 Running 0 5m

* 1. Use the oc exec command with the -it options to execute the mysqlshow command on the mysqlclient pod. Connect as the redhat user and specify the host as the IP address of the rhel9-mysql pod. When prompted, enter redhat123 for the password.
  2. [student@workstation ~]$ **oc exec -it mysqlclient \**
  3. **-- mysqlshow -u redhat -p -h *10.8.0.109***
  4. Enter password: **redhat123**
  5. +--------------------+
  6. | Databases |
  7. +--------------------+
  8. | information\_schema |
  9. | performance\_schema |
  10. | worldx |

+--------------------+

The worldx database on the rhel9-mysql pod is accessible to the mysql-client pod.

**Finish**

On the workstation machine, use the lab command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

[student@workstation ~]$ **lab finish pods-images**

Troubleshoot Containers and Pods

Objectives

* Troubleshoot a pod by starting additional processes on its containers, changing their ephemeral file systems, and opening short-lived network tunnels.

Container Troubleshooting Overview

Containers are designed to be immutable and ephemeral. A running container must be redeployed when changes are needed or when a new container image is available. However, you can change a running container without redeployment.

Updating a running container is best reserved for troubleshooting problematic containers. Red Hat does not generally recommend editing a running container to fix errors in a deployment. Changes to a running container are not captured in source control, but help to identify the needed corrections to the source code for the container functions. Capture these container updates in version control after you identify the necessary changes. Then, build a new container image and redeploy the application.

Custom alterations to a running container are incompatible with elegant architecture, reliability, and resilience for the environment.

CLI Troubleshooting Tools

Administrators use various tools to interact with, inspect, and alter running containers. Administrators can use commands such as oc get to gather initial details for a specified resource type. Other commands are available for detailed inspection of a resource, or to update a resource in real time.

**Note**

When interacting with the cluster containers, take suitable precautions with actively running components, services, and applications.

Use these tools to validate the functions and environment for a running container:

* The kubectl CLI provides the following commands:
  + kubectl describe: Display the details of a resource.
  + kubectl edit: Edit a resource configuration by using the system editor.
  + kubectl patch: Update a specific attribute or field for a resource.
  + kubectl replace: Deploy a new instance of the resource.
  + kubectl cp: Copy files and directories to and from containers.
  + kubectl exec: Execute a command within a specified container.
  + kubectl explain: Display documentation for a specified resource.
  + kubectl port-forward: Configure a port forwarder for a specified container.
  + kubectl logs: Retrieve the logs for a specified container.

Besides supporting the previous kubectl commands, the oc CLI adds the following commands for inspecting and troubleshooting running containers:

* The oc CLI provides the following commands:
  + oc status: Display the status of the containers in the selected namespace.
  + oc rsync: Synchronize files and directories to and from containers.
  + oc rsh: Start a remote shell within a specified container.

Editing Resources

Troubleshooting and remediation often begin with a phase of inspection and data gathering. When solving issues, the describe command can provide helpful details about the running resource, such as the definition of a container and its purpose.

The following example demonstrates use of the oc describe *RESOURCE* *NAME* command to retrieve information about a pod in the openshift-dns namespace:

[user@host ~]$ **oc describe pod dns-default-lt13h**

Name: dns-default-lt13h

Namespace: openshift-dns

Priority: 2000001000

Priority Class Name: system-node-critical

*...output omitted...*

Various CLI tools can apply a change that you determine is needed to a running container. The edit command opens the specified resource in the default editor for your environment. This editor is specified by setting either the KUBE\_EDITOR or the EDITOR environment variable, or otherwise with the vi editor in Linux or the Notepad application in Windows.

The following example demonstrates use of the oc edit *RESOURCE* *NAME* command to edit a running container:

[user@host ~]$ **oc edit pod mongo-app-sw88b**

# Please edit the object below. Lines beginning with a '#' will be ignored,

# and an empty file will abort the edit. If an error occurs while saving this file will be

# reopened with the relevant failures.

#

apiVersion: v1

kind: Pod

metadata:

annotations:

*...output omitted...*

You can also use the patch command to update fields of a resource.

The following example uses the patch command to update the container image that a pod uses:

[user@host ~]$ **oc patch pod valid-pod --type='json' \**

**-p='[{"op": "replace", "path": "/spec/containers/0/image", \**

**"value":"http://registry.access.redhat.com/ubi8/httpd-24"}]'**

**Note**

For more information about patching resources and the different merge methods, refer to [Update API Objects in Place Using kubectl patch](https://kubernetes.io/docs/tasks/manage-kubernetes-objects/update-api-object-kubectl-patch/).

Copy Files to and from Containers

Administrators can copy files and directories to or from a container to inspect, update, or correct functionality. Adding a configuration file or retrieving an application log are common use cases.

**Note**

To use the cp command with the kubectl CLI or the oc CLI, the tar binary must be present in the container. If the binary is absent, then an error message appears and the operation fails.

The following example demonstrates copying a file from a running container to a local directory by using the oc cp *SOURCE* *DEST* command:

[user@host ~]$ **oc cp apache-app-kc82c:/var/www/html/index.html /tmp/index.bak**

[user@host ~]$ **ls /tmp**

index.bak

The following example demonstrates use of the oc cp *SOURCE* *DEST* command to copy a file from a local directory to a directory in a running container:

[user@host ~]$ **oc cp /tmp/index.html apache-app-kc82c:/var/www/html/**

[user@host ~]$ **oc exec -it apache-app-kc82c -- ls /var/www/html**

index.bak

**Note**

Targeting a file path within a pod for either the *SOURCE* or *DEST* argument uses the *pod\_name:path* format, and can include the -c *container\_name* option to specify a container within the pod. If you omit the -c *container\_name* option, then the command targets the first container in the pod.

Additionally, when using the oc CLI, file and directory synchronization is available by using the oc rsync command.

The following example demonstrates use of the oc rsync *SOURCE\_NAME* *DEST* command to synchronize files from a running container to a local directory.

[user@host ~]$ **oc rsync apache-app-kc82c:/var/www/ /tmp/web\_files**

[user@host ~]$ **ls /tmp/web\_files**

cgi-bin

html

The oc rsync command uses the rsync client on your local system to copy changed files to and from a pod container. The rsync binary must be available locally and within the container for this approach. If the rsync binary is not found, then a tar archive is created on the local system and is sent to the container. The container then uses the tar utility to extract files from the archive. Without the rsync and tar binaries, an error message occurs and the oc rsync command fails.

**Note**

For Linux-based systems, you can install the rsync client and the tar utility on a local system by using a package manager, such as DNF. For Windows-based systems, you can install the cwRsync client. For more information about the cwRysnc client, refer to <https://www.itefix.net/cwrsync>.

Remote Container Access

Exposing a network port for a container is routine, especially for containers that provide a service. In a cluster, port forwarding connections are made through the kubelet, which maps a local port on your system to a port on a pod. Configuring port forwarding creates a request through the Kubernetes API, and creates a multiplexed stream, such as HTTP/2, with a port header that specifies the target port in the pod. The kubelet delivers the stream data to the target pod and port, and vice versa for egress data from the pod.

When troubleshooting an application that typically runs without a need to connect locally, you can use the port-forwarding function to expose connectivity to the pod for investigation. With this function, an administrator can connect on the new port and inspect the problematic application. After you remediate the issue, the application can be redeployed without the port-forward connection.

The following example demonstrates use of the oc port-forward *RESOURCE* *EXTERNAL\_PORT:CONTAINER\_PORT* command to listen locally on port 8080 and to forward connections to port 80 on the pod:

[user@host ~]$ **oc port-forward nginx-app-cc78k 8080:80**

Forwarding from 127.0.0.1:8080 -> 80

Forwarding from [::1]:8080 -> 80

Connect to Running Containers

Administrators use CLI tools to connect to a container via a shell for forensic inspections. With this approach, you can connect to, inspect, and run any available commands within the specified container.

The following example demonstrates use of the oc rsh *POD\_NAME* command to connect to a container via a shell:

[user@host ~]$ **oc rsh tomcat-app-jw53r**

sh-4.4#

The oc rsh command does not accept the -n *namespace* option. Therefore, you must change to the namespace of the pod before you execute the oc rsh command. If you need to connect to a specific container in a pod, then use the -c *container\_name* option to specify the container name. If you omit this option, then the command connects to the first container in the pod.

You can also connect to running containers from the web console by clicking the **Terminal** tab in the pod's principal menu.

|  |
| --- |
| A screenshot of a computer  Description automatically generated |

If you have more than one container, then you can change between them to connect to the CLI.

|  |
| --- |
| A screenshot of a computer  Description automatically generated |

Execute Commands in a Container

Passing commands to execute within a container from the CLI is another method for troubleshooting a running container. Use this method to send a command to run within the container, or to connect to the container, when further investigation is necessary.

Use the following command to pass and execute commands in a container:

**oc exec *POD | TYPE/NAME* [-c *container\_name*] -- *COMMAND* [*arg1* ... *argN*]**

If you omit the -c *container\_name* option, then the command targets the first container in the pod.

The following examples demonstrate the use of the oc exec command to execute the ls command in a container to list the contents of the container's root directory:

[user@host ~]$ **oc exec -it mariadb-lc78h -- ls /**

bin boot dev etc help.1 home lib lib64 ...

*...output omitted...*

[user@host ~]$ **oc exec mariadb-lc78h -- ls /**

bin

boot

dev

etc

*...output omitted...*

**Note**

It is common to add the -it flags to the kubectl exec or oc exec commands. These flags instruct the command to send STDIN to the container and STDOUT/STDERR back to the terminal. The format of the command output is impacted by the inclusion of the -it flags.

Container Events and Logs

Reviewing the historical actions for a container can offer insights into both the lifecycle and health of the deployment. Retrieving the cluster logs provides the chronological details of the container actions. Administrators inspect this log output for information and issues that occur in the running container.

For the following commands, use the -c *container\_name* to specify a container in the pod. If you omit this option, then the command targets the first container in the pod.

The following examples demonstrate use of the oc logs *POD\_NAME* command to retrieve the logs for a pod:

[user@host ~]$ **oc logs BIND9-app-rw43j**

Defaulted container "dns" out of: dns

.:5353

[INFO] plugin/reload: Running configuration SHA512 = 7c3d...3587

CoreDNS-1.9.2

*...output omitted...*

In Kubernetes, an event resource is a report of an event somewhere in the cluster. You can use the kubectl get events and oc get events commands to view pod events in a namespace:

[user@host ~]$ **oc get events**

LAST SEEN TYPE REASON OBJECT MESSAGE

*...output omitted...*

21m Normal AddedInterface pod/php-app-5d9b84b588-kzfxd Add eth0 [10.8.0.93/23] from ovn-kubernetes

21m Normal Pulled pod/php-app-5d9b84b588-kzfxd Container image "registry.ocp4.example.com:8443/redhattraining/php-webapp:v4" already present on machine

21m Normal Created pod/php-app-5d9b84b588-kzfxd Created container php-webapp

21m Normal Started pod/php-app-5d9b84b588-kzfxd Started container php-webapp

Available Linux Commands in Containers

The use of Linux commands for troubleshooting applications can also help with troubleshooting containers. However, when connecting to a container, only the defined tools and applications within the container are available. You can augment the environment inside the container by adding the tools from this section or any other remedial tools to the container image.

Before you add tools to a container image, consider how the tools affect your container image.

* Additional tools increase the size of the image, which might impact container performance.
* Tools might require additional update packages and licensing terms, which can impact the ease of updating and distributing the container image.
* Hackers might exploit tools in the image.

Troubleshooting from Inside the Cluster

It is routine to troubleshoot a cluster, its components, or the running applications by connecting remotely. This approach assumes that the administrator's computer contains the necessary tools for the work. When unanticipated issues arise, the necessary tools might not be available from an administrator's computer or an alternative machine.

Administrators can alternatively author and deploy a container within the cluster for investigation and remediation. By creating a container image that includes the cluster troubleshooting tools, you have a reliable environment to perform these tasks from any computer with access to the cluster. This approach ensures that an administrator always has access to the tools for reliable troubleshooting and remediation of issues.

Additionally, administrators should plan to author a container image that provides the most valuable troubleshooting tools for containerized applications. In this way, you deploy this "toolbox" container to supplement the forensic process and to provide an environment with the required commands and tools for troubleshooting problematic containers. For example, the "toolbox" container can test how resources operate inside a cluster, such as to confirm whether a pod can connect to resources outside the cluster. Regular cluster users can also create a "toolbox" container to help with application troubleshooting. For example, a regular user could run a pod with a MySQL client to connect to another pod that runs a MySQL server.

Although this approach falls outside the focus of this course, because it is more application-level remediation than container-level troubleshooting, it is important to realize that containers have such capacity.

Guided Exercise: Troubleshoot Containers and Pods

Troubleshoot and fix a failed MySQL pod and manually initialize a database with test data.

**Outcomes**

* Investigate errors with creating a pod.
* View the status, logs, and events for a pod.
* Copy files into a running pod.
* Connect to a running pod by using port forwarding.

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

This command ensures that the cluster and all exercise resources are available.

[student@workstation ~]$ **lab start pods-troubleshooting**

**Instructions**

1. Log in to the OpenShift cluster and create the pods-troubleshooting project.
   1. Log in to the OpenShift cluster as the developer user with the oc command.
   2. [student@workstation ~]$ **oc login -u developer -p developer \**
   3. **https://api.ocp4.example.com:6443**

*...output omitted...*

* 1. Create the pods-troubleshooting project.
  2. [student@workstation ~]$ **oc new-project pods-troubleshooting**

*...output omitted...*

1. Create a MySQL pod called mysql-server with the oc run command. Use the registry.ocp4.example.com:8443/rhel9/mysql-80:1228 container image for the pod. Specify the environment variables with the following values:

| **Variable** | **Value** |
| --- | --- |
| MYSQL\_USER | redhat |
| MYSQL\_PASSWORD | redhat123 |
| MYSQL\_DATABASE | world |

1. Then, view the status of the pod with the oc get command.
   1. Create the mysql-server pod with the oc run command. Specify the environment values with the --env option.
   2. [student@workstation ~]$ **oc run mysql-server \**
   3. **--image registry.ocp4.example.com:8443/rhel9/mysql-80:1228 \**
   4. **--env MYSQL\_USER=redhat \**
   5. **--env MYSQL\_PASSWORD=redhat123 \**
   6. **--env MYSQL\_DATABASE=world**

pod/mysql created

* 1. After a few moments, retrieve the status of the pod. Execute the oc get pods command a few times to view the status updates for the pod.
  2. [student@workstation ~]$ **oc get pods**
  3. NAME READY STATUS RESTARTS AGE

mysql-server 0/1 ErrImagePull 0 30s

[studet@workstation ~]$ **oc get pods**

NAME READY STATUS RESTARTS AGE

mysql-server 0/1 ImagePullBackoff 0 45s

The pod failed to start.

1. Identify the root cause of the pod's failure.
   1. Retrieve the pod's logs with the oc logs command.
   2. [student@workstation ~]$ **oc logs mysql-server**

Error from server (BadRequest): container "mysql-server" in pod "mysql-server" is waiting to start: trying and failing to pull image

The logs state that the pod cannot pull the container image.

* 1. Retrieve the events log with the oc get events command.
  2. student@workstation ~]$ **oc get events**
  3. LAST SEEN TYPE REASON OBJECT MESSAGE
  4. 33s Normal Scheduled pod/mysql-server Successfully assigned pods-troubleshooting/mysql-server to master01 by master01
  5. 31s Normal AddedInterface pod/mysql-server Add eth0 [10.8.0.68/23] from ovn-kubernetes
  6. 16s Normal Pulling pod/mysql-server Pulling image "registry.ocp4.example.com:8443/rhel9/mysql-80:1228"
  7. 16s Warning Failed pod/mysql-server **Failed to pull image "registry.ocp4.example.com:8443/rhel9/mysql-80:1228": rpc error: code = Unknown desc = reading manifest 1228 in registry.ocp4.example.com:8443/rhel9/mysql-80: manifest unknown: manifest unknown**
  8. 16s Warning Failed pod/mysql-server Error: ErrImagePull
  9. 4s Normal BackOff pod/mysql-server Back-off pulling image "registry.ocp4.example.com:8443/rhel9/mysql-80:1228"

4s Warning Failed pod/mysql-server Error: ImagePullBackOff

The output states that the image pull failed because the 1228 manifest is unknown. This failure could mean that the manifest, or image tag, does not exist in the repository.

* 1. Inspect the available manifest in the registry.ocp4.example.com:8443/rhel9/mysql-80 container repository. Authenticate to the container repository with the skopeo login command as the developer user with the developer password. Then, use the skopeo inspect command to retrieve the available manifests in the registry.ocp4.example.com:8443/rhel9/mysql-80 repository.
  2. [student@workstation ~]$ **skopeo login registry.ocp4.example.com:8443**
  3. Username: **developer**
  4. Password: **developer**

Login Succeeded!

[student@workstation ~]$ **skopeo inspect \**

**docker://registry.ocp4.example.com:8443/rhel9/mysql-80**

*...output omitted...*

"Name": "registry.ocp4.example.com:8443/rhel9/mysql-80",

"Digest": "sha256:d282...f38f",

"RepoTags": [

"1-237",

"1-228",

"1-228-source",

"1-224",

"1-224-source",

"latest",

"1"

],

*...output omitted...*

The 1228 manifest, or tag, is not available in the repository, which means that the registry.ocp4.example.com:8443/rhel9/mysql-80:1228 image does not exist. However, the 1-228 tag does exist.

1. The pod failed to start because of a typing error in the image tag. Update the pod's configuration to use the registry.ocp4.example.com:8443/rhel9/mysql-80:1-228 container image. Confirm that the pod is re-created after editing the resource.
   1. Edit the pod's configuration with the oc edit command. Locate the .spec.containers.image object. Update the value to the registry.ocp4.example.com:8443/rhel9/mysql-80:1-228 container image and save the change.

[student@workstation ~]$ **oc edit pod/mysql-server**

*...output omitted...*

apiVersion: v1

kind: Pod

metadata:

*...output omitted...*

spec:

containers:

- image: registry.ocp4.example.com:8443/rhel9/**mysql-80:1-228**

*...output omitted...*

* 1. Verify the status of the mysql-server pod with the oc get command. The pod's status might take a few moments to update after the resource edit. Repeat the oc get command until the pod's status changes.
  2. [student@workstation ~]$ **oc get pods**
  3. NAME READY STATUS RESTARTS AGE

mysql-server 1/1 Running 0 10m

The mysql-server pod successfully pulled the image and created the container. The pod now shows a Running status.

1. Prepare the database on the mysql-server pod. Copy the ~/DO180/labs/pods-troubleshooting/world\_x.sql file to the mysql-server pod. Then, connect to the pod and execute the world\_x.sql file to populate the world database.
   1. Use the oc cp command to copy the world\_x.sql file in the ~/DO180/labs/pods-troubleshooting directory to the /tmp/ directory on the mysql-server pod.
   2. [student@workstation ~]$ **oc cp ~/DO180/labs/pods-troubleshooting/world\_x.sql \**

**mysql-server:/tmp/**

* 1. Confirm that the world\_x.sql file is accessible within the mysql-server pod with the oc exec command.
  2. [student@workstation ~]$ **oc exec -it pod/mysql-server -- ls -la /tmp/**
  3. total 0
  4. drwxrwxrwx. 1 root root 22 Nov 4 14:45 .
  5. dr-xr-xr-x. 1 root root 50 Nov 4 14:34 ..

-rw-rw-r--. 1 1000680000 root 558791 Nov 4 14:45 world\_x.sql

* 1. Connect to the mysql-server pod with the oc rsh command. Then, log in to MySQL as the redhat user with the redhat123 password.

[student@workstation ~]$ **oc rsh mysql-server**

sh-5.1$ **mysql -u redhat -p**

Enter password: ***redhat123***

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

*...output omitted...*

mysql>

* 1. From the MySQL prompt, select the world database. Source the world\_x.sql script inside the pod to initialize and populate the world database.
  2. mysql> **USE world;**

Database changed

mysql> **SOURCE /tmp/world\_x.sql;**

*...output omitted...*

* 1. Execute the SHOW TABLES; command to confirm that the database now contains tables. Then, exit the database and the pod.
  2. mysql> **SHOW TABLES;**
  3. -----------------
  4. | Tables\_in\_test |
  5. -----------------
  6. | city |
  7. | country |
  8. | countryinfo |
  9. | countrylanguage |
  10. -----------------

4 rows in set (0.00 sec)

mysql> **exit;**

Bye

sh-5.1$ **exit**

exit

[student@workstation ~]$

1. Configure port forwarding and then use the MySQL client on the workstation machine to connect to the world database on the mysql-server pod. Confirm that you can access data within the world database from the workstation machine.
   1. From the workstation machine, use the oc port-forward command to forward the 3306 local port to the 3306 port on the mysql-server pod.
   2. [student@workstation ~]$ **oc port-forward mysql-server 3306:3306**
   3. Forwarding from 127.0.0.1:3306 -> 3306

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

* 1. Open another terminal window on the workstation machine. Connect to the world database with the local MySQL client on the workstation machine. Log in as the redhat user with the redhat123 password. Specify the host as the localhost 127.0.0.1 IP address and use 3306 as the port.
  2. [student@workstation ~]$ **mysql -u redhat -p -h 127.0.0.1 -P 3306**
  3. Enter password: **redhat123**
  4. Welcome to the MySQL monitor. Commands end with ; or \g.
  5. *...output omitted...*

mysql>

* 1. Select the world database and execute the SHOW TABLES; command.
  2. mysql> **USE world;**

Database changed

mysql> **SHOW TABLES;**

-----------------

| Tables\_in\_test |

-----------------

| city |

| country |

| countryinfo |

| countrylanguage |

-----------------

4 rows in set (0.01 sec)

* 1. Confirm that you can retrieve data from the country table. Execute the SELECT COUNT(\*) FROM country command to retrieve the number of countries within the country table.
  2. mysql> **SELECT COUNT(\*) FROM country;**
  3. ----------
  4. | COUNT(\*) |
  5. ----------
  6. | 239 |
  7. ----------

1 row in set (0.01 sec)

* 1. Exit the database.
  2. mysql> **exit;**
  3. Bye

[student@workstation ~]$

* 1. Return to the terminal that is executing the oc port-forward command. Press **Ctrl**+**C** to end the connection.
  2. [student@workstation ~]$ oc port-forward mysql-server 3306:3306
  3. Forwarding from 127.0.0.1:3306 -> 3306
  4. Forwarding from [::1]:3306 -> 3306
  5. Handling connection for 3306
  6. Handling connection for 3306

**^C**[student@workstation ~]$

**Finish**

On the workstation machine, use the lab command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

[student@workstation ~]$ **lab finish pods-troubleshooting**

[Previous](https://rha.ole.redhat.com/rha/app/courses/do180-4.14/pages/ch03s05/c2358540-87d5-48de-b49e-6f23bdcd629c)[Next](https://rha.ole.redhat.com/rha/app/courses/do180-4.14/pages/ch03s07/c2358540-87d5-48de-b49e-6f23bdcd629c)

Lab: Run Applications as Containers and Pods

Run a web server as a pod and insert a debug page that displays diagnostic information.

**Outcomes**

* Deploy a pod from a container image.
* Retrieve the status and events of a pod.
* Troubleshoot a failed pod.
* Edit pod resources.
* Copy files to a running pod for diagnostic purposes.
* Use port forwarding to connect to a running pod.

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

This command ensures that exercise resources are available.

[student@workstation ~]$ **lab start pods-review**

**Instructions**

The API URL of your OpenShift cluster is https://api.ocp4.example.com:6443, and the oc command is already installed on your workstation machine.

Log in to the OpenShift cluster as the developer user with the developer password.

Use the pods-review project for your work.

1. Log in to the OpenShift cluster and change to the pods-review project.
   1. Log in to the OpenShift cluster.
   2. [student@workstation ~]$ **oc login -u developer -p developer \**
   3. **https://api.ocp4.example.com:6443**

*...output omitted...*

* 1. Select the pods-review project.
  2. [student@workstation ~]$ **oc project pods-review**
  3. Now using project "pods-review" on server "https://api.ocp4.example.com:6443".

*...output omitted...*

1. Hide Solution
2. Deploy a pod named webphp that uses the registry.ocp4.example.com:8443/redhattraining/webphp:v1 container image. Determine why the pod fails to start.

Show Solution

1. Troubleshoot the failed webphp pod by creating a debug pod.

Show Solution

1. The application developer resolved the identified issue in the registry.ocp4.example.com:8443/redhattraining/webphp:v2 container image. In a terminal window, edit the webphp pod resource to use the v2 image tag. Retrieve the status of the webphp pod. Then, confirm that the user in the container is an unprivileged user and belongs to the root group. Confirm that the root group permissions are correct for the /run/httpd directory.
   1. Use the terminal to edit the webphp pod resource.

[student@workstation ~]$ **oc edit pod/webphp**

* 1. Update the .spec.containers.image object value to use the :v2 image tag.
  2. *...output omitted...*
  3. spec:
  4. containers:
  5. - image: registry.ocp4.example.com:8443/redhattraining/webphp:**v2**
  6. imagePullPolicy: IfNotPresent

*...output omitted...*

* 1. Verify the status of the webphp pod.
  2. [student@workstation ~]$ **oc get pods**
  3. NAME READY STATUS RESTARTS AGE

webphp 1/1 Running 9 (2m9s ago) 18m

* 1. Retrieve the UID and GID of the user in the container to confirm that the user is an unprivileged user.
  2. [student@workstation ~]$ **oc exec -it webphp -- id**

uid=1000680000(1000680000) gid=0(root) groups=0(root),1000680000

Your UID and GID values might differ from the previous output.

* 1. Confirm that the permissions for the /run/httpd directory are correct.
  2. [student@workstation ~]$ **oc exec -it webphp -- ls -la /run/**
  3. total 0
  4. drwxr-xr-x. 1 root root 70 Dec 20 19:01 .
  5. dr-xr-xr-x. 1 root root 39 Dec 20 19:01 ..
  6. -rw-r--r--. 1 root root 0 Dec 20 18:45 .containerenv
  7. drwxrwx---. 1 root root 41 Dec 20 19:01 httpd
  8. drwxr-xr-x. 2 root root 6 Oct 26 11:10 lock
  9. drwxrwxr-x. 1 root root 41 Dec 20 19:01 php-fpm

drwxr-xr-x. 4 root root 80 Dec 20 19:01 secrets

1. Hide Solution
2. Connect port 8080 on the Workstation machine to port 8080 on the webphp pod. In a new terminal window, retrieve the content of the pod's 127.0.0.1:8080/index.php web page to confirm that the pod is operational.

**Note**

The terminal window that you connect to the webphp pod must remain open for the remainder of the lab. This connection is necessary for the final lab step and for the lab grade command.

* 1. Connect to port 8080 on the webphp pod.
  2. [student@workstation ~]$ **oc port-forward pod/webphp 8080:8080**
  3. Forwarding from 127.0.0.1:8080 -> 8080

Forwarding from [::1]:8080 -> 8080

* 1. Open a second terminal window and then retrieve the 127.0.0.1:8080/index.php web page on the webphp pod.
  2. [student@workstation ~]$ **curl 127.0.0.1:8080/index.php**
  3. <html>
  4. <body>
  5. Hello, World!
  6. </body>

</html>

Hide Solution

1. An issue occurs with the PHP application that is running on the webphp pod. To debug the issue, the application developer requires diagnostic and configuration information for the PHP instance that is running on the webphp pod.

The ~/DO180/labs/pods-review directory contains a phpinfo.php file to generate debugging information for a PHP instance. Copy the phpinfo.php file to the /var/www/html/ directory on the webphp pod.

Then, confirm that the PHP debugging information is displayed when accessing the 127.0.0.1:8080/phpinfo.php from a web browser.

* 1. In the second terminal, copy the ~/DO180/labs/pods-review/phpinfo.php file to the webphp pod as the /var/www/html/phpinfo.php file.
  2. [student@workstation ~]$ **oc cp ~/DO180/labs/pods-review/phpinfo.php \**

**webphp:/var/www/html/phpinfo.php**

* 1. Open a web browser and access the 127.0.0.1:8080/phpinfo.php web page. Confirm that PHP debugging information is displayed.

A screenshot of a computer

Description automatically generated

PHP debugging information

* 1. Return to the terminal that is executing the oc port-forward command. Press **Ctrl**+**C** to end the connection.
  2. [student@workstation ~]$ **oc port-forward pod/webphp 8080:8080**
  3. Forwarding from 127.0.0.1:8080 -> 8080
  4. Forwarding from [::1]:8080 -> 8080
  5. Handling connection for 8080

**^C**[student@workstation ~]$

Hide Solution

**Evaluation**

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

[student@workstation ~]$ **lab grade pods-review**

**Finish**

As the student user on the workstation machine, use the lab command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

[student@workstation ~]$ **lab finish pods-review**

Quiz: Run Applications as Containers and Pods

**Run Applications as Containers and Pods**

Choose the correct answers to the following questions:

As the student user on the workstation machine, use Skopeo to log in to the registry.ocp4.example.com:8443 classroom container registry as the developer user with the developer password. Then, use the skopeo and oc image commands to answer the following questions.

Top of Form

|  |  |  |
| --- | --- | --- |
|  | | |
| **1.** | Which command is executed in the registry.ocp4.example.com:8443/ubi8/php-74:latest container image? |  |
| A |  | httpd -D FOREGROUND |
| B |  | run-mysqld |
| C |  | php-fpm -D, httpd -D FOREGROUND |
| D |  | /bin/sh -c $STI\_SCRIPTS\_PATH/usage |
| E |  | /bin/sh -c php-fpm -D |

1. CheckResetShow Solution

Bottom of Form

Top of Form

|  |  |  |
| --- | --- | --- |
|  | | |
| **2.** | Using the UID value for the registry.ocp4.example.com:8443/ubi8/php-74:latest container image, identify the two correct deployment scenarios. (Choose two.) |  |
| A |  | The container executes as the root privileged user when a regular OpenShift user account deploys it in a pod. |
| B |  | The container user UID value is 1001 when a regular OpenShift user account deploys it in a pod. |
| C |  | The container executes as a non-privileged user when it is deployed in a pod. |
| D |  | The container executes as the root privileged user when an OpenShift cluster administrator deploys it in a pod. |
| E |  | The container user UID value is 1001 when an OpenShift cluster administrator deploys it in a pod. |

1. CheckResetShow Solution

Bottom of Form

Top of Form

|  |  |  |
| --- | --- | --- |
|  | | |
| **3.** | What is the ID/hash for the registry.ocp4.example.com:8443/rhel9/mysql-80:1-237 container image? |  |
| A |  | sha256:90b4…​a3a4 |
| B |  | sha256:4243…​71a6 |
| C |  | sha256:3843…​7caf |
| D |  | sha256:d282…​f38f |
| E |  | sha256:0027…​8c72 |

1. CheckResetShow Solution

Bottom of Form

Top of Form

|  |  |  |
| --- | --- | --- |
|  | | |
| **4.** | Which two environment variables and their values are specified in the registry.ocp4.example.com:8443/rhel8/postgresql-13:latest container image? (Choose two.) |  |
| A |  | NAME=PHP 7.4 |
| B |  | APP\_DATA=/opt/app-root/src/bin |
| C |  | HOME=/var/lib/pgsql |
| D |  | PHP\_SYSCONF\_FILE=/etc/ |
| E |  | PGUSER=postgres |

1. CheckResetShow Solution

Bottom of Form

Top of Form

|  |  |  |
| --- | --- | --- |
|  | | |
| **5.** | Which three tags are available for the registry.ocp4.example.com:8443/ubi8/ubi container image? (Choose three.) |  |
| A |  | v1 |
| B |  | 1 |
| C |  | latest |
| D |  | 8.1 |
| E |  | 8.4 |
| F |  | 1.8 |
| G |  | 8.0 |

1. CheckResetShow Solution

Bottom of Form

Top of Form

|  |  |  |
| --- | --- | --- |
|  | | |
| **6.** | Which two container images run as a privileged user when an OpenShift cluster administrator deploys a pod from the image? (Choose two.) |  |
| A |  | registry.ocp4.example.com:8443/rhel8/postgresql-13:latest |
| B |  | registry.ocp4.example.com:8443/ubi8/php-74:latest |
| C |  | registry.ocp4.example.com:8443/ubi8/nodejs-16:latest |
| D |  | registry.ocp4.example.com:8443/rhel8/mysql-80:latest |
| E |  | registry.ocp4.example.com:8443/ubi8/python-39 |

1. CheckResetShow Solution

Bottom of Form

Top of Form

|  |  |  |
| --- | --- | --- |
|  | | |
| **7.** | Which ports are exposed in the registry.ocp4.example.com:8443/rhel8/mysql-80 container image? |  |
| A |  | 8080/tcp |
| B |  | 3363/tcp |
| C |  | 8443/tcp |
| D |  | 3306/tcp |
| E |  | 3360/tcp |

1. CheckResetShow Solution

Bottom of Form

[Previous](https://rha.ole.redhat.com/rha/app/courses/do180-4.14/pages/ch03s07/c2358540-87d5-48de-b49e-6f23bdcd629c)[Next](https://rha.ole.redhat.com/rha/app/courses/do180-4.14/pages/ch03s09/c2358540-87d5-48de-b49e-6f23bdcd629c)

Summary

* A container is an encapsulated process that includes the required runtime dependencies for an application to run.
* OpenShift uses Kubernetes to manage pods. Pods consist of one or more containers that share resources, such as selected namespaces and networking, and represent a single application.
* Container images can create container instances, which are executable versions of the image, and include references to networking, disks, and other runtime necessities.
* Container image registries, such as Quay.io and the Red Hat Container Catalog, are the preferred way to distribute container images to many users and hosts.
* The oc image command and Skopeo, among other tools, can inspect and manage container images.
* Containers are immutable and ephemeral. Thus, updating a running container is best reserved for troubleshooting problematic containers.