Chapter 3. Run Applications as Containers and Pods

Abstract

Goal	Run and troubleshoot containerized applications as unmanaged Kubernetes pods.
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.

Running Containers in Pods

Kubernetes and Red Hat 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.

Use the `run` command to create single pods for interactive debugging or temporary tasks. For enterprise applications, using declarative YAML manifests with workload resources such as `Deployment` resources is the recommended practice for ensuring reproducibility and manageability. 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/ubi10/httpd-24` container image.

```
[user@host ~]$ oc run web-server \
--image registry.access.redhat.com/ubi10/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.

```
oc 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/ubi10/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 define a restart policy for a pod by using the `--restart` option. A pod restart policy determines how the cluster responds when containers in that pod exit. The accepted values are `Always`, `OnFailure`, or `Never`.

Policy	Description
`Always`	The cluster continuously tries to restart a container that exits, whether it exited successfully or with an error. This policy is the default.
`OnFailure`	The cluster restarts a container only if it exits with an error.
`Never`	The cluster does not try to restart exited or failed containers. 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/ubi10/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 ~]$ oc run -it my-app --rm \
--image registry.access.redhat.com/ubi10/ubi --restart Never --command -- date
Mon Feb 20 22:38:50 UTC 2023
pod "my-app" 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, Red Hat 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...
```

The values for `supplemental-groups` and `uid-range` use the `base/size` format. In the `1000710000/10000` example, the `1000710000` number is the starting ID of the block that is allocated to this project. The `10000` number is the size of the block. Therefore, the project can assign up to 10,000 unique UIDs and GIDs to containers that are running within it, starting from the base ID of `1000710000`.

With Red Hat OpenShift default security policies, regular cluster users cannot choose the `USER` or UIDs for their containers. When a regular cluster user creates a pod, Red Hat OpenShift ignores the `USER` instruction in the container image. Instead, Red Hat OpenShift assigns a random UID to the container's primary process from the `uid-range` that is defined in the project's annotations.

The primary group ID (GID) of the process is always `0`, which is the `root` group. Because the process runs with a non-root UID but belongs to the `root` group, any files or directories that the container writes to must be owned by the `root` group and have group write permissions.

This feature is essential for security. Although the process in the container belongs to the `root` group, its high-numbered, non-root UID makes it 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 process in the container runs as the `root` privileged account, with a UID of `0`. 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. By assigning a distinct block of UIDs to each project, Red Hat OpenShift ensures that containers in different projects do not run with the same UID, which is a critical security practice.

Pod Security

The Kubernetes Pod Security Admission (PSA) controller issues a warning when a pod is created without a defined security context. Red Hat OpenShift Container Platform 4.18 integrates the PSA controller, which enforces security profiles (privileged, baseline, restricted) at the namespace level. Security contexts grant or deny OS-level privileges to pods.

Red Hat OpenShift uses the Security Context Constraints (SCC) controller to provide safe defaults for pod security. SCCs grant pods the necessary permissions to meet the requirements of the enforced PSA profile. You can ignore pod security warnings in these course exercises. SCC controllers are discussed in more detail in the
Red Hat OpenShift Administration II: Operating a Production Kubernetes Cluster (DO280) course.

Execute Commands in Running Containers

To execute a command in a running container in a pod, you can use the `exec` command with the `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 ~]$ oc 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 ~]$ oc 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 `oc` CLI provides. The command includes the following options:

Option	Description
`-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 specific container in a multicontainer pod. If this option is omitted, then logs are shown from the first container that is defined in the pod or from the container that the `kubectl.kubernetes.io/default-container` annotation on the pod specifies.
`-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.

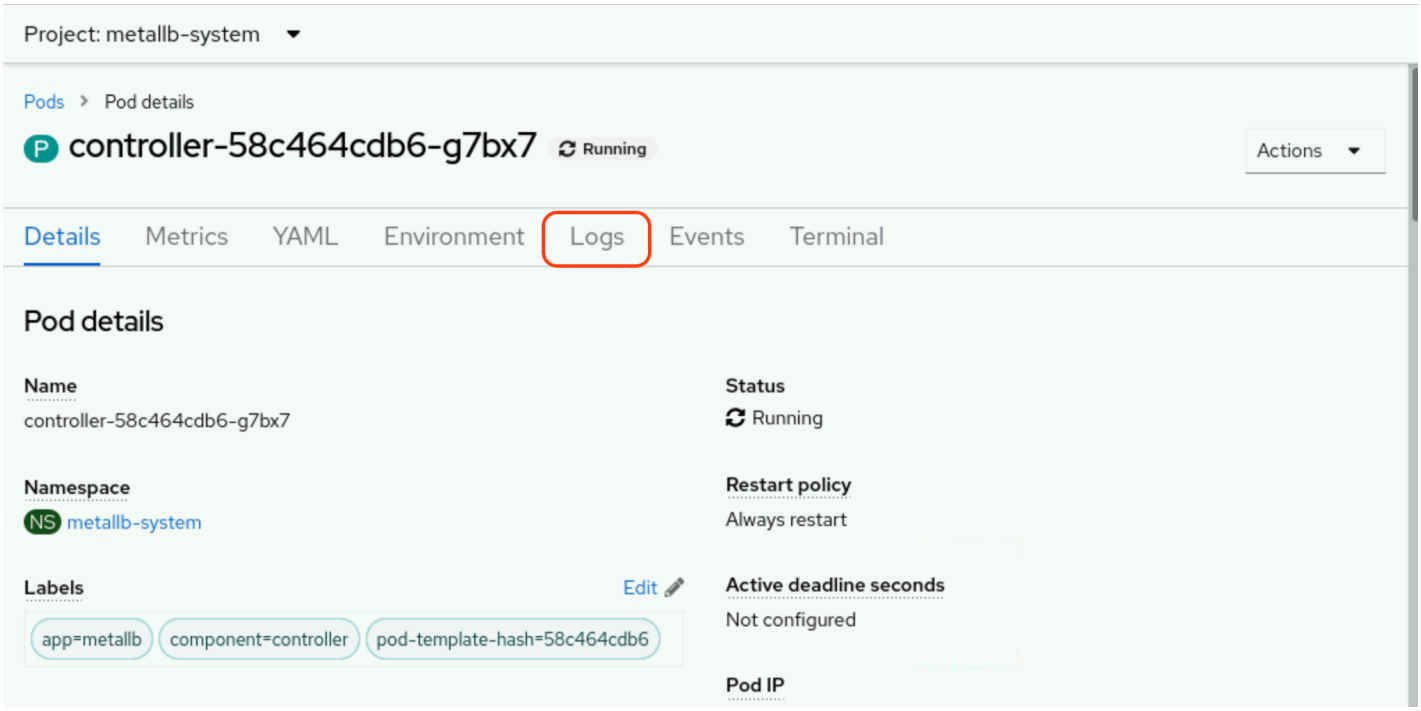
The following example restricts the `oc logs` command output to the 10 most recent log lines:

```
[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 20210
514 (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".
```

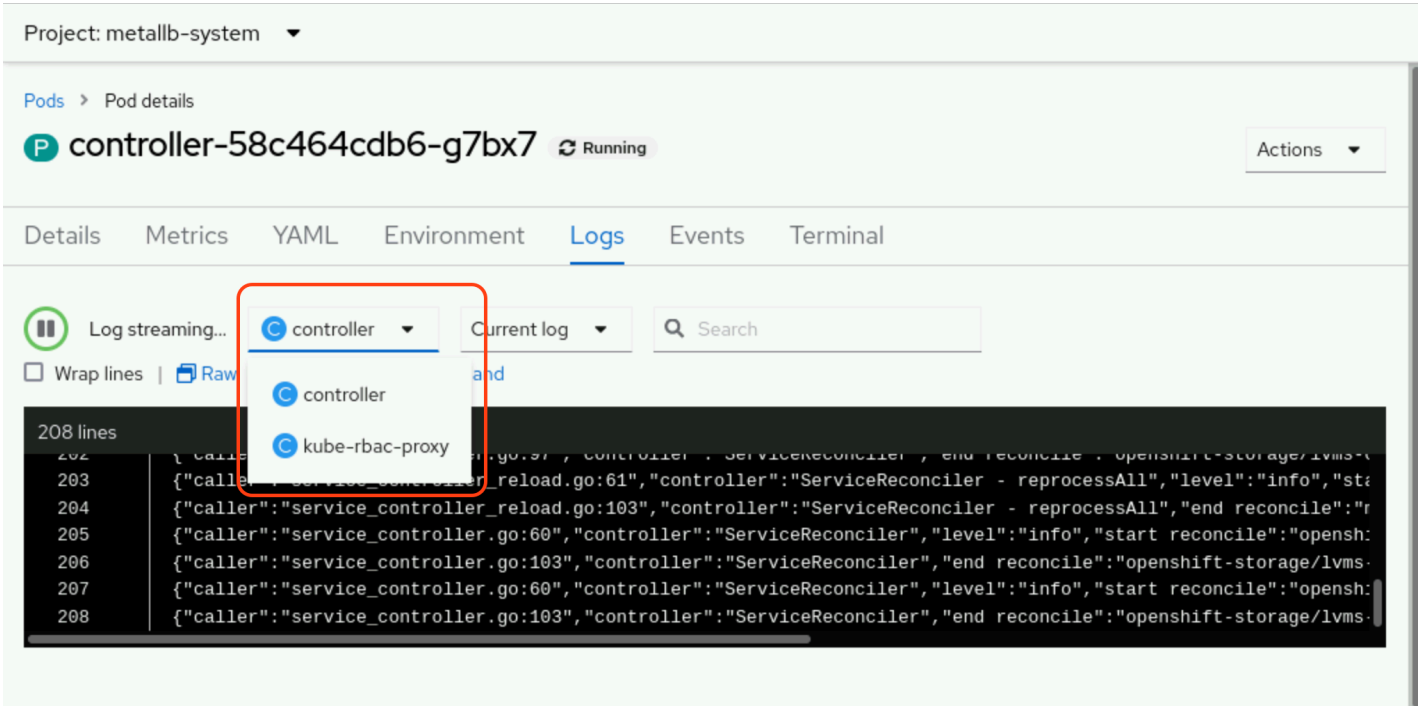
You can also use the `attach` *`pod-name`* `-c` *`container-name`* `-it` command to connect to and start an interactive session on a running container in a pod. The `-c` *`container-name`* option is required for multicontainer pods. If the container name is omitted, then Kubernetes uses the `kubectl.kubernetes.io/default-container` 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.



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

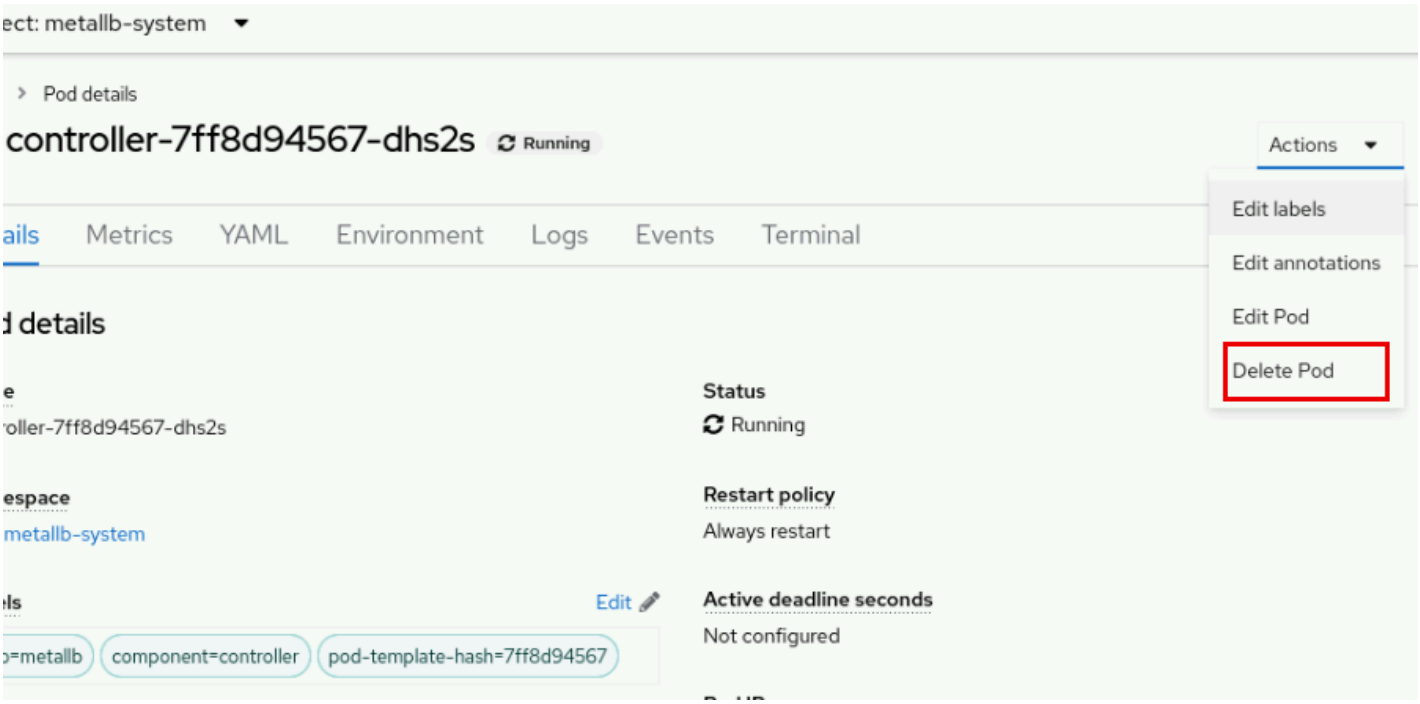


Deleting Resources

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.



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

```
[user@host ~]$ oc 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 | oc 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 ~]$ oc delete pod php-app --force
```

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

```
[user@host ~]$ oc 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 a Red Hat 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 Red Hat 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:

Command	Description
`crictl pods`	List all pods on a node.
`crictl image`	List 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 ~]$ oc 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. Red Hat 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 po...
```

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...
```

Alternatively, use the built-in methods such as the `go-template` option to parse out the ID without using external tools.

```
sh-5.1# crictl inspect --output go-template \
--template '{{.info.pid}}' 27943ae4f3024
43453
```

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 --switche...
4026531837 user      530     1 root       /usr/lib/systemd/systemd --switche...
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 lau...
1000690+      64       1  0 18:49 ?        00:00:00 postgres: stats collector
1000690+      65       1  0 18:49 ?        00:00:00 postgres: logical replic...
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 lau...
1000690+      64       1  0 18:49 ?        00:00:00 postgres: stats collector
1000690+      65       1  0 18:49 ?        00:00:00 postgres: logical replic...
root       10058       0  0 20:45 ?        00:00:00 ps -ef
```

REFERENCES

Container Runtime Interface (CRI) CLI

A Guide to OpenShift and UIDs

For more information about resource log files, refer to the *Red Hat Logging* documentation at https://docs.redhat.com/en/documentation/red_hat_openshift_logging/6.3/

Manage Containers in Namespaces by Using nsenter