Red Hat Enterprise Linux 8

Building, running, and managing containers

Using Podman, Buildah, and Skopeo on Red Hat Enterprise Linux 8

Last Updated: 2022-12-05

Red Hat Enterprise Linux 8 Building, running, and managing containers

Using Podman, Buildah, and Skopeo on Red Hat Enterprise Linux 8

Legal Notice

Abstract

Red Hat Enterprise Linux 8 offers a number of command-line tools for working with container images. Podman manages pods and container images. With Buildah you can build, update, and manage container images. With Skopeo, you copy and inspect images in remote repositories.

Table of Contents

MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright's message.

PROVIDING FEEDBACK ON RED HAT DOCUMENTATION

We appreciate your feedback on our documentation. Let us know how we can improve it.

Submitting comments on specific passages

1. View the documentation in the **Multi-page HTML** format and ensure that you see the **Feedback** button in the upper right corner after the page fully loads.
2. Use your cursor to highlight the part of the text that you want to comment on.
3. Click the **Add Feedback** button that appears near the highlighted text.
4. Add your feedback and click **Submit**.

Submitting feedback through Bugzilla (account required)

1. Log in to the Bugzilla website.
2. Select the correct version from the **Version** menu.
3. Enter a descriptive title in the **Summary** field.
4. Enter your suggestion for improvement in the **Description** field. Include links to the relevant parts of the documentation.
5. Click **Submit Bug**.

CHAPTER 1. STARTING WITH CONTAINERS

Linux containers have emerged as a key open source application packaging and delivery technology, combining lightweight application isolation with the flexibility of image-based deployment methods. Red Hat Enterprise Linux implements Linux containers using core technologies such as:

- Control groups (cgroups) for resource management
- Namespaces for process isolation
- SELinux for security
- Secure multi-tenancy

These technologies reduce the potential for security exploits and provide you with an environment for producing and running enterprise-quality containers.

Red Hat OpenShift provides powerful command-line and Web UI tools for building, managing, and running containers in units referred to as pods. Red Hat allows you to build and manage individual containers and container images outside of OpenShift. This guide describes the tools provided to perform those tasks that run directly on RHEL systems.

Unlike other container tools implementations, the tools described here do not center around the monolithic Docker container engine and **docker** command. Instead, Red Hat provides a set of command-line tools that can operate without a container engine. These include:

- **podman** - for directly managing pods and container images (**run**, **stop**, **start**, **ps**, **attach**, **exec**, and so on)
- **buildah** - for building, pushing, and signing container images
- **skopeo** - for copying, inspecting, deleting, and signing images
- **runc** - for providing container run and build features to podman and buildah
- **crun** - an optional runtime that can be configured and gives greater flexibility, control, and security for rootless containers

Because these tools are compatible with the Open Container Initiative (OCI), they can be used to manage the same Linux containers that are produced and managed by Docker and other OCI-compatible container engines. However, they are especially suited to run directly on Red Hat Enterprise Linux, in single-node use cases.

For a multi-node container platform, see OpenShift and Using the CRI-O Container Engine for details.

1.1. CHARACTERISTICS OF PODMAN, BUILDAH, AND SKOPEO

The Podman, Skopeo, and Buildah tools were developed to replace Docker command features. Each tool in this scenario is more lightweight and focused on a subset of features.

The main advantages of Podman, Skopeo and Buildah tools include:

- Running in rootless mode - rootless containers are much more secure, as they run without any added privileges

- No daemon required - these tools have much lower resource requirements at idle, because if you are not running containers, Podman is not running. Docker, conversely, have a daemon always running
- Native systemd integration - Podman allows you to create systemd unit files and run containers as system services

The characteristics of Podman, Skopeo, and Buildah include:

- Podman, Buildah, and the CRI-O container engine all use the same back-end store directory, **/var/lib/containers**, instead of using the Docker storage location **/var/lib/docker**, by default.
- Although Podman, Buildah, and CRI-O share the same storage directory, they cannot interact with each other's containers. Those tools can share images.
- To interact programmatically with Podman, you can use the Podman v2.0 RESTful API, it works in both a rootful and a rootless environment. For more information, see chapter Using the container-tools API.

Additional resources

- Say "Hello" to Buildah, Podman, and Skopeo
- Podman and Buildah for Docker users
- Buildah: A tool for building OCI container images
- Podman: A tool for managing OCI containers and pods
- Skopeo: A tool for copying and inspecting container images

1.2. OVERVIEW OF PODMAN COMMANDS

Table 1.1 shows a list of commands you can use with the **podman** command. Use **podman -h** to see a list of all Podman commands.

Table 1.1. Commands supported by podman

podman command	Description	podman command	Description
attach	Attach to a running container	**commit**	Create new image from changed container
build	Build an image using Containerfile instructions	**create**	Create, but do not start, a container
diff	Inspect changes on container's filesystems	**exec**	Run a process in a running container
export	Export container's filesystem contents as a tar archive	**help, h**	Shows a list of commands or help for one command

history	Show history of a specified image	**images**	List images in local storage
import	Import a tarball to create a filesystem image	**info**	Display system information
inspect	Display the configuration of a container or image	**kill**	Send a specific signal to one or more running containers
load	Load an image from an archive	**login**	Login to a container registry
logout	Logout of a container registry	**logs**	Fetch the logs of a container
mount	Mount a working container's root filesystem	**pause**	Pauses all the processes in one or more containers
ps	List containers	**port**	List port mappings or a specific mapping for the container
pull	Pull an image from a registry	**push**	Push an image to a specified destination
restart	Restart one or more containers	**rm**	Remove one or more containers from the host. Add **-f** if running.
rmi	removes one or more images from local storage	**run**	run a command in a new container
save	Save image to an archive	**search**	search registry for image
start	Start one or more containers	**stats**	Display percentage of CPU, memory, network I/O, block I/O and PIDs for one or more containers
stop	Stop one or more containers	**tag**	Add an additional name to a local image

top	Display the running processes of a container	**umount, unmount**	Unmount a working container's root filesystem
unpause	Unpause the processes in one or more containers	**version**	Display podman version information
wait	Block on one or more containers		

Additional resources

- Podman Basics Cheat Sheet
- 5 Podman features to try now

1.3. RUNNING CONTAINERS WITHOUT DOCKER

Red Hat removed the Docker container engine and the docker command from RHEL 8.

If you still want to use Docker in RHEL, you can get Docker from different upstream projects, but it is unsupported in RHEL 8.

- You can install the **podman-docker** package, every time you run a **docker** command, it actually runs a **podman** command.
- Podman also supports the Docker Socket API, so the **podman-docker** package also sets up a link between **/var/run/docker.sock** and **/var/run/podman/podman.sock**. As a result, you can continue to run your Docker API commands with **docker-py** and **docker-compose** tools without requiring the Docker daemon. Podman will service the requests.
- The **podman** command, like the **docker** command, can build container images from a **Containerfile** or **Dockerfile**. The available commands that are usable inside a **Containerfile** and a **Dockerfile** are equivalent.
- Options to the **docker** command that are not supported by **podman** include network, node, plugin (**podman** does not support plugins), rename (use rm and create to rename containers with **podman**), secret, service, stack, and swarm (**podman** does not support Docker Swarm). The container and image options are used to run subcommands that are used directly in **podman**.

Additional resources

- Podman and Buildah for Docker users

1.4. CHOOSING A RHEL ARCHITECTURE FOR CONTAINERS

Red Hat provides container images and container-related software for the following computer architectures:

- AMD64 and Intel 64 (base and layered images; no support for 32-bit architectures)

- PowerPC 8 and 9 64-bit (base image and most layered images)
- 64-bit IBM Z (base image and most layered images)
- ARM 64-bit (base image only)

Although not all Red Hat images were supported across all architectures at first, nearly all are now available on all listed architectures.

Additional resources

- Universal Base Images (UBI): Images, repositories, and packages

1.5. GETTING CONTAINER TOOLS

This procedure shows how you can install the **container-tools** module which contains the Podman, Buildah, Skopeo, CRIU, Udica, and all required libraries.

Procedure

1. Install RHEL.
2. Register RHEL: Enter your user name and password. The user name and password are the same as your login credentials for Red Hat Customer Portal:

```
# subscription-manager register
Registering to: subscription.rhsm.redhat.com:443/subscription
Username: <username>
Password: <password>
```

3. Subscribe to RHEL.
   - To auto-subscribe to RHEL:

```
# subscription-manager attach --auto
```

   - To subscribe to RHEL by Pool ID:

```
# subscription-manager attach --pool PoolID
```

4. Install the **container-tools** module:

```
# yum module install -y container-tools
```

5. Optional. Install the **podman-docker** package:

```
# yum install -y podman-docker
```

   The **podman-docker** package replaces the Docker command-line interface and **docker-api** with the matching Podman commands instead.

1.6. SETTING UP ROOTLESS CONTAINERS

Running the container tools such as Podman, Skopeo, or Buildah as a user with superuser privileges (root user) is the best way to ensure that your containers have full access to any feature available on your system. However, with the feature called "Rootless Containers" generally available as of Red Hat Enterprise Linux 8.1, you can work with containers as a regular user.

Although container engines, such as Docker, let you run Docker commands as a regular (non-root) user, the Docker daemon that carries out those requests runs as root. As a result, regular users can make requests through their containers that can harm the system. By setting up rootless container users, system administrators prevent potentially damaging container activities from regular users, while still allowing those users to safely run most container features under their own accounts.

This procedure describes how to set up your system to use Podman, Skopeo, and Buildah tools to work with containers as a non-root user (rootless). It also describes some of the limitations you will encounter, because regular user accounts do not have full access to all operating system features that their containers might need to run.

Prerequisites

- You need to become a root user to set up your RHEL system to allow non-root user accounts to use container tools.

Procedure

1. Install RHEL.
2. Install the **podman** package:

```
# yum install podman -y
```

3. Create a new user account:

```
# useradd -c "Joe Jones" joe
# passwd joe
```

   - The user is automatically configured to be able to use rootless Podman.
   - The **useradd** command automatically sets the range of accessible user and group IDs automatically in the **/etc/subuid** and **/etc/subgid** files.
   - If you change the **/etc/subuid** or **/etc/subgid** manually, you have to run the **podman system migrate** command to allow the new changes to be applied.

4. Connect to the user:

```
$ ssh joe@server.example.com
```

NOTE

Do not use **su** or **su -** commands because these commands do not set the correct environment variables.

5. Pull the **registry.access.redhat.com/ubi8/ubi** container image:

```
$ podman pull registry.access.redhat.com/ubi8/ubi
```

6. Run the container named **myubi** and display the OS version:

```
$ podman run --rm --name=myubi registry.access.redhat.com/ubi8/ubi \ cat /etc/os-release
NAME="Red Hat Enterprise Linux"
VERSION="8 (Plow)"
```

Additional resources

- Rootless containers with Podman: The basics
- **podman-system-migrate** man page

1.7. UPGRADING TO ROOTLESS CONTAINERS

This section shows how to upgrade to rootless containers from Red Hat Enterprise Linux 7. You must configure user and group IDs manually.

Here are some things to consider when upgrading to rootless containers from Red Hat Enterprise Linux 7:

- If you set up multiple rootless container users, use unique ranges for each user.
- Use 65536 UIDs and GIDs for maximum compatibility with existing container images, but the number can be reduced.
- Never use UIDs or GIDs under 1000 or reuse UIDs or GIDs from existing user accounts (which, by default, start at 1000).

Prerequisites

- The user account has been created.

Procedure

- Run the **usermod** command to assign UIDs and GIDs to a user:

```
# usermod --add-subuids 200000-201000 --add-subgids 200000-201000 username
```

  - The **usermod --add-subuid** command manually adds a range of accessible user IDs to the user's account.
  - The **usermod --add-subgids** command manually adds a range of accessible user GIDs and group IDs to the user's account.

Verification steps

- Check that the UIDs and GIDs are set properly:

```
# grep username /etc/subuid /etc/subgid
#/etc/subuid:username:200000:1001
#/etc/subgid:username:200000:1001
```

1.8. SPECIAL CONSIDERATIONS FOR ROOTLESS CONTAINERS

There are several considerations when running containers as a non-root user:

- The path to the host container storage is different for root users (**/var/lib/containers/storage**) and non-root users (**$HOME/.local/share/containers/storage**).
- Users running rootless containers are given special permission to run as a range of user and group IDs on the host system. However, they have no root privileges to the operating system on the host.
- If you change the **/etc/subuid** or **/etc/subgid** manually, you have to run the **podman system migrate** command to allow the new changes to be applied.
- If you need to configure your rootless container environment, create configuration files in your home directory (**$HOME/.config/containers**). Configuration files include **storage.conf** (for configuring storage) and **containers.conf** (for a variety of container settings). You could also create a **registries.conf** file to identify container registries that are available when you use Podman to pull, search, or run images.
- There are some system features you cannot change without root privileges. For example, you cannot change the system clock by setting a **SYS_TIME** capability inside a container and running the network time service (**ntpd**). You have to run that container as root, bypassing your rootless container environment and using the root user's environment. For example:

  ```
  # podman run -d --cap-add SYS_TIME ntpd
  ```

  Note that this example allows **ntpd** to adjust time for the entire system, and not just within the container.
- A rootless container cannot access a port numbered less than 1024. Inside the rootless container namespace it can, for example, start a service that exposes port 80 from an httpd service from the container, but it is not accessible outside of the namespace:

  ```
  $ podman run -d httpd
  ```

  However, a container would need root privileges, using the root user's container environment, to expose that port to the host system:

  ```
  # podman run -d -p 80:80 httpd
  ```
- The administrator of a workstation can allow users to expose services on ports numbered lower than 1024, but they should understand the security implications. A regular user could, for example, run a web server on the official port 80 and make external users believe that it was configured by the administrator. This is acceptable on a workstation for testing, but might not be a good idea on a network-accessible development server, and definitely should not be done on production servers. To allow users to bind to ports down to port 80 run the following command:

  ```
  # echo 80 > /proc/sys/net/ipv4/ip_unprivileged_port_start
  ```

Additional resources

- Shortcomings of Rootless Podman

1.9. ADDITIONAL RESOURCES

- A Practical Introduction to Container Terminology

CHAPTER 2. TYPES OF CONTAINER IMAGES

The container image is a binary that includes all of the requirements for running a single container, and metadata describing its needs and capabilities.

There are two types of container images:

- Red Hat Enterprise Linux Base Images (RHEL base images)
- Red Hat Universal Base Images (UBI images)

Both types of container images are built from portions of Red Hat Enterprise Linux. By using these containers, users can benefit from great reliability, security, performance and life cycles.

The main difference between the two types of container images is that the UBI images allow you to share container images with others. You can build a containerized application using UBI, push it to your choice of registry server, easily share it with others, and even deploy it on non-Red Hat platforms. The UBI images are designed to be a foundation for cloud-native and web applications use cases developed in containers.

2.1. GENERAL CHARACTERISTICS OF RHEL CONTAINER IMAGES

Following characteristics apply to both RHEL base images and UBI images.

In general, RHEL container images are:

- **Supported**: Supported by Red Hat for use with containerized applications. They contain the same secured, tested, and certified software packages found in Red Hat Enterprise Linux.
- **Cataloged**: Listed in the Red Hat Container Catalog, with descriptions, technical details, and a health index for each image.
- **Updated**: Offered with a well-defined update schedule, to get the latest software, see Red Hat Container Image Updates article.
- **Tracked**: Tracked by Red Hat Product Errata to help understand the changes that are added into each update.
- **Reusable**: The container images need to be downloaded and cached in your production environment once. Each container image can be reused by all containers that include it as their foundation.

2.2. CHARACTERISTICS OF UBI IMAGES

The UBI images allow you to share container images with others. Four UBI images are offered: micro, minimal, standard, and init. Pre-build language runtime images and YUM repositories are available to build your applications.

Following characteristics apply to UBI images:

- **Built from a subset of RHEL content** Red Hat Universal Base images are built from a subset of normal Red Hat Enterprise Linux content.
- **Redistributable**: UBI images allow standardization for Red Hat customers, partners, ISVs, and others. With UBI images, you can build your container images on a foundation of official Red Hat software that can be freely shared and deployed.

- **Provide a set of four base images** micro, minimal, standard, and init.
- **Provide a set of pre-built language runtime container images**The runtime images based on Application Streams provide a foundation for applications that can benefit from standard, supported runtimes such as python, perl, php, dotnet, nodejs, and ruby.
- **Provide a set of associated YUM repositories** YUM repositories include RPM packages and updates that allow you to add application dependencies and rebuild UBI container images.
    - The **ubi-8-baseos** repository holds the redistributable subset of RHEL packages you can include in your container.
    - The **ubi-8-appstream** repository holds Application streams packages that you can add to a UBI image to help you standardize the environments you use with applications that require particular runtimes.
    - **Adding UBI RPMs**: You can add RPM packages to UBI images from preconfigured UBI repositories. If you happen to be in a disconnected environment, you must allowlist the UBI Content Delivery Network (https://cdn-ubi.redhat.com) to use that feature. See the Connect to https://cdn-ubi.redhat.com solution for details.
- **Licensing**: You are free to use and redistribute UBI images, provided you adhere to the Red Hat Universal Base Image End User Licensing Agreement.

NOTE

All of the layered images are based on UBI images. To check on which UBI image is your image based, display the Containerfile in the Red Hat Container Catalog and ensure that the UBI image contains all required content.

Additional resources

- Introducing the Red Hat Universal Base Image
- Universal Base Images (UBI): Images, repositories, and packages
- All You Need to Know About Red Hat Universal Base Image
- FAQ - Universal Base Images

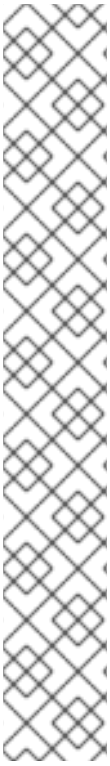
2.3. UNDERSTANDING THE UBI STANDARD IMAGES

The standard images (named **ubi**) are designed for any application that runs on RHEL. The key features of UBI standard images include:

- **init system**: All the features of the systemd initialization system you need to manage systemd services are available in the standard base images. These init systems let you install RPM packages that are pre-configured to start up services automatically, such as a Web server (**httpd**) or FTP server (**vsftpd**).
- **yum**: You have access to free yum repositories for adding and updating software. You can use the standard set of **yum** commands (**yum**, **yum-config-manager**, **yumdownloader**, and so on).
- **utilities**: Utilities include **tar**, **dmidecode**, **gzip**, **getfacl** and further acl commands, **dmsetup** and further device mapper commands, between other utilities not mentioned here.

2.4. UNDERSTANDING THE UBI INIT IMAGES

The UBI init images, named **ubi-init**, contain the systemd initialization system, making them useful for building images in which you want to run systemd services, such as a web server or file server. The init image contents are less than what you get with the standard images, but more than what is in the minimal images.



NOTE

Because the **ubi8-init** image builds on top of the **ubi8** image, their contents are mostly the same. However, there are a few critical differences:

- **ubi8-init**:
  - CMD is set to /**sbin**/**init** to start the systemd Init service by default
  - includes **ps** and process related commands (**procps-ng** package)
  - sets **SIGRTMIN+3** as the **StopSignal**, as systemd in **ubi8-init** ignores normal signals to exit (**SIGTERM** and **SIGKILL**), but will terminate if it receives **SIGRTMIN+3**
- **ubi8**:
  - CMD is set to /**bin**/**bash**
  - does not include **ps** and process related commands (**procps-ng** package)
  - does not ignore normal signals to exit (**SIGTERM** and **SIGKILL**)

2.5. UNDERSTANDING THE UBI MINIMAL IMAGES

The UBI minimal images, named **ubi-minimal** offer a minimized pre-installed content set and a package manager (**microdnf**`). As a result, you can use a **Containerfile** while minimizing the dependencies included in the image.

The key features of UBI minimal images include:

- **Small size**: Minimal images are about 92M on disk and 32M, when compressed. This makes it less than half the size of the standard images.
- **Software installation (microdnf)**: Instead of including the fully-developed **yum** facility for working with software repositories and RPM software packages, the minimal images includes the **microdnf** utility. The **microdnf** is a scaled-down version of **dnf** allowing you to enable and disable repositories, remove and update packages, and clean out cache after packages have been installed.
- **Based on RHEL packaging**: Minimal images incorporate regular RHEL software RPM packages, with a few features removed. Minimal images do not include initialization and service management system, such as systemd or System V init, Python run-time environment, and some shell utilities. You can rely on RHEL repositories for building your images, while carrying the smallest possible amount of overhead.
- **Modules for microdnf are supported**: Modules used with **microdnf** command let you install multiple versions of the same software, when available. You can use **microdnf module enable**, **microdnf module disable**, and **microdnf module reset** to enable, disable, and reset a module

stream, respectively.

- For example, to enable the **nodejs:14** module stream inside the UBI minimal container, enter:

```
# microdnf module enable nodejs:14
Downloading metadata...
...
Enabling module streams:
   nodejs:14

Running transaction test...
```

Red Hat only supports the latest version of UBI and does not support parking on a dot release. If you need to park on a specific dot release, please take a look at Extended Update Support.

2.6. UNDERSTANDING THE UBI MICRO IMAGES

The **ubi-micro** is the smallest possible UBI image, obtained by excluding a package manager and all of its dependencies which are normally included in a container image. This minimizes the attack surface of container images based on the **ubi-micro** image and is suitable for minimal applications, even if you use UBI Standard, Minimal, or Init for other applications. The container image without the Linux distribution packaging is called a Distroless container image.

CHAPTER 3. WORKING WITH CONTAINER IMAGES

The Podman tool is designed to work with container images. You can use this tool to pull the image, inspect, tag, save, load, redistribute, and define the image signature.

3.1. CONTAINER REGISTRIES

A container registry is a repository or collection of repositories for storing container images and container-based application artifacts. The registries that Red Hat provides are:

- registry.redhat.io (requires authentication)
- registry.access.redhat.com (requires no authentication)
- registry.connect.redhat.com (holds Red Hat Partner Connect program images)

To get container images from a remote registry, such as Red Hat's own container registry, and add them to your local system, use the **podman pull** command:

```
# podman pull <registry>[:<port>]/[<namespace>/]<name>:<tag>
```

where **<registry>[:<port>]/[<namespace>/]<name>:<tag>** is the name of the container image.

For example, the **registry.redhat.io/ubi8/ubi** container image is identified by:

- Registry server (**registry.redhat.io**)
- Namespace (**ubi8**)
- Image name (**ubi**)

If there are multiple versions of the same image, add a tag to explicitly specify the image name. By default, Podman uses the **:latest** tag, for example **ubi8/ubi:latest**.

Some registries also use *<namespace>* to distinguish between images with the same *<name>* owned by different users or organizations. For example:

Namespace	Examples (*<namespace>*/*<name>*)
organization	**redhat/kubernetes**, **google/kubernetes**
login (user name)	**alice/application**, **bob/application**
role	**devel/database**, **test/database**, **prod/database**

For details on the transition to registry.redhat.io, see Red Hat Container Registry Authentication . Before you can pull containers from registry.redhat.io, you need to authenticate using your RHEL Subscription credentials.

3.2. CONFIGURING CONTAINER REGISTRIES

You can find the list of container registries in the **registries.conf** configuration file. As a root user, edit the **/etc/containers/registries.conf** file to change the default system-wide search settings.

As a user, create the **$HOME/.config/containers/registries.conf** file to override the system-wide settings.

```
unqualified-search-registries = ["registry.fedoraproject.org", "registry.access.redhat.com", "docker.io"]
```

By default, the **podman pull** and **podman search** commands search for container images from registries listed in the **unqualified-search-registries** list in the given order.

Configuring a local container registry

You can configure a local container registry without the TLS verification. You have two options on how to disable TLS verification. First, you can use the **--tls-verify=false** option in Podman. Second, you can set **insecure=true** in the **registries.conf** file:

```
[[registry]]
location="localhost:5000"
insecure=true
```

Blocking a registry, namespace, or image

You can define registries the local system is not allowed to access. You can block a specific registry by setting **blocked=true**.

```
[[registry]]
location = "registry.example.org"
blocked = true
```

You can also block a namespace by setting the prefix to **prefix="registry.example.org/namespace"**. For example, pulling the image using the **podman pull registry. example.org/example/image:latest** command will be blocked, because the specified prefix is matched.

```
[[registry]]
location = "registry.example.org"
prefix="registry.example.org/namespace"
blocked = true
```

NOTE

prefix is optional, default value is the same as the **location** value.

You can block a specific image by setting **prefix="registry.example.org/namespace/image"**.

```
[[registry]]
location = "registry.example.org"
prefix="registry.example.org/namespace/image"
blocked = true
```

Mirroring registries

You can set a registry mirror in cases you cannot access the original registry. For example, you cannot connect to the internet, because you work in a highly-sensitive environment. You can specify

multiple mirrors that are contacted in the specified order. For example, when you run **podman pull registry.example.com/myimage:latest** command, the **mirror-1.com** is tried first, then **mirror-2.com**.

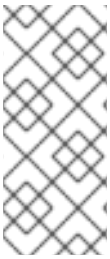
```
[[registry]]
location="registry.example.com"
[[registry.mirror]]
location="mirror-1.com"
[[registry.mirror]]
location="mirror-2.com"
```

Additional resources

- How to manage Linux container registries

3.3. SEARCHING FOR CONTAINER IMAGES

Using the **podman search** command you can search selected container registries for images. You can also search for images in the Red Hat Container Catalog. The Red Hat Container Registry includes the image description, contents, health index, and other information.



NOTE

The **podman search** command is not a reliable way to determine the presence or existence of an image. The **podman search** behavior of the v1 and v2 Docker distribution API is specific to the implementation of each registry. Some registries may not support searching at all. Searching without a search term only works for registries that implement the v2 API. The same holds for the **docker search** command.

This section explains how to search for the **postgresql-10** images in the quay.io registry.

Prerequisites

- The registry is configured.

Procedure

1. Authenticate to the registry:

```
# podman login quay.io
```

2. Search for the image:

   - To search for a particular image on a specific registry, enter:

```
# podman search quay.io/postgresql-10
INDEX      NAME                                      DESCRIPTION          STARS   OFFICIAL
AUTOMATED
redhat.io  registry.redhat.io/rhel8/postgresql-10         This container image ...  0
redhat.io  registry.redhat.io/rhscl/postgresql-10-rhel7   PostgreSQL is an  ...     0
```

   - Alternatively, to display all images provided by a particular registry, enter:

```
# podman search quay.io/
```

- To search for the image name in all registries, enter:

```
# podman search postgresql-10
```

To display the full descriptions, pass the **--no-trunc** option to the command.

Additional resources

- **podman-search** man page

3.4. PULLING IMAGES FROM REGISTRIES

Use the **podman pull** command to get the image to your local system.

Procedure

1. Log in to the registry.redhat.io registry:

```
$ podman login registry.redhat.io
Username: <username>
Password: <password>
Login Succeeded!
```

2. Pull the registry.redhat.io/ubi8/ubi container image:

```
$ podman pull registry.redhat.io/ubi8/ubi
```

Verification steps

- List all images pulled to your local system:

```
$ podman images
REPOSITORY                    TAG     IMAGE ID      CREATED      SIZE
registry.redhat.io/ubi8/ubi   latest  3269c37eae33  7 weeks ago  208 MB
```

Additional resources

- **podman-pull** man page

3.5. CONFIGURING SHORT-NAME ALIASES

Red Hat recommends always to pull an image by its fully-qualified name. However, it is customary to pull images by short names. For instance, you can use **ubi8** instead of **registry.access.redhat.com/ubi8:latest**.

The **registries.conf** file allows to specify aliases for short names, giving administrators full control over where images are pulled from. Aliases are specified in the **[aliases]** table in the form **"name" = "value"**. You can see the lists of aliases in the **/etc/containers/registries.conf.d** directory. Red Hat ships a set of aliases in this directory. For example, **podman pull ubi8** directly resolves to the right image, that is **registry.access.redhat.com/ubi8:latest**.

For example:

```
unqualified-search-registries=["registry.fedoraproject.org", “quay.io"]

[aliases]
"fedora"="registry.fedoraproject.org/fedora"
```

The short-names modes are:

- **enforcing**: If no matching alias is found during the image pull, Podman prompts the user to choose one of the unqualified-search registries. If the selected image is pulled successfully, Podman automatically records a new short-name alias in the **$HOME/.cache/containers/short-name-aliases.conf** file (rootless user) or in the **/var/cache/containers/short-name-aliases.conf** (root user). If the user cannot be prompted (for example, stdin or stdout are not a TTY), Podman fails. Note that the **short-name-aliases.conf** file has precedence over the **registries.conf** file if both specify the same alias.
- **permissive**: Similar to enforcing mode, but Podman does not fail if the user cannot be prompted. Instead, Podman searches in all unqualified-search registries in the given order. Note that no alias is recorded.
- **disabled**: All unqualified-search registries are tried in a given order, no alias is recorded.

NOTE

Red Hat recommends using fully qualified image names including registry, namespace, image name, and tag. When using short names, there is always an inherent risk of spoofing. Add registries that are trusted, that is, registries that do not allow unknown or anonymous users to create accounts with arbitrary names. For example, a user wants to pull the example container image from **example.registry.com registry**. If **example.registry.com** is not first in the search list, an attacker could place a different example image at a registry earlier in the search list. The user would accidentally pull and run the attacker image rather than the intended content.

Additional resources

- Container image short names in Podman

3.6. PULLING CONTAINER IMAGES USING SHORT-NAME ALIASES

You can use secure short names to get the image to your local system. The following procedure describes how to pull a **fedora** or **nginx** container image.

Procedure

- Pull the container image:
  - Pull the **fedora** image:

```
$ podman pull fedora
Resolved "fedora" as an alias (/etc/containers/registries.conf.d/000-shortnames.conf)
Trying to pull registry.fedoraproject.org/fedora:latest…
...
Storing signatures
...
```

Alias is found and the **registry.fedoraproject.org/fedora** image is securely pulled. The **unqualified-search-registries** list is not used to resolve **fedora** image name.

- Pull the **nginx** image:

```
$ podman pull nginx
? Please select an image:
registry.access.redhat.com/nginx:latest
registry.redhat.io/nginx:latest
  ▸ docker.io/library/nginx:latest
✔ docker.io/library/nginx:latest
Trying to pull docker.io/library/nginx:latest…
...
Storing signatures
...
```

If no matching alias is found, you are prompted to choose one of the **unqualified-search-registries** list. If the selected image is pulled successfully, a new short-name alias is recorded locally, otherwise an error occurs.

Verification

- List all images pulled to your local system:

```
$ podman images
REPOSITORY                              TAG     IMAGE ID      CREATED       SIZE
registry.fedoraproject.org/fedora       latest  28317703decd  12 days ago    184 MB
docker.io/library/nginx                 latest  08b152afcfae  13 days ago    137 MB
```

Additional resources

- Container image short names in Podman

3.7. LISTING IMAGES

Use the **podman images** command to list images in your local storage.

Prerequisites

- A pulled image is available on the local system.

Procedure

- List all images in the local storage:

```
$ podman images
REPOSITORY                      TAG     IMAGE ID      CREATED      SIZE
registry.access.redhat.com/ubi8/ubi  latest  3269c37eae33  6 weeks ago  208 MB
```

Additional resources

- **podman-images** man page

3.8. INSPECTING LOCAL IMAGES

After you pull an image to your local system and run it, you can use the **podman inspect** command to investigate the image. For example, use it to understand what the image does and check what software is inside the image. The **podman inspect** command displays information on containers and images identified by name or ID.

Prerequisites

- A pulled image is available on the local system.

Procedure

- Inspect the **registry.redhat.io/ubi8/ubi** image:

```
$ podman inspect registry.redhat.io/ubi8/ubi
…
 "Cmd": [
        "/bin/bash"
    ],
    "Labels": {
        "architecture": "x86_64",
        "build-date": "2020-12-10T01:59:40.343735",
        "com.redhat.build-host": "cpt-1002.osbs.prod.upshift.rdu2.redhat.com",
        "com.redhat.component": "ubi8-container",
        "com.redhat.license_terms": "https://www.redhat.com/...,
    "description": "The Universal Base Image is ...
    }
...
```

The **"Cmd"** key specifies a default command to run within a container. You can override this command by specifying a command as an argument to the **podman run** command. This ubi8/ubi container will execute the bash shell if no other argument is given when you start it with **podman run**. If an **"Entrypoint"** key was set, its value would be used instead of the **"Cmd"** value, and the value of **"Cmd"** is used as an argument to the Entrypoint command.

Additional resources

- **podman-inspect** man page

3.9. INSPECTING REMOTE IMAGES

Use the **skopeo inspect** command to display information about an image from a remote container registry before you pull the image to your system.

Procedure

- Inspect the **registry.redhat.io/ubi8/ubi-init** image:

```
# skopeo inspect docker://registry.redhat.io/ubi8/ubi-init
{
    "Name": "registry.redhat.io/ubi8/ubi8-init",
    "Digest": "sha256:c6d1e50ab...",
    "RepoTags": [
```

```
        ...
        "latest"
    ],
   "Created": "2020-12-10T07:16:37.250312Z",
    "DockerVersion": "1.13.1",
    "Labels": {
        "architecture": "x86_64",
        "build-date": "2020-12-10T07:16:11.378348",
        "com.redhat.build-host": "cpt-1007.osbs.prod.upshift.rdu2.redhat.com",
        "com.redhat.component": "ubi8-init-container",
        "com.redhat.license_terms": "https://www.redhat.com/en/about/red-hat-end-user-
license-agreements#UBI",
        "description": "The Universal Base Image Init is designed to run an init system as PID 1
for running multi-services inside a container
        ...
    }
}
```

Additional resources

- **skopeo-inspect** man page

3.10. COPYING CONTAINER IMAGES

You can use the **skopeo copy** command to copy a container image from one registry to another. For example, you can populate an internal repository with images from external registries, or sync image registries in two different locations.

Procedure

- Copy the **skopeo** container image from **docker://quay.io** to **docker://registry.example.com**:

```
$ skopeo copy docker://quay.io/skopeo/stable:latest
docker://registry.example.com/skopeo:latest
```

Additional resources

- **skopeo-copy** man page

3.11. COPYING IMAGE LAYERS TO A LOCAL DIRECTORY

You can use the **skopeo copy** command to copy the layers of a container image to a local directory.

Procedure

1. Create the **/var/lib/images/nginx** directory:

```
$ mkdir -p /var/lib/images/nginx
```

2. Copy the layers of the **docker://docker.io/nginx:latest image** to the newly created directory:

```
$ skopeo copy docker://docker.io/nginx:latest dir:/var/lib/images/nginx
```

Verification

- Display the content of the **/var/lib/images/nginx** directory:

```
$ ls /var/lib/images/nginx
08b11a3d692c1a2e15ae840f2c15c18308dcb079aa5320e15d46b62015c0f6f3
...
4fcb23e29ba19bf305d0d4b35412625fea51e82292ec7312f9be724cb6e31ffd  manifest.json
version
```

Additional resources

- **skopeo-copy** man page

3.12. TAGGING IMAGES

Use the **podman tag** command to add an additional name to a local image. This additional name can consist of several parts: *registryhost/username/NAME:tag*.

Prerequisites

- A pulled image is available on the local system.

Procedure

1. List all images:

```
$ podman images
REPOSITORY                    TAG     IMAGE ID      CREATED      SIZE
registry.redhat.io/ubi8/ubi   latest  3269c37eae33  7 weeks ago  208 MB
```

2. Assign the **myubi** name to the **registry.redhat.io/ubi8/ubi** image using either:
   - The image name:

```
$ podman tag registry.redhat.io/ubi8/ubi myubi
```

   - The image ID:

```
$ podman tag 3269c37eae33 myubi
```

     Both commands give you the same result.

3. List all images:

```
$ podman images
REPOSITORY                    TAG     IMAGE ID      CREATED      SIZE
registry.redhat.io/ubi8/ubi   latest  3269c37eae33  2 months ago  208 MB
localhost/myubi               latest  3269c37eae33  2 months ago  208 MB
```

   Notice that the default tag is **latest** for both images. You can see all the image names are assigned to the single image ID 3269c37eae33.

4. Add the **8** tag to the **registry.redhat.io/ubi8/ubi** image using either:

- The image name:

```
$ podman tag registry.redhat.io/ubi8/ubi myubi:8
```

- The image ID:

```
$ podman tag 3269c37eae33 myubi:8
```

Both commands give you the same result.

5. List all images:

```
$ podman images
REPOSITORY                    TAG     IMAGE ID      CREATED       SIZE
registry.redhat.io/ubi8/ubi   latest  3269c37eae33  2 months ago  208 MB
localhost/myubi               latest  3269c37eae33  2 months ago  208 MB
localhost/myubi               8       3269c37eae33  2 months ago  208 MB
```

Notice that the default tag is **latest** for both images. You can see all the image names are assigned to the single image ID 3269c37eae33.

After tagging the **registry.redhat.io/ubi8/ubi** image, you have three options to run the container:

- by ID (**3269c37eae33**)
- by name (**localhost/myubi:latest**)
- by name (**localhost/myubi:8**)

Additional resources

- **podman-tag** man page

3.13. SAVING AND LOADING IMAGES

Use the **podman save** command to save an image to a container archive. You can restore it later to another container environment or send it to someone else. You can use **--format** option to specify the archive format. The supported formats are:

- **docker-archive**
- **oci-archive**
- **oci-dir** (directory with oci manifest type)
- **docker-dir** (directory with v2s2 manifest type)

The default format is the **docker-dir** format.

Use the **podman load** command to load an image from the container image archive into the container storage.

Prerequisites

- A pulled image is available on the local system.

Procedure

1. Save the **registry.redhat.io/rhel8/rsyslog** image as a tarball:
   - In the default **docker-dir** format:

     ```
     $ podman save -o myrsyslog.tar registry.redhat.io/rhel8/rsyslog:latest
     ```

   - In the **oci-archive** format, using the **--format** option:

     ```
     $ podman save -o myrsyslog-oci.tar --format=oci-archive
     registry.redhat.io/rhel8/rsyslog
     ```

     The **myrsyslog.tar** and **myrsyslog-oci.tar** archives are stored in your current directory. The next steps are performed with the **myrsyslog.tar** tarball.
2. Check the file type of **myrsyslog.tar**:

   ```
   $ file myrsyslog.tar
   myrsyslog.tar: POSIX tar archive
   ```

3. To load the **registry.redhat.io/rhel8/rsyslog:latest** image from the **myrsyslog.tar**:

   ```
   $ podman load -i myrsyslog.tar
   ...
   Loaded image(s): registry.redhat.io/rhel8/rsyslog:latest
   ```

Additional resources

- **podman-save** man page

3.14. REDISTRIBUTING UBI IMAGES

Use **podman push** command to push a UBI image to your own, or a third party, registry and share it with others. You can upgrade or add to that image from UBI yum repositories as you like.

Prerequisites

- A pulled image is available on the local system.

Procedure

1. Optional: Add an additional name to the **ubi** image:

   ```
   # podman tag registry.redhat.io/ubi8/ubi registry.example.com:5000/ubi8/ubi
   ```

2. Push the **registry.example.com:5000/ubi8/ubi** image from your local storage to a registry:

   ```
   # podman push registry.example.com:5000/ubi8/ubi
   ```

   IMPORTANT

   While there are few restrictions on how you use these images, there are some restrictions about how you can refer to them. For example, you cannot call those images Red Hat

certified or Red Hat supported unless you certify it through the Red Hat Partner Connect Program, either with Red Hat Container Certification or Red Hat OpenShift Operator Certification.

3.15. REMOVING IMAGES

Use the **podman rmi** command to remove locally stored container images. You can remove an image by its ID or name.

Procedure

1. List all images on your local system:

```
$ podman images
REPOSITORY                      TAG     IMAGE ID      CREATED      SIZE
registry.redhat.io/rhel8/rsyslog    latest  4b32d14201de  7 weeks ago  228 MB
registry.redhat.io/ubi8/ubi         latest  3269c37eae33  7 weeks ago  208 MB
localhost/myubi                 X.Y     3269c37eae33  7 weeks ago  208 MB
```

2. List all containers:

```
$ podman ps -a
CONTAINER ID  IMAGE                                COMMAND         CREATED        STATUS
PORTS   NAMES
7ccd6001166e  registry.redhat.io/rhel8/rsyslog:latest  /bin/rsyslog.sh  6 seconds ago  Up 5
seconds ago          mysyslog
```

   To remove the **registry.redhat.io/rhel8/rsyslog** image, you have to stop all containers running from this image using the **podman stop** command. You can stop a container by its ID or name.

3. Stop the **mysyslog** container:

```
$ podman stop mysyslog
7ccd6001166e9720c47fbeb077e0afd0bb635e74a1b0ede3fd34d09eaf5a52e9
```

4. Remove the **registry.redhat.io/rhel8/rsyslog** image:

```
$ podman rmi registry.redhat.io/rhel8/rsyslog
```

   - To remove multiple images:

```
$ podman rmi registry.redhat.io/rhel8/rsyslog registry.redhat.io/ubi8/ubi
```

   - To remove all images from your system:

```
$ podman rmi -a
```

   - To remove images that have multiple names (tags) associated with them, add the **-f** option to remove them:

```
$ podman rmi -f 1de7d7b3f531
1de7d7b3f531...
```

Additional resources

- **podman-rmi** man page

CHAPTER 4. SIGNING CONTAINER IMAGES

You can use a GNU Privacy Guard (GPG) signature or a sigstore signature to sign your container image. Both signing techniques are generally compatible with any OCI compliant container registries. You can use Podman to sign the image before pushing it into a remote registry and configure consumers so that any unsigned image is rejected. Signing container images helps to prevent supply chain attacks.

Signing using GPG keys requires deploying a separate lookaside server to distribute signatures. The lookaside server can be any HTTP server. Starting with Podman version 4.2, you can use the sigstore format of container signatures. Compared to the GPG keys, the separate lookaside server is not required because the sigstore signatures are stored in the container registry.

4.1. SIGNING CONTAINER IMAGES WITH GPG SIGNATURES

You can sign images using a GNU Privacy Guard (GPG) key.

Prerequisites

- The GPG tool is installed.
- The lookaside web server is set up and you can publish files on it.
  - You can check the system-wide registries configuration in the **/etc/containers/registries.d/default.yaml** file. The **lookaside-staging** option references a file path for signature writing and is typically set on hosts publishing signatures.

```
# cat /etc/containers/registries.d/default.yaml
...
docker:
    <registry>:
        lookaside: https://registry-lookaside.example.com
        lookaside-staging: file:///var/lib/containers/sigstore
...
```

Procedure

1. Generate a GPG key:

```
# gpg --full-gen-key
```

2. Export the public key:

```
# gpg --output <path>/key.gpg --armor --export <username>@redhat.com
```

3. Build the container image using **Containerfile** in the current directory:

```
$ podman build -t <registry>/<namespace>/<image>
```

   Replace ***<registry>***, ***<namespace>***, and ***<image>*** with the container image identifiers. For more details, see Container registries.

4. Sign the image and push it to the registry:

```
$ podman push \ --sign-by <username>@redhat.com \
<registry>/<namespace>/<image>
```

NOTE

If you need to sign existing images while moving them across container registries, you can use the **skopeo copy** command.

5. Optional. Display the new image signature:

```
# (cd /var/lib/containers/sigstore/; find . -type f)
./<image>@sha256=<digest>/signature-1
```

6. Copy your local signatures to the lookaside web server:

```
# rsync -a /var/lib/containers/sigstore user@registry-lookaside.example.com:/registry-lookaside/webroot/sigstore
```

The signatures are stored in the location determined by the **lookaside-staging** option, in this case, **/var/lib/containers/sigstore** directory.

Verification

- For more details, see Verifying GPG image signatures.

Additional resources

- **podman-image-trust** man page
- **podman-push** man page
- **podman-build** man page
- How to generate GPG key pairs

4.2. VERIFYING GPG IMAGE SIGNATURES

You can verify that a container image is correctly signed with a GPG key using the following procedure.

Prerequisites

- The web server for a signature reading is set up and you can publish files on it.
  - You can check the system-wide registries configuration in the **/etc/containers/registries.d/default.yaml** file. The **lookaside** option references a web server for signature reading. The **lookaside** option has to be set for verifying signatures.

```
# cat /etc/containers/registries.d/default.yaml
...
docker:
    <registry>:
```

```
        lookaside: https://registry-lookaside.example.com
        lookaside-staging: file:///var/lib/containers/sigstore
...
```

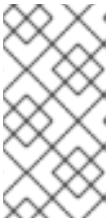
Procedure

1. Update a trust scope for the ***<registry>***:

```
$ podman image trust set -f <path>/key.gpg <registry>/<namespace>
```

2. Optional. Verify the trust policy configuration by displaying the **/etc/containers/policy.json** file:

```
$ cat /etc/containers/policy.json
{
  ...
  "transports": {
    "docker": {
      "<registry>/<namespace>": [
        {
          "type": "signedBy",
          "keyType": "GPGKeys",
          "keyPath": "<path>/key.gpg"
        }
      ]
    }
  }
}
```



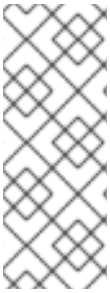
NOTE

Typically, the **/etc/containers.policy.json** file is configured at a level of organization where the same keys are used. For example, ***<registry>/<namespace>*** for a public registry, or just a ***<registry>*** for a single-company dedicated registry.

3. Pull the image:

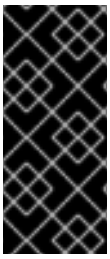
```
# podman pull <registry>/<namespace>/<image>
...
Storing signatures
e7d92cdc71feacf90708cb59182d0df1b911f8ae022d29e8e95d75ca6a99776a
```

The **podman pull** command enforces signature presence as configured, no extra options are required.



NOTE

You can edit the system-wide registry configuration in the **/etc/containers/registries.d/default.yaml** file. You can also edit the registry or repository configuration section in any YAML file in the **/etc/containers/registries.d** directory. All YAML files are read and the filename can be arbitrary. A single scope (default-docker, registry, or namespace) can only exist in one file within the **/etc/containers/registries.d** directory.



IMPORTANT

The system-wide registries configuration in the **/etc/containers/registries.d/default.yaml** file allows accessing the published signatures. The **sigstore** and **sigstore-staging** options are now deprecated. These options refer to signing storage, and they are not connected to the sigstore signature format. Use the new equivalent **lookaside** and **lookaside-staging** options instead.

Additional resources

- **podman-image-trust** man page
- **podman-pull** man page

4.3. SIGNING CONTAINER IMAGES WITH SIGSTORE SIGNATURES

Starting with Podman version 4.2, you can use the sigstore format of container signatures.

Prerequisites

- The public and private key pair exists.

NOTE

The generation of public and private keys is not implemented. You must use the upstream Cosign project to generate a public and private key pair:

1. Install the cosign tool:

```
$ git clone https://github.com/sigstore/cosign
$ cd cosign
$ make ./cosign
```

2. Generate a public and private key pair:

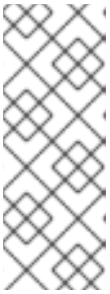
```
$ ./cosign generate-key-pair
...
Private key written to cosign.key
Public key written to cosign.pub
```

Procedure

1. Add the following content to the **/etc/containers/registries.conf.d/default.yaml** file: docker:

```
<registry>:
   use-sigstore-attachments: true
```

By setting the **use-sigstore-attachments** option, Podman and Skopeo can read and write the container sigstore signatures together with the image and save them in the same repository as the signed image.



NOTE

You can edit the system-wide registry configuration in the **/etc/containers/registries.d/default.yaml** file. You can also edit the registry or repository configuration section in any YAML file in the **/etc/containers/registries.d** directory. All YAML files are read and the filename can be arbitrary. A single scope (default-docker, registry, or namespace) can only exist in one file within the **/etc/containers/registries.d** directory.

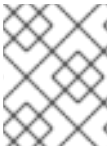
2. Build the container image using **Containerfile** in the current directory:

```
$ podman build -t <registry>/<namespace>/<image>
```

3. Sign the image and push it to the registry:

```
$ podman push --sign-by-sigstore-private-key ./cosign.key
<registry>/<namespace>/image>
```

The **podman push** command pushes the ***<registry>/<namespace>/<image>*** local image to the remote registry as ***<registry>/<namespace>/<image>***. The **--sign-by-sigstore-private-key** option adds a sigstore signature using the **cosign.key** private key to the ***<registry>/<namespace>/<image>*** image. The image and the sigstore signature are uploaded to the remote registry.



NOTE

If you need to sign existing images while moving them across container registries, you can use the **skopeo copy** command.

Verification

- For more details, see Verifying sigstore image signatures.

Additional resources

- **podman-push** man page
- **podman-build** man page
- Sigstore: An open answer to software supply chain trust and security

4.4. VERIFYING SIGSTORE IMAGE SIGNATURES

You can verify that a container image is correctly signed using the following procedure.

Procedure

1. Add the following content to the **/etc/containers/registries.conf.d/default.yaml** file: docker:

```
<registry>:
   use-sigstore-attachments: true
```

By setting the **use-sigstore-attachments** option, Podman and Skopeo can read and write the container sigstore signatures together with the image and save them in the same repository as the signed image.

NOTE

You can edit the system-wide registry configuration in the **/etc/containers/registries.d/default.yaml** file. You can also edit the registry or repository configuration section in any YAML file in the **/etc/containers/registries.d** directory. All YAML files are read and the filename can be arbitrary. A single scope (default-docker, registry, or namespace) can only exist in one file within the **/etc/containers/registries.d** directory.

2. Edit the **/etc/containers/policy.json** file to enforce sigstore signature presence:

```
...
"transports": {
        "docker": {
            "<registry>/<namespace>": [
                {
                    "type": "sigstoreSigned",
                    "keyPath": "/some/path/to/cosign.pub"
                }
            ]
        }
    }
...
```

By modifying the **/etc/containers/policy.json** configuration file, you change the trust policy configuration. Podman, Buildah, and Skopeo enforce the existence of the container image signatures.

3. Pull the image:

```
$ podman pull <registry>/<namespace>/<image>
```

The **podman pull** command enforces signature presence as configured, no extra options are required.

Additional resources

- Sigstore: An open answer to software supply chain trust and security

CHAPTER 5. WORKING WITH CONTAINERS

Containers represent a running or stopped process created from the files located in a decompressed container image. You can use the Podman tool to work with containers.

5.1. PODMAN RUN COMMAND

The **podman run** command runs a process in a new container based on the container image. If the container image is not already loaded then **podman run** pulls the image, and all image dependencies, from the repository in the same way running **podman pull *image***, before it starts the container from that image. The container process has its own file system, its own networking, and its own isolated process tree.

The **podman run** command has the form:

```
podman run [options] image [command [arg ...]]
```

Basic options are:

- **--detach (-d)**: Runs the container in the background and prints the new container ID.
- **--attach (-a)**: Runs the container in the foreground mode.
- **--name (-n)**: Assigns a name to the container. If a name is not assigned to the container with **--name** then it generates a random string name. This works for both background and foreground containers.
- **--rm**: Automatically remove the container when it exits. Note that the container will not be removed when it could not be created or started successfully.
- **--tty (-t)**: Allocates and attaches the pseudo-terminal to the standard input of the container.
- **--interactive (-i)**: For interactive processes, use **-i** and **-t** together to allocate a terminal for the container process. The **-i -t** is often written as **-it**.

5.2. RUNNING COMMANDS IN A CONTAINER FROM THE HOST

This procedure shows how to use the **podman run** command to display the type of operating system of the container.

Prerequisites

- The Podman tool is installed.

```
# yum module install -y container-tools
```

Procedure

1. Display the type of operating system of the container based on the **registry.access.redhat.com/ubi8/ubi** container image using the **cat /etc/os-release** command:

```
$ podman run --rm registry.access.redhat.com/ubi8/ubi cat /etc/os-release
NAME="Red Hat Enterprise Linux"
...
```

```
ID="rhel"
...
HOME_URL="https://www.redhat.com/"
BUG_REPORT_URL="https://bugzilla.redhat.com/"

REDHAT_BUGZILLA_PRODUCT="Red Hat Enterprise Linux 8"
...
```

2. Optional: List all containers.

```
$ podman ps
CONTAINER ID  IMAGE   COMMAND  CREATED  STATUS  PORTS   NAMES
```

Because of the **--rm** option you should not see any container. The container was removed.

Additional resources

- **podman-run** man page

5.3. RUNNING COMMANDS INSIDE THE CONTAINER

This procedure shows how you can use the **podman run** command to run a container interactively.

Prerequisites

- The Podman tool is installed.

```
# yum module install -y container-tools
```

Procedure

1. Run the container named **myubi** based on the **registry.redhat.io/ubi8/ubi** image:

```
$ podman run --name=myubi -it registry.access.redhat.com/ubi8/ubi /bin/bash
[root@6ccffd0f6421 /]#
```

- The **-i** option creates an interactive session. Without the **-t** option, the shell stays open, but you cannot type anything to the shell.
- The **-t** option opens a terminal session. Without the **-i** option, the shell opens and then exits.

2. Install the **procps-ng** package containing a set of system utilities (for example **ps**, **top**, **uptime**, and so on):

```
[root@6ccffd0f6421 /]# yum install procps-ng
```

3. Use the **ps -ef** command to list current processes:

```
# ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 12:55 pts/0    00:00:00 /bin/bash
root          31       1  0 13:07 pts/0    00:00:00 ps -ef
```

4. Enter **exit** to exit the container and return to the host:

```
# exit
```

5. Optional: List all containers:

```
$ podman ps
CONTAINER ID  IMAGE                            COMMAND    CREATED         STATUS
PORTS   NAMES
1984555a2c27  registry.redhat.io/ubi8/ubi:latest  /bin/bash  21 minutes ago  Exited (0) 21
minutes ago          myubi
```

You can see that the container is in Exited status.

Additional resources

- **podman-run** man page

5.4. LISTING CONTAINERS

Use the **podman ps** command to list the running containers on the system.

Prerequisites

- The Podman tool is installed.

```
#  yum module install -y container-tools
```

Procedure

1. Run the container based on **registry.redhat.io/rhel8/rsyslog** image:

```
$ podman run -d registry.redhat.io/rhel8/rsyslog
```

2. List all containers:
   - To list all running containers:

```
$ podman ps
CONTAINER ID IMAGE              COMMAND        CREATED       STATUS
PORTS NAMES
74b1da000a11 rhel8/rsyslog /bin/rsyslog.sh 2 minutes ago Up About a minute
musing_brown
```

   - To list all containers, running or stopped:

```
$ podman ps -a
CONTAINER ID IMAGE         COMMAND    CREATED    STATUS              PORTS
NAMES     IS INFRA
d65aecc325a4 ubi8/ubi      /bin/bash  3 secs ago Exited (0) 5 secs ago peaceful_hopper
false
74b1da000a11 rhel8/rsyslog rsyslog.sh 2 mins ago Up About a minute     musing_brown
false
```

If there are containers that are not running, but were not removed (**--rm** option), the containers are present and can be restarted.

Additional resources

- **podman-ps** man page

5.5. STARTING CONTAINERS

If you run the container and then stop it, and not remove it, the container is stored on your local system ready to run again. You can use the **podman start** command to re-run the containers. You can specify the containers by their container ID or name.

Prerequisites

- The Podman tool is installed.

```
# yum module install -y container-tools
```

- At least one container has been stopped.

Procedure

1. Start the **myubi** container:
   - In the non interactive mode:

```
$ podman start myubi
```

     Alternatively, you can use **podman start 1984555a2c27**.

   - In the interactive mode, use **-a** (**--attach**) and **-t** (**--interactive**) options to work with container bash shell:

```
$ podman start -a -i myubi
```

     Alternatively, you can use **podman start -a -i 1984555a2c27**.

2. Enter **exit** to exit the container and return to the host:

```
[root@6ccffd0f6421 /]# exit
```

Additional resources

- **podman-start** man page

5.6. INSPECTING CONTAINERS FROM THE HOST

Use the **podman inspect** command to inspect the metadata of an existing container in a JSON format. You can specify the containers by their container ID or name.

Prerequisites

- The Podman tool is installed.

```
# yum module install -y container-tools
```

Procedure

- Inspect the container defined by ID 64ad95327c74:
  - To get all metadata:

```
$ podman inspect 64ad95327c74
[
    {
        "Id":
"64ad95327c740ad9de468d551c50b6d906344027a0e645927256cd061049f681",
        "Created": "2021-03-02T11:23:54.591685515+01:00",
        "Path": "/bin/rsyslog.sh",
        "Args": [
            "/bin/rsyslog.sh"
        ],
        "State": {
            "OciVersion": "1.0.2-dev",
            "Status": "running",
            ...
```

  - To get particular items from the JSON file, for example, the **StartedAt** timestamp:

```
$ podman inspect --format='{{.State.StartedAt}}' 64ad95327c74
2021-03-02 11:23:54.945071961 +0100 CET
```

    The information is stored in a hierarchy. To see the container **StartedAt** timestamp (**StartedAt** is under **State**), use the **--format** option and the container ID or name.

Examples of other items you might want to inspect include:

- **.Path** to see the command run with the container
- **.Args** arguments to the command
- **.Config.ExposedPorts** TCP or UDP ports exposed from the container
- **.State.Pid** to see the process id of the container
- **.HostConfig.PortBindings** port mapping from container to host

Additional resources

- **podman-inspect** man page

5.7. MOUNTING DIRECTORY ON LOCALHOST TO THE CONTAINER

This procedure shows how you can make log messages from inside a container available to the host system by mounting the host **/dev/log** device inside the container.

Prerequisites

- The Podman tool is installed.

```
# yum module install -y container-tools
```

Procedure

1. Run the container named **log_test** and mount the host **/dev/log** device inside the container:

```
# podman run --name="log_test" -v /dev/log:/dev/log --rm \ registry.redhat.io/ubi8/ubi
logger "Testing logging to the host"
```

2. Use the **journalctl** utility to display logs:

```
# journalctl -b | grep Testing
Dec 09 16:55:00 localhost.localdomain root[14634]: Testing logging to the host
```

The **--rm** option removes the container when it exits.

Additional resources

- **podman-run** man page

5.8. MOUNTING A CONTAINER FILESYSTEM

Use the **podman mount** command to mount a working container root filesystem in a location accessible from the host.

Prerequisites

- The Podman tool is installed.

```
# yum module install -y container-tools
```

Procedure

1. Run the container named **mysyslog**:

```
# podman run -d --name=mysyslog registry.redhat.io/rhel8/rsyslog
```

2. Optional: List all containers:

```
# podman ps -a
CONTAINER ID  IMAGE                                COMMAND         CREATED         STATUS
PORTS   NAMES
c56ef6a256f8  registry.redhat.io/rhel8/rsyslog:latest  /bin/rsyslog.sh  20 minutes ago  Up 20
minutes ago                mysyslog
```

3. Mount the **mysyslog** container:

```
# podman mount mysyslog
/var/lib/containers/storage/overlay/990b5c6ddcdeed4bde7b245885ce4544c553d108310e2b797d7be46750894719/merged
```

4. Display the content of the mount point using **ls** command:

```
# ls
/var/lib/containers/storage/overlay/990b5c6ddcdeed4bde7b245885ce4544c553d108310e2b797d7be46750894719/merged
bin boot dev etc home lib lib64 lost+found media mnt opt proc root run sbin srv sys tmp usr var
```

5. Display the OS version:

```
# cat
/var/lib/containers/storage/overlay/990b5c6ddcdeed4bde7b245885ce4544c553d108310e2b797d7be46750894719/merged/etc/os-release
NAME="Red Hat Enterprise Linux"
VERSION="8 (Ootpa)"
ID="rhel"
ID_LIKE="fedora"
...
```

Additional resources

- **podman-mount** man page

5.9. RUNNING A SERVICE AS A DAEMON WITH A STATIC IP

The following example runs the **rsyslog** service as a daemon process in the background. The **--ip** option sets the container network interface to a particular IP address (for example, 10.88.0.44). After that, you can run the **podman inspect** command to check that you set the IP address properly.

Prerequisites

- The Podman tool is installed.

```
# yum module install -y container-tools
```

Procedure

1. Set the container network interface to the IP address 10.88.0.44:

```
# podman run -d --ip=10.88.0.44 registry.access.redhat.com/rhel8/rsyslog
efde5f0a8c723f70dd5cb5dc3d5039df3b962fae65575b08662e0d5b5f9fbe85
```

2. Check that the IP address is set properly:

```
# podman inspect efde5f0a8c723 | grep 10.88.0.44
"IPAddress": "10.88.0.44",
```

Additional resources

- **podman-inspect** man page
- **podman-run** man page

5.10. EXECUTING COMMANDS INSIDE A RUNNING CONTAINER

Use the **podman exec** command to execute a command in a running container and investigate that container. The reason for using the **podman exec** command instead of **podman run** command is that you can investigate the running container without interrupting the container activity.

Prerequisites

- The Podman tool is installed.

```
# yum module install -y container-tools
```

- The container is running.

Procedure

1. Execute the **rpm -qa** command inside the **myrsyslog** container to list all installed packages:

```
$ podman exec -it myrsyslog rpm -qa
tzdata-2020d-1.el8.noarch
python3-pip-wheel-9.0.3-18.el8.noarch
redhat-release-8.3-1.0.el8.x86_64
filesystem-3.8-3.el8.x86_64
...
```

2. Execute a **/bin/bash** command in the **myrsyslog** container:

```
$ podman exec -it myrsyslog /bin/bash
```

3. Install the **procps-ng** package containing a set of system utilities (for example **ps**, **top**, **uptime**, and so on):

```
# yum install procps-ng
```

4. Inspect the container:
   - To list every process on the system:

```
# ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 10:23 ?        00:00:01 /usr/sbin/rsyslogd -n
root           8       0  0 11:07 pts/0    00:00:00 /bin/bash
root          47       8  0 11:13 pts/0    00:00:00 ps -ef
```

   - To display file system disk space usage:

```
# df -h
Filesystem      Size  Used Avail Use% Mounted on
fuse-overlayfs   27G  7.1G   20G  27% /
```

```
tmpfs           64M     0   64M   0% /dev
tmpfs          269M  936K  268M   1% /etc/hosts
shm             63M     0   63M   0% /dev/shm
...
```

- To display system information:

```
# uname -r
4.18.0-240.10.1.el8_3.x86_64
```

- To display amount of free and used memory in megabytes:

```
# free --mega
total        used        free      shared  buff/cache   available
Mem:         2818         615        1183          12        1020        1957
Swap:        3124           0        3124
```

Additional resources

- **podman-exec** man page

5.11. SHARING FILES BETWEEN TWO CONTAINERS

You can use volumes to persist data in containers even when a container is deleted. Volumes can be used for sharing data among multiple containers. The volume is a folder which is stored on the host machine. The volume can be shared between the container and the host.

Main advantages are:

- Volumes can be shared among the containers.
- Volumes are easier to back up or migrate.
- Volumes do not increase the size of the containers.

Prerequisites

- The Podman tool is installed.

```
# yum module install -y container-tools
```

Procedure

1. Create a volume:

```
$ podman volume create hostvolume
```

2. Display information about the volume:

```
$ podman volume inspect hostvolume
[
    {
        "name": "hostvolume",
```

```
        "labels": {},
        "mountpoint":
"/home/username/.local/share/containers/storage/volumes/hostvolume/_data",
        "driver": "local",
        "options": {},
        "scope": "local"
    }
]
```

Notice that it creates a volume in the volumes directory. You can save the mount point path to the variable for easier manipulation: **$ mntPoint=$(podman volume inspect hostvolume --format {{.Mountpoint}})**.

Notice that if you run **sudo podman volume create hostvolume**, then the mount point changes to **/var/lib/containers/storage/volumes/hostvolume/_data**.

3. Create a text file inside the directory using the path that is stored in the **mntPoint** variable:

```
$ echo "Hello from host" >> $mntPoint/host.txt
```

4. List all files in the directory defined by the **mntPoint** variable:

```
$ ls $mntPoint/
host.txt
```

5. Run the container named **myubi1** and map the directory defined by the **hostvolume** volume name on the host to the **/containervolume1** directory on the container:

```
$ podman run -it --name myubi1 -v hostvolume:/containervolume1
registry.access.redhat.com/ubi8/ubi /bin/bash
```

Note that if you use the volume path defined by the **mntPoint** variable (**-v $mntPoint:/containervolume1**), data can be lost when running **podman volume prune** command, which removes unused volumes. Always use **-v *hostvolume_name:/containervolume_name***.

6. List the files in the shared volume on the container:

```
# ls /containervolume1
host.txt
```

You can see the **host.txt** file which you created on the host.

7. Create a text file inside the **/containervolume1** directory:

```
# echo "Hello from container 1" >> /containervolume1/container1.txt
```

8. Detach from the container with **CTRL+p** and **CTRL+q**.

9. List the files in the shared volume on the host, you should see two files:

```
$ ls $mntPoint
container1.rxt  host.txt
```

At this point, you are sharing files between the container and host. To share files between two containers, run another container named **myubi2**.

10. Run the container named **myubi2** and map the directory defined by the **hostvolume** volume name on the host to the /**containervolume2** directory on the container:

```
$ podman run -it --name myubi2 -v hostvolume:/containervolume2
registry.access.redhat.com/ubi8/ubi /bin/bash
```

11. List the files in the shared volume on the container:

```
# ls /containervolume2
container1.txt host.txt
```

You can see the **host.txt** file which you created on the host and **container1.txt** which you created inside the **myubi1** container.

12. Create a text file inside the /**containervolume2** directory:

```
# echo "Hello from container 2" >> /containervolume2/container2.txt
```

13. Detach from the container with **CTRL+p** and **CTRL+q**.

14. List the files in the shared volume on the host, you should see three files:

```
$ ls $mntPoint
container1.rxt  container2.txt host.txt
```

Additional resources

- **podman-volume** man page

5.12. EXPORTING AND IMPORTING CONTAINERS

You can use the **podman export** command to export the file system of a running container to a tarball on your local machine. For example, if you have a large container that you use infrequently or one that you want to save a snapshot of in order to revert back to it later, you can use the **podman export** command to export a current snapshot of your running container into a tarball.

You can use the **podman import** command to import a tarball and save it as a filesystem image. Then you can run this filesystem image or you can use it as a layer for other images.

Prerequisites

- The Podman tool is installed.

```
#  yum module install -y container-tools
```

Procedure

1. Run the **myubi** container based on the **registry.access.redhat.com/ubi8/ubi** image:

```
$ podman run -dt --name=myubi registry.access.redhat.com/8/ubi
```

2. Optional: List all containers:

```
$ podman ps -a
CONTAINER ID  IMAGE                                 COMMAND         CREATED     STATUS
PORTS   NAMES
a6a6d4896142  registry.access.redhat.com/8:latest   /bin/bash       7 seconds ago  Up 7
seconds ago          myubi
```

3. Attach to the **myubi** container:

```
$ podman attach myubi
```

4. Create a file named **testfile**:

```
[root@a6a6d4896142 /]# echo "hello" > testfile
```

5. Detach from the container with **CTRL+p** and **CTRL+q**.

6. Export the file system of the **myubi** as a **myubi-container.tar** on the local machine:

```
$ podman export -o myubi.tar a6a6d4896142
```

7. Optional: List the current directory content:

```
$ ls -l
-rw-r--r--. 1 user user 210885120 Apr  6 10:50 myubi-container.tar
...
```

8. Optional: Create a **myubi-container** directory, extract all files from the **myubi-container.tar** archive. List a content of the **myubi-directory** in a tree-like format:

```
$ mkdir myubi-container
$ tar -xf myubi-container.tar -C myubi-container
$ tree -L 1 myubi-container
├── bin -> usr/bin
├── boot
├── dev
├── etc
├── home
├── lib -> usr/lib
├── lib64 -> usr/lib64
├── lost+found
├── media
├── mnt
├── opt
├── proc
├── root
├── run
├── sbin -> usr/sbin
├── srv
├── sys
├── testfile
├── tmp
├── usr
```

```
└── var

20 directories, 1 file
```

You can see that the **myubi-container.tar** contains the container file system.

9. Import the **myubi.tar** and saves it as a filesystem image:

```
$ podman import myubi.tar myubi-imported
Getting image source signatures
Copying blob 277cab30fe96 done
Copying config c296689a17 done
Writing manifest to image destination
Storing signatures
c296689a17da2f33bf9d16071911636d7ce4d63f329741db679c3f41537e7cbf
```

10. List all images:

```
$ podman images
REPOSITORY                          TAG     IMAGE ID      CREATED         SIZE
docker.io/library/myubi-imported    latest  c296689a17da  51 seconds ago  211 MB
```

11. Display the content of the **testfile** file:

```
$ podman run -it --name=myubi-imported docker.io/library/myubi-imported cat testfile
hello
```

Additional resources

- **podman-export** man page
- **podman-import** man page

5.13. STOPPING CONTAINERS

Use the **podman stop** command to stop a running container. You can specify the containers by their container ID or name.

Prerequisites

- The Podman tool is installed.

```
# yum module install -y container-tools
```

- At least one container is running.

Procedure

- Stop the **myubi** container:
  - Using the container name:

```
$ podman stop myubi
```

- Using the container ID:

```
$ podman stop 1984555a2c27
```

To stop a running container that is attached to a terminal session, you can enter the **exit** command inside the container.

The **podman stop** command sends a SIGTERM signal to terminate a running container. If the container does not stop after a defined period (10 seconds by default), Podman sends a SIGKILL signal.

You can also use the **podman kill** command to kill a container (SIGKILL) or send a different signal to a container. Here is an example of sending a SIGHUP signal to a container (if supported by the application, a SIGHUP causes the application to re-read its configuration files):

```
# *podman kill --signal="SIGHUP" 74b1da000a11*
74b1da000a114015886c557deec8bed9dfb80c888097aa83f30ca4074ff55fb2
```

Additional resources

- **podman-stop** man page
- **podman-kill** man page

5.14. REMOVING CONTAINERS

Use the **podman rm** command to remove containers. You can specify containers with the container ID or name.

Prerequisites

- The Podman tool is installed.

```
# yum module install -y container-tools
```

- At least one container has been stopped.

Procedure

1. List all containers, running or stopped:

```
$ podman ps -a
CONTAINER ID IMAGE        COMMAND    CREATED    STATUS                PORTS NAMES
IS INFRA
d65aecc325a4 ubi8/ubi      /bin/bash  3 secs ago Exited (0) 5 secs ago peaceful_hopper false
74b1da000a11 rhel8/rsyslog rsyslog.sh 2 mins ago Up About a minute     musing_brown
false
```

2. Remove the containers:

- To remove the **peaceful_hopper** container:

```
$ podman rm peaceful_hopper
```

Notice that the **peaceful_hopper** container was in Exited status, which means it was stopped and it can be removed immediately.

- To remove the **musing_brown** container, first stop the container and then remove it:

```
$ podman stop musing_brown
$ podman rm musing_brown
```

NOTE

- To remove multiple containers:

```
$ podman rm clever_yonath furious_shockley
```

- To remove all containers from your local system:

```
$ podman rm -a
```

Additional resources

- **podman-rm** man page

5.15. THE RUNC CONTAINER RUNTIME

The runc container runtime is a lightweight, portable implementation of the Open Container Initiative (OCI) container runtime specification. The runc runtime shares a lot of low-level code with Docker but it is not dependent on any of the components of the Docker platform. The runc supports Linux namespaces, live migration, and has portable performance profiles.

It also provides full support for Linux security features such as SELinux, control groups (cgroups), seccomp, and others. You can build and run images with runc, or you can run OCI-compatible images with runc.

5.16. THE CRUN CONTAINER RUNTIME

The crun is a fast and low-memory footprint OCI container runtime written in C. The crun binary is up to 50 times smaller and up to twice as fast as the runc binary. Using crun, you can also set a minimal number of processes when running your container. The crun runtime also supports OCI hooks.

Additional features of crun include:

- Sharing files by group for rootless containers
- Controlling the stdout and stderr of OCI hooks
- Running older versions of systemd on cgroup v2
- A C library that is used by other programs
- Extensibility
- Portability

Additional resources

- An introduction to crun, a fast and low-memory footprint container runtime

5.17. RUNNING CONTAINERS WITH RUNC AND CRUN

With runc or crun, containers are configured using bundles. A bundle for a container is a directory that includes a specification file named **config.json** and a root filesystem. The root filesystem contains the contents of the container.

NOTE

The ***<runtime>*** can be crun or runc.

Procedure

1. Pull the **registry.access.redhat.com/ubi8/ubi** container image:

```
# podman pull registry.access.redhat.com/ubi8/ubi
```

2. Export the **registry.access.redhat.com/ubi8/ubi** image to the **rhel.tar** archive:

```
# podman export $(podman create registry.access.redhat.com/ubi8/ubi) > rhel.tar
```

3. Create the **bundle/rootfs** directory:

```
# mkdir -p bundle/rootfs
```

4. Extract the **rhel.tar** archive into the **bundle/rootfs** directory:

```
# tar -C bundle/rootfs -xf rhel.tar
```

5. Create a new specification file named **config.json** for the bundle:

```
# <runtime> spec -b bundle
```

   - The **-b** option specifies the bundle directory. The default value is the current directory.

6. Optional. Change the settings:

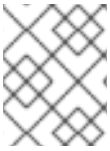
```
# vi bundle/config.json
```

7. Create an instance of a container named **myubi** for a bundle:

```
# <runtime> create -b bundle/ myubi
```

8. Start a **myubi** container:

```
# <runtime> start myubi
```



NOTE

The name of a container instance must be unique to the host. To start a new instance of a container: **# <runtime> start <container_name>**

Verification

- List containers started by ***<runtime>***:

```
# <runtime> list
ID          PID         STATUS      BUNDLE         CREATED                          OWNER
myubi       0           stopped     /root/bundle   2021-09-14T09:52:26.659714605Z   root
```

Additional resources

- **crun** man page
- **runc** man page
- An introduction to crun, a fast and low-memory footprint container runtime

5.18. TEMPORARILY CHANGING THE CONTAINER RUNTIME

You can use the **podman run** command with the **--runtime** option to change the container runtime.

NOTE

The ***<runtime>*** can be crun or runc.

Prerequisites

- The Podman tool is installed.

```
#  yum module install -y container-tools
```

Procedure

- Pull the **registry.access.redhat.com/ubi8/ubi** container image:

```
$ podman pull registry.access.redhat.com/ubi8/ubi
```

1. Change the container runtime using the **--runtime** option:

```
$ podman run --name=myubi -dt --runtime=<runtime> ubi8
bashe4654eb4df12ac031f1d0f2657dc4ae6ff8eb0085bf114623b66cc664072e69b
```

2. Optional. List all images:

```
$ podman ps -a
CONTAINER ID  IMAGE                                COMMAND  CREATED        STATUS
PORTS   NAMES
e4654eb4df12  registry.access.redhat.com/ubi8:latest  bash     4 seconds ago  Up 4
seconds ago          myubi
```

–

Verification

- Ensure that the OCI runtime is set to ***<runtime>*** in the myubi container:

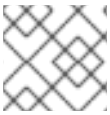
```
$ podman inspect myubi --format "{{.OCIRuntime}}"
<runtime>
```

Additional resources

- An introduction to crun, a fast and low-memory footprint container runtime

5.19. PERMANENTLY CHANGING THE CONTAINER RUNTIME

You can set the container runtime and its options in the **/etc/containers/containers.conf** configuration file as a root user or in the **$HOME/.config/containers/containers.conf** configuration file as a non-root user.



NOTE

The ***<runtime>*** can be crun or runc runtime.

Prerequisites

- The Podman tool is installed.

```
#  yum module install -y container-tools
```

Procedure

- Change the runtime in the **/etc/containers/containers.conf** file:

```
# vim /etc/containers/containers.conf
[engine]
runtime = "<runtime>"
```

- Run the container named myubi:

```
# podman run --name=myubi -dt ubi8 bash
Resolved "ubi8" as an alias (/etc/containers/registries.conf.d/001-rhel-shortnames.conf)
Trying to pull registry.access.redhat.com/ubi8:latest…
...
Storing signatures
```

Verification

- Ensure that the OCI runtime is set to ***<runtime>*** in the **myubi** container:

```
# podman inspect myubi --format "{{.OCIRuntime}}"
<runtime>
```

Additional resources

- An introduction to crun, a fast and low-memory footprint container runtime
- **containers.conf** man page

5.20. CREATING SELINUX POLICIES FOR CONTAINERS

To generate SELinux policies for containers, use the UDICA tool. For more information, see Introduction to the udica SELinux policy generator.

CHAPTER 6. ADDING SOFTWARE TO A UBI CONTAINER

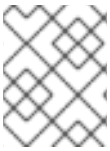
Red Hat Universal Base Images (UBIs) are built from a subset of the RHEL content. UBIs also provide a subset of RHEL packages that are freely available to install for use with UBI. To add or update software to a running container, you can use the yum repositories that include RPM packages and updates. UBIs provide a set of pre-built language runtime container images such as Python, Perl, Node.js, Ruby, and so on.

To add packages from UBI repositories to running UBI containers:

- On UBI init and UBI standard images, use the **yum** command
- On UBI minimal images, use the **microdnf** command

NOTE

Installing and working with software packages directly in running containers adds packages temporarily. The changes are not saved in the container image. To make package changes persistent, see section Building an image from a Containerfile with Buildah.



NOTE

When you add software to a UBI container, procedures differ for updating UBIs on a subscribed RHEL host or on an unsubscribed (or non-RHEL) system.

6.1. USING THE UBI INIT IMAGES

This procedure shows how to build a container using a **Containerfile** that installs and configures a Web server (**httpd**) to start automatically by the systemd service (**/sbin/init**) when the container is run on a host system. The **podman build** command builds an image using instructions in one or more **Containerfiles** and a specified build context directory. The context directory can be specified as the URL of an archive, Git repository or **Containerfile**. If no context directory is specified, then the current working directory is considered as the build context, and must contain the **Containerfile**. You can also specify a **Containerfile** with the **--file** option.

Procedure

1. Create a **Containerfile** with the following contents to a new directory:

```
FROM registry.access.redhat.com/ubi8/ubi-init RUN yum -y install httpd; yum clean all; systemctl enable httpd; RUN echo "Successful Web Server Test" > /var/www/html/index.html RUN mkdir /etc/systemd/system/httpd.service.d/; echo -e '[Service]\nRestart=always' > /etc/systemd/system/httpd.service.d/httpd.conf EXPOSE 80 CMD [ "/sbin/init" ]
```

   The **Containerfile** installs the **httpd** package, enables the **httpd** service to start at boot time, creates a test file (**index.html**), exposes the Web server to the host (port 80), and starts the systemd init service (**/sbin/init**) when the container starts.

2. Build the container:

```
# podman build --format=docker -t mysysd .
```

3. Optional. If you want to run containers with systemd and SELinux is enabled on your system, you must set the **container_manage_cgroup** boolean variable:

```
# setsebool -P container_manage_cgroup 1
```

4. Run the container named **mysysd_run**:

```
# podman run -d --name=mysysd_run -p 80:80 mysysd
```

The **mysysd** image runs as the **mysysd_run** container as a daemon process, with port 80 from the container exposed to port 80 on the host system.

NOTE

In rootless mode, you have to choose host port number >= 1024. For example:

```
$ podman run -d --name=mysysd -p 8081:80 mysysd
```

To use port numbers < 1024, you have to modify the **net.ipv4.ip_unprivileged_port_start** variable:

```
# sysctl net.ipv4.ip_unprivileged_port_start=80
```

5. Check that the container is running:

```
# podman ps
a282b0c2ad3d  localhost/mysysd:latest  /sbin/init  15 seconds ago  Up 14 seconds ago
0.0.0.0:80->80/tcp  mysysd_run
```

6. Test the web server:

```
# curl localhost/index.html
Successful Web Server Test
```

Additional resources

- Shortcomings of Rootless Podman

6.2. USING THE UBI MICRO IMAGES

This procedure shows how to build a **ubi-micro** container image using the Buildah tool.

Prerequisites

- The **container-tools** module is installed.

```
# yum module install -y container-tools
```

Procedure

1. Pull and build the **registry.access.redhat.com/ubi8/ubi-micro** image:

```
# microcontainer=$(buildah from registry.access.redhat.com/ubi8/ubi-micro)
```

2. Mount a working container root filesystem:

```
# micromount=$(buildah mount $microcontainer)
```

3. Install the **httpd** service to the **micromount** directory:

```
# yum install \ --installroot $micromount \ --releasever 8 \ --setopt
install_weak_deps=false \ --nodocs -y \ httpd
# yum clean all \ --installroot $micromount
```

4. Unmount the root file system on the working container:

```
# buildah umount $microcontainer
```

5. Create the **ubi-micro-httpd** image from a working container:

```
# buildah commit $microcontainer ubi-micro-httpd
```

Verification steps

1. Display details about the **ubi-micro-httpd** image:

```
# podman images ubi-micro-httpd
localhost/ubi-micro-httpd latest 7c557e7fbe9f  22 minutes ago  151 MB
```

6.3. ADDING SOFTWARE TO A UBI CONTAINER ON A SUBSCRIBED HOST

If you are running a UBI container on a registered and subscribed RHEL host, the RHEL Base and AppStream repositories are enabled inside the standard UBI container, along with all the UBI repositories.

Additional resources

- Universal Base Images (UBI): Images, repositories, packages, and source code

6.4. ADDING SOFTWARE IN A STANDARD UBI CONTAINER

To add software inside the standard UBI container, disable non-UBI yum repositories to ensure the containers you build can be redistributed.

Procedure

1. Pull and run the **registry.access.redhat.com/ubi8/ubi** image:

```
$ podman run -it --name myubi registry.access.redhat.com/ubi8/ubi
```

2. Add a package to the **myubi** container.

   - To add a package that is in the UBI repository, disable all yum repositories except for UBI

- To add a package that is in the UBI repository, disable all yum repositories except for UBI repositories. For example, to add the **bzip2** package:

```
# yum install --disablerepo= --enablerepo=ubi-8-appstream-rpms --enablerepo=ubi-8-baseos-rpms bzip2*
```

- To add a package that is not in the UBI repository, do not disable any repositories. For example, to add the **zsh** package:

```
# yum install zsh
```

- To add a package that is in a different host repository, explicitly enable the repository you need. For example, to install the **python38-devel** package from the **codeready-builder-for-rhel-8-x86_64-rpms** repository:

```
# yum install --enablerepo=codeready-builder-for-rhel-8-x86_64-rpms python38-devel
```

Verification steps

1. List all enabled repositories inside the container:

```
# yum repolist
```

2. Ensure that the required repositories are listed.
3. List all installed packages:

```
# rpm -qa
```

4. Ensure that the required packages are listed.

NOTE

Installing Red Hat packages that are not inside the Red Hat UBI repositories can limit the ability to distribute the container outside of subscribed RHEL systems.

6.5. ADDING SOFTWARE IN A MINIMAL UBI CONTAINER

UBI yum repositories are enabled inside UBI Minimal images by default.

Procedure

1. Pull and run the **registry.access.redhat.com/ubi8/ubi-minimal** image:

```
$ podman run -it --name myubimin registry.access.redhat.com/ubi8/ubi-minimal
```

2. Add a package to the **myubimin** container:

- To add a package that is in the UBI repository, do not disable any repositories. For example, to add the **bzip2** package:

```
# microdnf install bzip2
```

-

- To add a package that is in a different host repository, explicitly enable the repository you need. For example, to install the **python38-devel** package from the **codeready-builder-for-rhel-8-x86_64-rpms** repository:

```
# microdnf install --enablerepo=codeready-builder-for-rhel-8-x86_64-rpms
python38-devel
```

Verification steps

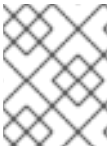
1. List all enabled repositories inside the container:

```
# microdnf repolist
```

2. Ensure that the required repositories are listed.
3. List all installed packages:

```
# rpm -qa
```

4. Ensure that the required packages are listed.



NOTE

Installing Red Hat packages that are not inside the Red Hat UBI repositories can limit the ability to distribute the container outside of subscribed RHEL systems.

6.6. ADDING SOFTWARE TO A UBI CONTAINER ON A UNSUBSCRIBED HOST

You do not have to disable any repositories when adding software packages on unsubscribed RHEL systems.

Procedure

- Add a package to a running container based on the UBI standard or UBI init images. Do not disable any repositories. Use the **podman run** command to run the container. then use the **yum install** command inside a container.
  - For example, to add the **bzip2** package to the UBI standard based container:

```
$ podman run -it --name myubi registry.access.redhat.com/ubi8/ubi
# yum install bzip2
```

  - For example, to add the **bzip2** package to the UBI init based container:

```
$ podman run -it --name myubimin registry.access.redhat.com/ubi8/ubi-minimal
# microdnf install bzip2
```

Verification steps

1. List all enabled repositories:

- To list all enabled repositories inside the containers based on UBI standard or UBI init images:

```
# yum repolist
```

- To list all enabled repositories inside the containers based on UBI minimal containers:

```
# microdnf repolist
```

2. Ensure that the required repositories are listed.
3. List all installed packages:

```
# rpm -qa
```

4. Ensure that the required packages are listed.

6.7. BUILDING UBI-BASED IMAGES

You can create a UBI-based web server container from a **Containerfile** using the Buildah utility. You have to disable all non-UBI yum repositories to ensure that your image contains only Red Hat software that you can redistribute.

NOTE

For UBI minimal images, use `microdnf` instead of `{PackageManagerCommand}`:
[literal,subs="+quotes,attributes",options="nowrap",role=white-space-pre]

`RUN microdnf update -y && rm -rf /var/cache/yum` and
`RUN microdnf install httpd -y && microdnf clean all` commands.

Procedure

1. Create a **Containerfile**:

```
FROM registry.access.redhat.com/ubi8/ubi
USER root
LABEL maintainer="John Doe"
# Update image
RUN yum update --disablerepo=* --enablerepo=ubi-8-appstream-rpms --enablerepo=ubi-8-baseos-rpms -y && rm -rf /var/cache/yum
RUN yum install --disablerepo=* --enablerepo=ubi-8-appstream-rpms --enablerepo=ubi-8-baseos-rpms httpd -y && rm -rf /var/cache/yum
# Add default Web page and expose port
RUN echo "The Web Server is Running" > /var/www/html/index.html
EXPOSE 80
# Start the service
CMD ["-D", "FOREGROUND"]
ENTRYPOINT ["/usr/sbin/httpd"]
```

2. Build the container image:

```
# buildah bud -t johndoe/webserver .
STEP 1: FROM registry.access.redhat.com/ubi8/ubi:latest
STEP 2: USER root
STEP 3: LABEL maintainer="John Doe"
STEP 4: RUN yum update --disablerepo=* --enablerepo=ubi-8-appstream-rpms --
enablerepo=ubi-8-baseos-rpms -y
...
Writing manifest to image destination
Storing signatures
--> f9874f27050
f9874f270500c255b950e751e53d37c6f8f6dba13425d42f30c2a8ef26b769f2
```

Verification steps

1. Run the web server:

```
# podman run -d --name=myweb -p 80:80 johndoe/webserver
bbe98c71d18720d966e4567949888dc4fb86eec7d304e785d5177168a5965f64
```

2. Test the web server:

```
# curl http://localhost/index.html
The Web Server is Running
```

6.8. USING APPLICATION STREAM RUNTIME IMAGES

Runtime images based on Application Streams offer a set of container images that you can use as the basis for your container builds.

Supported runtime images are Python, Ruby, s2-core, s2i-base, .NET Core, PHP. The runtime images are available in the Red Hat Container Catalog.

NOTE

Because these UBI images contain the same basic software as their legacy image counterparts, you can learn about those images from the Using Red Hat Software Collections Container Images guide .

Additional resources

- Red Hat Container Catalog
- Red Hat Container Image Updates

6.9. GETTING UBI CONTAINER IMAGE SOURCE CODE

Source code is available for all Red Hat UBI-based images in the form of downloadable container images. Source container images cannot be run, despite being packaged as containers. To install Red Hat source container images on your system, use the **skopeo** command, not the **podman pull** command.

Source container images are named based on the binary containers they represent. For example, for a particular standard RHEL UBI 8 container **registry.access.redhat.com/ubi8:8.1-397** append **-source** to get the source container image (**registry.access.redhat.com/ubi8:8.1-397-source**).

Procedure

1. Use the **skopeo copy** command to copy the source container image to a local directory:

```
$ skopeo copy \ docker://registry.access.redhat.com/ubi8:8.1-397-source \
dir:$HOME/TEST
...
Copying blob 477bc8106765 done
Copying blob c438818481d3 done
...
Writing manifest to image destination
Storing signatures
```

2. Use the **skopeo inspect** command to inspect the source container image:

```
$ skopeo inspect dir:$HOME/TEST
{
    "Digest":
"sha256:7ab721ef3305271bbb629a6db065c59bbeb87bc53e7cbf88e2953a1217ba7322",
    "RepoTags": [],
    "Created": "2020-02-11T12:14:18.612461174Z",
    "DockerVersion": "",
    "Labels": null,
    "Architecture": "amd64",
    "Os": "linux",
    "Layers": [
        "sha256:1ae73d938ab9f11718d0f6a4148eb07d38ac1c0a70b1d03e751de8bf3c2c87fa",
        "sha256:9fe966885cb8712c47efe5ecc2eaa0797a0d5ffb8b119c4bd4b400cc9e255421",
        "sha256:61b2527a4b836a4efbb82dfd449c0556c0f769570a6c02e112f88f8bbcd90166",
        ...
        "sha256:cc56c782b513e2bdd2cc2af77b69e13df4ab624ddb856c4d086206b46b9b9e5f",
        "sha256:dcf9396fdada4e6c1ce667b306b7f08a83c9e6b39d0955c481b8ea5b2a465b32",

"sha256:feb6d2ae252402ea6a6fca8a158a7d32c7e4572db0e6e5a5eab15d4e0777951e"
    ],
    "Env": null
}
```

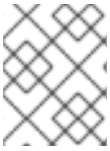
3. Unpack all the content:

```
$ cd $HOME/TEST
$ for f in $(ls); do tar xvf $f; done
```

4. Check the results:

```
$ find blobs/ rpm_dir/
blobs/
blobs/sha256
blobs/sha256/10914f1fff060ce31388f5ab963871870535aaaa551629f5ad182384d60fdf82
rpm_dir/
rpm_dir/gzip-1.9-4.el8.src.rpm
```

If the results are correct, the image is ready to be used.



NOTE

It could take several hours after a container image is released for its associated source container to become available.

Additional resources

- **skopeo-copy** man page
- **skopeo-inspect** man page

CHAPTER 7. WORKING WITH PODS

Containers are the smallest unit that you can manage with Podman, Skopeo and Buildah container tools. A Podman pod is a group of one or more containers. The Pod concept was introduced by Kubernetes. Podman pods are similar to the Kubernetes definition. Pods are the smallest compute units that you can create, deploy, and manage in OpenShift or Kubernetes environments. Every Podman pod includes an infra container. This container holds the namespaces associated with the pod and allows Podman to connect other containers to the pod. It allows you to start and stop containers within the pod and the pod will stay running. The default infra container on the **registry.access.redhat.com/ubi8/pause** image.

7.1. CREATING PODS

This procedure shows how to create a pod with one container.

Prerequisites

- The Podman tool is installed.

```
# yum module install -y container-tools
```

Procedure

1. Create an empty pod:

```
$ podman pod create --name mypod
223df6b390b4ea87a090a4b5207f7b9b003187a6960bd37631ae9bc12c433aff
The pod is in the initial state Created.
```

   The pod is in the initial state Created.

2. Optional: List all pods:

```
$ podman pod ps
POD ID        NAME    STATUS    CREATED               # OF CONTAINERS   INFRA ID
223df6b390b4  mypod   Created   Less than a second ago  1               3afdcd93de3e
```

   Notice that the pod has one container in it.

3. Optional: List all pods and containers associated with them:

```
$ podman ps -a --pod
CONTAINER ID  IMAGE                            COMMAND  CREATED              STATUS   PORTS
NAMES              POD
3afdcd93de3e  registry.access.redhat.com/ubi8/pause          Less than a second ago
Created        223df6b390b4-infra  223df6b390b4
```

   You can see that the pod ID from **podman ps** command matches the pod ID in the **podman pod ps** command. The default infra container is based on the **registry.access.redhat.com/ubi8/pause** image.

4. Run a container named **myubi** in the existing pod named **mypod**:

```
$ podman run -dt --name myubi --pod mypod registry.access.redhat.com/ubi8/ubi
/bin/bash
5df5c48fea87860cf75822ceab8370548b04c78be9fc156570949013863ccf71
```

5. Optional: List all pods:

```
$ podman pod ps
POD ID        NAME    STATUS    CREATED               # OF CONTAINERS   INFRA ID
223df6b390b4  mypod   Running   Less than a second ago  2               3afdcd93de3e
```

You can see that the pod has two containers in it.

6. Optional: List all pods and containers associated with them:

```
$ podman ps -a --pod
CONTAINER ID  IMAGE                                 COMMAND    CREATED
STATUS                  PORTS  NAMES              POD
5df5c48fea87  registry.access.redhat.com/ubi8/ubi:latest  /bin/bash  Less than a second ago
Up Less than a second ago         myubi              223df6b390b4
3afdcd93de3e  registry.access.redhat.com/ubi8/pause                    Less than a
second ago  Up Less than a second ago         223df6b390b4-infra  223df6b390b4
```

Additional resources

- **podman-pod-create** man page
- Podman: Managing pods and containers in a local container runtime

7.2. DISPLAYING POD INFORMATION

This procedure provides information on how to display pod information.

Prerequisites

- The Podman tool is installed.

```
# yum module install -y container-tools
```

- The pod has been created. For details, see section Creating pods.

Procedure

- Display active processes running in a pod:
  - To display the running processes of containers in a pod, enter:

```
$ podman pod top mypod
USER   PID   PPID   %CPU    ELAPSED         TTY     TIME   COMMAND
0      1     0      0.000   24.077433518s   ?       0s     /pause
root   1     0      0.000   24.078146025s   pts/0   0s     /bin/bash
```

  - To display a live stream of resource usage stats for containers in one or more pods, enter:

```
$ podman pod stats -a --no-stream
ID            NAME             CPU %   MEM USAGE / LIMIT   MEM %   NET IO    BLOCK IO
PIDS
a9f807ffaacd  frosty_hodgkin   --      3.092MB / 16.7GB    0.02%   -- / --   -- / --    2
3b33001239ee  sleepy_stallman  --      -- / --             --      -- / --   -- / --    --
```

- To display information describing the pod, enter:

```
$ podman pod inspect mypod
{
   "Id": "db99446fa9c6d10b973d1ce55a42a6850357e0cd447d9bac5627bb2516b5b19a",
   "Name": "mypod",
   "Created": "2020-09-08T10:35:07.536541534+02:00",
   "CreateCommand": [
      "podman",
      "pod",
      "create",
      "--name",
      "mypod"
   ],
   "State": "Running",
   "Hostname": "mypod",
   "CreateCgroup": false,
   "CgroupParent": "/libpod_parent",
   "CgroupPath":
"/libpod_parent/db99446fa9c6d10b973d1ce55a42a6850357e0cd447d9bac5627bb2516b5
b19a",
   "CreateInfra": false,
   "InfraContainerID":
"891c54f70783dcad596d888040700d93f3ead01921894bc19c10b0a03c738ff7",
   "SharedNamespaces": [
      "uts",
      "ipc",
      "net"
   ],
   "NumContainers": 2,
   "Containers": [
      {
         "Id":
"891c54f70783dcad596d888040700d93f3ead01921894bc19c10b0a03c738ff7",
         "Name": "db99446fa9c6-infra",
         "State": "running"
      },
      {
         "Id":
"effc5bbcfe505b522e3bf8fbb5705a39f94a455a66fd81e542bcc27d39727d2d",
         "Name": "myubi",
         "State": "running"
      }
   ]
}
```

You can see information about containers in the pod.

Additional resources

- **podman pod top** man page
- **podman-pod-stats** man page
- **podman-pod-inspect** man page

7.3. STOPPING PODS

You can stop one or more pods using the **podman pod stop** command.

Prerequisites

- The Podman tool is installed.

```
# yum module install -y container-tools
```

- The pod has been created. For details, see section Creating pods.

Procedure

1. Stop the pod **mypod**:

```
$ podman pod stop mypod
```

2. Optional: List all pods and containers associated with them:

```
$ podman ps -a --pod
CONTAINER ID  IMAGE                                COMMAND    CREATED            STATUS
PORTS   NAMES              POD ID        PODNAME
5df5c48fea87  registry.redhat.io/ubi8/ubi:latest  /bin/bash  About a minute ago  Exited (0) 7
seconds ago          myubi              223df6b390b4  mypod

3afdcd93de3e  registry.access.redhat.com/8/pause                       About a minute ago
Exited (0) 7 seconds ago         8a4e6527ac9d-infra  223df6b390b4  mypod
```

You can see that the pod **mypod** and container **myubi** are in "Exited" status.

Additional resources

- **podman-pod-stop** man page

7.4. REMOVING PODS

You can remove one or more stopped pods and containers using the **podman pod rm** command.

Prerequisites

- The Podman tool is installed.

```
# yum module install -y container-tools
```

- The pod has been created. For details, see section Creating pods.

- The pod has been stopped. For details, see section Stopping pods.

Procedure

1. Remove the pod **mypod**, type:

```
$ podman pod rm mypod
223df6b390b4ea87a090a4b5207f7b9b003187a6960bd37631ae9bc12c433aff
```

   Note that removing the pod automatically removes all containers inside it.

2. Optional: Check that all containers and pods were removed:

```
$ podman ps
$ podman pod ps
```

Additional resources

- **podman-pod-rm** man page

CHAPTER 8. MANAGING A CONTAINER NETWORK

This chapter provides information on how to manage a container network.

8.1. LISTING CONTAINER NETWORKS

In Podman, there are two network behaviors - rootless and rootful:

- Rootless networking - the network is setup automatically, the container does not have an IP address.
- Rootful networking - the container has an IP address.

Procedure

- List all networks as a root user:

```
# podman network ls
NETWORK ID    NAME        VERSION     PLUGINS
2f259bab93aa  podman      0.4.0       bridge,portmap,firewall,tuning
```

  - By default, Podman provides a bridged network.
  - List of networks for a rootless user is the same as for a rootful user.

Additional resources

- **podman-network-ls** man page

8.2. INSPECTING A NETWORK

Display the IP range, enabled plugins, type of network, and so on, for a specified network listed by the **podman network ls** command.

Procedure

- Inspect the default **podman** network:

```
$ podman network inspect podman
[
   {
      "cniVersion": "0.4.0",
      "name": "podman",
      "plugins": [
         {
            "bridge": "cni-podman0",
            "hairpinMode": true,
            "ipMasq": true,
            "ipam": {
               "ranges": [
                  [
                     {
                        "gateway": "10.88.0.1",
                        "subnet": "10.88.0.0/16"
```

```
                    }
                ]
            ],
            "routes": [
                {
                    "dst": "0.0.0.0/0"
                }
            ],
            "type": "host-local"
        },
        "isGateway": true,
        "type": "bridge"
    },
    {
        "capabilities": {
            "portMappings": true
        },
        "type": "portmap"
    },
    {
        "type": "firewall"
    },
    {
        "type": "tuning"
    }
  ]
 }
]
```

You can see the IP range, enabled plugins, type of network, and other network settings.

Additional resources

- **podman-network-inspect** man page

8.3. CREATING A NETWORK

Use the **podman network create** command to create a new network.

NOTE

By default, Podman creates an external network. You can create an internal network using the **podman network create --internal** command. Containers in an internal network can communicate with other containers on the host, but cannot connect to the network outside of the host nor be reached from it.

Procedure

- Create the external network named **mynet**:

```
# podman network create mynet
/etc/cni/net.d/mynet.conflist
```

Verification

- List all networks:

```
# podman network ls
NETWORK ID    NAME        VERSION     PLUGINS
2f259bab93aa  podman      0.4.0       bridge,portmap,firewall,tuning
11c844f95e28  mynet       0.4.0       bridge,portmap,firewall,tuning,dnsname
```

You can see the created **mynet** network and default **podman** network.

NOTE

Beginning with Podman 4.0, the DNS plugin is enabled by default if you create a new external network using the **podman network create** command.

Additional resources

- **podman-network-create** man page

8.4. CONNECTING A CONTAINER TO A NETWORK

Use the **podman network connect** command to connect the container to the network.

Prerequisites

- A network has been created using the **podman network create** command.
- A container has been created.

Procedure

- Connect a container named **mycontainer** to a network named **mynet**:

```
# podman network connect mynet mycontainer
```

Verification

- Verify that the **mycontainer** is connected to the **mynet** network:

```
# podman inspect --format='{{.NetworkSettings.Networks}}' mycontainer
map[podman:0xc00042ab40 mynet:0xc00042ac60]
```

You can see that **mycontainer** is connected to **mynet** and **podman** networks.

Additional resources

- **podman-network-connect** man page

8.5. DISCONNECTING A CONTAINER FROM A NETWORK

Use the **podman network disconnect** command to disconnect the container from the network.

Prerequisites

- A network has been created using the **podman network create** command.
- A container is connected to a network.

Procedure

- Disconnect the container named **mycontainer** from the network named **mynet**:

```
# podman network disconnect mynet mycontainer
```

Verification

- Verify that the **mycontainer** is disconnected from the **mynet** network:

```
# podman inspect --format='{{.NetworkSettings.Networks}}' mycontainer
map[podman:0xc000537440]
```

You can see that **mycontainer** is disconnected from the **mynet** network, **mycontainer** is only connected to the default **podman** network.

Additional resources

- **podman-network-disconnect** man page

8.6. REMOVING A NETWORK

Use the **podman network rm** command to remove a specified network.

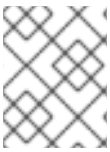
Procedure

1. List all networks:

```
# podman network ls
NETWORK ID    NAME        VERSION     PLUGINS
2f259bab93aa  podman      0.4.0       bridge,portmap,firewall,tuning
11c844f95e28  mynet       0.4.0       bridge,portmap,firewall,tuning,dnsname
```

2. Remove the **mynet** network:

```
# podman network rm mynet
mynet
```



NOTE

If the removed network has associated containers with it, you have to use the **podman network rm -f** command to delete containers and pods.

Verification

- Check if **mynet** network was removed:

```
# podman network ls
NETWORK ID    NAME        VERSION     PLUGINS
2f259bab93aa  podman      0.4.0       bridge,portmap,firewall,tuning
```

Additional resources

- **podman-network-rm** man page

8.7. REMOVING ALL UNUSED NETWORKS

Use the **podman network prune** to remove all unused networks. An unused network is a network which has no containers connected to it. The **podman network prune** command does not remove the default **podman** network.

Procedure

- Remove all unused networks:

```
# podman network prune
WARNING! This will remove all networks not used by at least one container.
Are you sure you want to continue? [y/N] y
```

Verification

- Verify that all networks were removed:

```
# podman network ls
NETWORK ID    NAME        VERSION     PLUGINS
2f259bab93aa  podman      0.4.0       bridge,portmap,firewall,tuning
```

Additional resources

- **podman-network-prune** man page

CHAPTER 9. COMMUNICATING AMONG CONTAINERS

This chapter provides information on how to communicate among containers.

9.1. THE NETWORK MODES AND LAYERS

There are several different network modes in Podman:

- **bridge** - creates another network on the default bridge network
- **container:<id>** - uses the same network as the container with **<id>** id
- **host** - uses the host network stack
- **network-id** - uses a user-defined network created by the **podman** network create command
- **private** - creates a new network for the container
- **slirp4nets** - creates a user network stack with slirp4netns, the default option for rootless containers

NOTE

The host mode gives the container full access to local system services such as D-bus, a system for interprocess communication (IPC), and is therefore considered insecure.

9.2. INSPECTING A NETWORK SETTINGS OF A CONTAINER

Use the **podman inspect** command with the **--format** option to display individual items from the **podman inspect** output.

Procedure

1. Display the IP address of a container:

```
# podman inspect --format='{{.NetworkSettings.IPAddress}}' containerName
```

2. Display all networks to which container is connected:

```
# podman inspect --format='{{.NetworkSettings.Networks}}' containerName
```

3. Display port mappings:

```
# podman inspect --format='{{.NetworkSettings.Ports}}' containerName
```

Additional resources

- **podman-inspect** man page

9.3. COMMUNICATING BETWEEN A CONTAINER AND AN APPLICATION

You can communicate between a container and an application. An application ports are in either listening or open state. These ports are automatically exposed to the container network, therefore, you can reach those containers using these networks. By default, the web server listens on port 80. Using this procedure, the **myubi** container communicates with the **web-container** application.

Procedure

1. Start the container named **web-container**:

```
# podman run -dt --name=web-container docker.io/library/httpd
```

2. List all containers:

```
# podman ps -a

CONTAINER ID  IMAGE                           COMMAND         CREATED        STATUS
PORTS       NAMES
b8c057333513  docker.io/library/httpd:latest  httpd-foreground  4 seconds ago  Up 5 seconds
ago             web-container
```

3. Inspect the container and display the IP address:

```
# podman inspect --format='{{.NetworkSettings.IPAddress}}' web-container

10.88.0.2
```

4. Run the **myubi** container and verify that web server is running:

```
# podman run -it --name=myubi ubi8/ubi curl 10.88.0.2:80

<html><body><h1>It works!</h1></body></html>
```

9.4. COMMUNICATING BETWEEN A CONTAINER AND A HOST

By default, the **podman** network is a bridge network. It means that a network device is bridging a container network to your host network.

Prerequisites

- The **web-container** is running. For more information, see section Communicating between a container and an application.

Procedure

1. Verify that the bridge is configured:

```
# podman network inspect podman | grep bridge

    "bridge": "cni-podman0",
    "type": "bridge"
```

2. Display the host network configuration:

```
# ip addr show cni-podman0

6: cni-podman0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default qlen 1000
    link/ether 62:af:a1:0a:ca:2e brd ff:ff:ff:ff:ff:ff
    inet 10.88.0.1/16 brd 10.88.255.255 scope global cni-podman0
       valid_lft forever preferred_lft forever
    inet6 fe80::60af:a1ff:fe0a:ca2e/64 scope link
       valid_lft forever preferred_lft forever
```

You can see that the **web-container** has an IP of the **cni-podman0** network and the network is bridged to the host.

3. Inspect the **web-container** and display its IP address:

```
# podman inspect --format='{{.NetworkSettings.IPAddress}}' web-container

10.88.0.2
```

4. Access the **web-container** directly from the host:

```
$ curl 10.88.0.2:80

<html><body><h1>It works!</h1></body></html>
```

Additional resources

- **podman-network** man page

9.5. COMMUNICATING BETWEEN CONTAINERS USING PORT MAPPING

The most convenient way to communicate between two containers is to use published ports. Ports can be published in two ways: automatically or manually.

Procedure

1. Run the unpublished container:

```
# podman run -dt --name=web1 ubi8/httpd-24
```

2. Run the automatically published container:

```
# podman run -dt --name=web2 -P ubi8/httpd-24
```

3. Run the manually published container and publish container port 80:

```
# podman run -dt --name=web3 -p 9090:80 ubi8/httpd-24
```

4. List all containers:

```
# podman ps
```

```
CONTAINER ID  IMAGE                                  COMMAND           CREATED
STATUS            PORTS                                  NAMES
f12fa79b8b39  registry.access.redhat.com/ubi8/httpd-24:latest  /usr/bin/run-http...  23
seconds ago  Up 24 seconds ago                                       web1
9024d9e815e2  registry.access.redhat.com/ubi8/httpd-24:latest  /usr/bin/run-http...  13
seconds ago  Up 13 seconds ago  0.0.0.0:43595->8080/tcp, 0.0.0.0:42423->8443/tcp  web2
03bc2a019f1b  registry.access.redhat.com/ubi8/httpd-24:latest  /usr/bin/run-http...  2 seconds
ago   Up 2 seconds ago   0.0.0.0:9090->80/tcp                        web3
```

You can see that:

- Container **web1** has no published ports and can be reached only by container network or a bridge.
- Container **web2** has automatically mapped ports 43595 and 42423 to publish the application ports 8080 and 8443, respectively.

NOTE

The automatic port mapping is possible because the **registry.access.redhat.com/8/httpd-24** image has the **EXPOSE 8080** and **EXPOSE 8443** commands in the Containerfile.

- Container **web3** has a manually published port. The host port 9090 is mapped to the container port 80.

5. Display the IP addresses of **web1** and **web3** containers:

```
# podman inspect --format='{{.NetworkSettings.IPAddress}}' web1
# podman inspect --format='{{.NetworkSettings.IPAddress}}' web3
```

6. Reach **web1** container using <IP>:<port> notation:

```
# curl 10.88.0.14:8080
...
<title>Test Page for the HTTP Server on Red Hat Enterprise Linux</title>
...
```

7. Reach **web2** container using localhost:<port> notation:

```
# curl localhost:43595
...
<title>Test Page for the HTTP Server on Red Hat Enterprise Linux</title>
...
```

8. Reach **web3** container using <IP>:<port> notation:

```
# curl 10.88.0.14:9090
...
<title>Test Page for the HTTP Server on Red Hat Enterprise Linux</title>
...
```

9.6. COMMUNICATING BETWEEN CONTAINERS USING DNS

When a DNS plugin is enabled, use a container name to address containers.

Prerequisites

- A network with the enabled DNS plugin has been created using the **podman network create** command.

Procedure

1. Run a **receiver** container attached to the **mynet** network:

```
# podman run -d --net mynet --name receiver ubi8 sleep 3000
```

2. Run a **sender** container and reach the **receiver** container by its name:

```
# podman run -it --rm --net mynet --name sender alpine ping receiver

PING rcv01 (10.89.0.2): 56 data bytes
64 bytes from 10.89.0.2: seq=0 ttl=42 time=0.041 ms
64 bytes from 10.89.0.2: seq=1 ttl=42 time=0.125 ms
64 bytes from 10.89.0.2: seq=2 ttl=42 time=0.109 ms
```

   Exit using the **CTRL+C**.

You can see that the **sender** container can ping the **receiver** container using its name.

9.7. COMMUNICATING BETWEEN TWO CONTAINERS IN A POD

All containers in the same pod share the IP addresses, MAC addresses and port mappings. You can communicate between containers in the same pod using localhost:port notation.

Procedure

1. Create a pod named **web-pod**:

```
$ podman pod create --name=web-pod
```

2. Run the web container named **web-container** in the pod:

```
$ podman container run -d --pod web-pod --name=web-container
docker.io/library/httpd
```

3. List all pods and containers associated with them:

```
$ podman ps --pod

CONTAINER ID  IMAGE                    COMMAND         CREATED       STATUS
PORTS       NAMES              POD ID       PODNAME
58653cf0cf09  k8s.gcr.io/pause:3.5                      4 minutes ago  Up 3 minutes ago
```

```
4e61a300c194-infra  4e61a300c194  web-pod
b3f4255afdb3  docker.io/library/httpd:latest  httpd-foreground  3 minutes ago  Up 3 minutes
ago              web-container  4e61a300c194  web-pod
```

4. Run the container in the **web-pod** based on the docker.io/library/fedora image:

```
$ podman container run -it --rm --pod web-pod docker.io/library/fedora curl localhost

<html><body><h1>It works!</h1></body></html>
```

You can see that the container can reach the **web-container**.

9.8. COMMUNICATING IN A POD

You must publish the ports for the container in a pod when a pod is created.

Procedure

1. Create a pod named **web-pod**:

```
# podman pod create --name=web-pod-publish -p 80:80
```

2. List all pods:

```
# podman pod ls

POD ID        NAME         STATUS   CREATED        INFRA ID      # OF CONTAINERS
26fe5de43ab3  publish-pod  Created  5 seconds ago  7de09076d2b3  1
```

3. Run the web container named **web-container** inside the **web-pod**:

```
# podman container run -d --pod web-pod-publish --name=web-container
docker.io/library/httpd
```

4. List containers

```
# podman ps

CONTAINER ID  IMAGE                 COMMAND          CREATED            STATUS
PORTS              NAMES
7de09076d2b3  k8s.gcr.io/pause:3.5                    About a minute ago  Up 23 seconds ago
0.0.0.0:80->80/tcp  26fe5de43ab3-infra
088befb90e59  docker.io/library/httpd  httpd-foreground  23 seconds ago      Up 23 seconds
ago  0.0.0.0:80->80/tcp  web-container
```

5. Verify that the **web-container** can be reached:

```
$ curl localhost:80

<html><body><h1>It works!</h1></body></html>
```

9.9. ATTACHING A POD TO THE CONTAINER NETWORK

Attach containers in pod to the network during the pod creation.

Procedure

1. Create a network named **pod-net**:

```
# podman network create pod-net

/etc/cni/net.d/pod-net.conflist
```

2. Create a pod **web-pod**:

```
# podman pod create --net pod-net --name web-pod
```

3. Run a container named **web-container** inside the **web-pod**:

```
# podman run -d --pod webt-pod --name=web-container docker.io/library/httpd
```

4. Optional. Display the pods the containers are associated with:

```
# podman ps -p

CONTAINER ID  IMAGE                           COMMAND           CREATED        STATUS
PORTS       NAMES               POD ID        PODNAME
b7d6871d018c  registry.access.redhat.com/ubi8/pause:latest                       9 minutes
ago  Up 6 minutes ago               a8e7360326ba-infra  a8e7360326ba  web-pod
645835585e24  docker.io/library/httpd:latest  httpd-foreground  6 minutes ago  Up 6 minutes
ago               web-container     a8e7360326ba  web-pod
```

Verification

- Show all networks connected to the container:

```
# podman ps --format="{{.Networks}}"

pod-net
```

CHAPTER 10. SETTING CONTAINER NETWORK MODES

This chapter provides information on how to set different network modes.

10.1. RUNNING CONTAINERS WITH A STATIC IP

The **podman run** command with the **--ip** option sets the container network interface to a particular IP address (for example, 10.88.0.44). To verify that you set the IP address correctly, run the **podman inspect** command.

Procedure

- Set the container network interface to the IP address 10.88.0.44:

```
# podman run -d --name=myubi --ip=10.88.0.44

registry.access.redhat.com/ubi8/ubi
efde5f0a8c723f70dd5cb5dc3d5039df3b962fae65575b08662e0d5b5f9fbe85
```

Verification

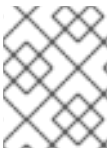
- Check that the IP address is set properly:

```
# podman inspect --format='{{.NetworkSettings.IPAddress}}' myubi

10.88.0.44
```

10.2. RUNNING THE DHCP PLUGIN WITHOUT SYSTEMD

Use the **podman run --network** command to connect to a user-defined network. While most of the container images do not have a DHCP client, the **dhcp** plugin acts as a proxy DHCP client for the containers to interact with a DHCP server.



NOTE

This procedure only applies to rootfull containers. Rootless containers do not use the **dhcp** plugin.

Procedure

1. Manually run the **dhcp** plugin:

```
# /usr/libexec/cni/dhcp daemon &

[1] 4966
```

2. Check that the **dhcp** plugin is running:

```
# ps -a | grep dhcp

4966 pts/1    00:00:00 dhcp
```

3. Run the **alpine** container:

```
# podman run -it --rm --network=example alpine ip addr show enp1s0

Resolved "alpine" as an alias (/etc/containers/registries.conf.d/000-shortnames.conf)
Trying to pull docker.io/library/alpine:latest...
...
Storing signatures

2: eth0@eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
UP
    link/ether f6:dd:1b:a7:9b:92 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.22/24 brd 192.168.1.255 scope global eth0
    ...
```

In this example:

- The **--network=example** option specifies the network named example to connect.
- The **ip addr show enp1s0** command inside the **alpine** container checks the IP address of the network interface **enp1s0**.
- The host network is 192.168.1.0/24
- The **eth0** interface leases an IP address of 192.168.1.122 for the alpine container.

NOTE

This configuration may exhaust the available DHCP addresses if you have a large number of short-lived containers and a DHCP server with long leases.

Additional resources

- Leasing routable IP addresses with Podman containers

10.3. RUNNING THE DHCP PLUGIN USING SYSTEMD

You can use the systemd unit file to run the **dhcp** plugin.

Procedure

1. Create the socket unit file:

```
# cat /usr/lib/systemd/system/io.podman.dhcp.socket
[Unit]
Description=DHCP Client for CNI

[Socket]
ListenStream=%t/cni/dhcp.sock
SocketMode=0600

[Install]
WantedBy=sockets.target
```

2. Create the service unit file:

```
# cat /usr/lib/systemd/system/io.podman.dhcp.service
[Unit]
Description=DHCP Client CNI Service
Requires=io.podman.dhcp.socket
After=io.podman.dhcp.socket

[Service]
Type=simple
ExecStart=/usr/libexec/cni/dhcp daemon
TimeoutStopSec=30
KillMode=process


[Install]
WantedBy=multi-user.target
Also=io.podman.dhcp.socket
```

3. Start the service immediately:

```
# systemctl --now enable io.podman.dhcp.socket
```

Verification

- Check the status of the socket:

```
# systemctl status io.podman.dhcp.socket
io.podman.dhcp.socket - DHCP Client for CNI
Loaded: loaded (/usr/lib/systemd/system/io.podman.dhcp.socket; enabled; vendor preset:
disabled)
Active: active (listening) since Mon 2022-01-03 18:08:10 CET; 39s ago
Listen: /run/cni/dhcp.sock (Stream)
CGroup: /system.slice/io.podman.dhcp.socket
```

Additional resources

- Leasing routable IP addresses with Podman containers

10.4. THE MACVLAN PLUGIN

Most of the container images do not have a DHCP client, the **dhcp** plugin acts as a proxy DHCP client for the containers to interact with a DHCP server.

The host system does not have network access to the container. To allow network connections from outside the host to the container, the container has to have an IP on the same network as the host. The **macvlan** plugin enables you to connect a container to the same network as the host.

NOTE

This procedure only applies to rootfull containers. Rootless containers are not able to use the **macvlan** and **dhcp** plugins.

NOTE

You can create a macvlan network using the **podman network create --macvlan** command.

Additional resources

- Leasing routable IP addresses with Podman containers
- **podman-network-create** man page

10.5. SWITCHING THE NETWORK STACK FROM CNI TO NETAVARK

Previously, containers were able to use DNS only when connected to the single Container Network Interface (CNI) plugin. Netavark is a network stack for containers. You can use Netavark with Podman and other Open Container Initiative (OCI) container management applications. The advanced network stack for Podman is compatible with advanced Docker functionalities. Now, containers in multiple networks access containers on any of those networks.

Netavark is capable of the following:

- Create, manage, and remove network interfaces, including bridge and MACVLAN interfaces.
- Configure firewall settings, such as network address translation (NAT) and port mapping rules.
- Support IPv4 and IPv6.
- Improve support for containers in multiple networks.

Procedure

1. If the **/etc/containers/container.conf** file does not exist, copy the **/usr/share/containers.conf** file to the **/etc/containers/** directory:

```
# cp /usr/share/containers/containers.conf /etc/containers/
```

2. Edit the **/etc/containers/container.conf** file, and add the following content to the **[network]** section:

```
network_backend="netavark"
```

3. If you have any containers or pods, reset the storage back to the initial state:

```
# podman system reset
```

4. Reboot the system:

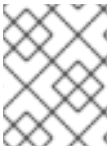
```
# reboot
```

Verification

- Verify that the network stack is changed to Netavark:

```
# cat /etc/containers/container.conf
```

```
...
[network]
network_backend="netavark"
...
```



NOTE

If you are using Podman 4.0.0 or later, use the **podman info** command to check the network stack setting.

Additional resources

- Podman 4.0's new network stack: What you need to know
- **podman-system-reset** man page

10.6. SWITCHING THE NETWORK STACK FROM NETAVARK TO CNI

Procedure

1. If the **/etc/containers/container.conf** file does not exist, copy the **/usr/share/containers.conf** file to the **/etc/containers/** directory:

```
# cp /usr/share/containers/containers.conf /etc/containers/
```

2. Edit the **/etc/containers/container.conf** file, and add the following content to the **[network]** section:

```
network_backend="cni"
```

3. If you have any containers or pods, reset the storage back to the initial state:

```
# podman system reset
```

4. Reboot the system:

```
# reboot
```

Verification

- Verify that the network stack is changed to CNI:

```
# cat /etc/containers/container.conf
...
[network]
network_backend="cni"
...
```

NOTE

If you are using Podman 4.0.0 or later, use the **podman info** command to check the network stack setting.

Additional resources

- Podman 4.0's new network stack: What you need to know
- **podman-system-reset** man page

CHAPTER 11. PORTING CONTAINERS TO OPENSHIFT USING PODMAN

This chapter describes how to generate portable descriptions of containers and pods using the YAML ("YAML Ain't Markup Language") format. The YAML is a text format used to describe the configuration data.

The YAML files are:

- Readable.
- Easy to generate.
- Portable between environments (for example between RHEL and OpenShift).
- Portable between programming languages.
- Convenient to use (no need to add all the parameters to the command line).

Reasons to use YAML files:

1. You can re-run a local orchestrated set of containers and pods with minimal input required which can be useful for iterative development.
2. You can run the same containers and pods on another machine. For example, to run an application in an OpenShift environment and to ensure that the application is working correctly. You can use **podman generate kube** command to generate a Kubernetes YAML file. Then, you can use **podman play** command to test the creation of pods and containers on your local system before you transfer the generated YAML files to the Kubernetes or OpenShift environment. Using the **podman play** command, you can also recreate pods and containers originally created in OpenShift or Kubernetes environments.

11.1. GENERATING A KUBERNETES YAML FILE USING PODMAN

This procedure describes how to create a pod with one container and generate the Kubernetes YAML file using the **podman generate kube** command.

Prerequisites

- The pod has been created. For details, see section Creating pods.

Procedure

1. List all pods and containers associated with them:

```
$ podman ps -a --pod
CONTAINER ID  IMAGE                                  COMMAND    CREATED
STATUS                  PORTS  NAMES              POD
5df5c48fea87  registry.access.redhat.com/ubi8/ubi:latest  /bin/bash  Less than a second ago
Up Less than a second ago         myubi              223df6b390b4
3afdcd93de3e  k8s.gcr.io/pause:3.1                        Less than a second ago  Up Less
than a second ago         223df6b390b4-infra  223df6b390b4
```

2. Use the pod name or ID to generate the Kubernetes YAML file:

```
$ podman generate kube mypod > mypod.yaml
```

Note that the **podman generate** command does not reflect any Logical Volume Manager (LVM) logical volumes or physical volumes that might be attached to the container.

3. Display the **mypod.yaml** file:

```
$ cat mypod.yaml
# Generation of Kubernetes YAML is still under development!
#
# Save the output of this file and use kubectl create -f to import
# it into Kubernetes.
#
# Created with podman-1.6.4
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: "2020-06-09T10:31:56Z"
  labels:
app: mypod
  name: mypod
spec:
  containers:
  - command:
        - /bin/bash
        env:
        - name: PATH
            value: /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
        - name: TERM
            value: xterm
        - name: HOSTNAME
        - name: container
            value: oci
        image: registry.access.redhat.com/ubi8/ubi:latest
        name: myubi
        resources: {}
        securityContext:
            allowPrivilegeEscalation: true
            capabilities: {}
            privileged: false
            readOnlyRootFilesystem: false
        tty: true
        workingDir: /
status: {}
```

Additional resources

- **podman-generate-kube** man page
- Podman: Managing pods and containers in a local container runtime

11.2. GENERATING A KUBERNETES YAML FILE IN OPENSHIFT ENVIRONMENT

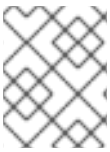
In the OpenShift environment, use the **oc create** command to generate the YAML files describing your application.

Procedure

- Generate the YAML file for your **myapp** application:

```
$ oc create myapp --image=me/myapp:v1 -o yaml --dry-run > myapp.yaml
```

The **oc create** command creates and run the **myapp** image. The object is printed using the **--dry-run** option and redirected into the **myapp.yaml** output file.



NOTE

In the Kubernetes environment, you can use the **kubectl create** command with the same flags.

11.3. STARTING CONTAINERS AND PODS WITH PODMAN

With the generated YAML files, you can automatically start containers and pods in any environment. Note that the YAML files must not be generated by the Podman. The **podman play kube** command allows you to recreate pods and containers based on the YAML input file.

Procedure

1. Create the pod and the container from the **mypod.yaml** file:

```
$ podman play kube mypod.yaml
Pod:
b8c5b99ba846ccff76c3ef257e5761c2d8a5ca4d7ffa3880531aec79c0dacb22
Container:
848179395ebd33dd91d14ffbde7ae273158d9695a081468f487af4e356888ece
```

2. List all pods:

```
$ podman pod ps
POD ID        NAME    STATUS    CREATED         # OF CONTAINERS   INFRA ID
b8c5b99ba846  mypod   Running   19 seconds ago  2                 aa4220eaf4bb
```

3. List all pods and containers associated with them:

```
$ podman ps -a --pod
CONTAINER ID  IMAGE                                  COMMAND    CREATED            STATUS
PORTS  NAMES              POD
848179395ebd  registry.access.redhat.com/ubi8/ubi:latest  /bin/bash  About a minute ago  Up
About a minute ago        myubi               b8c5b99ba846
aa4220eaf4bb  k8s.gcr.io/pause:3.1                        About a minute ago  Up About a
minute ago        b8c5b99ba846-infra  b8c5b99ba846
```

The pod IDs from **podman ps** command matches the pod ID from the **podman pod ps** command.

Additional resources

- **podman-play-kube** man page
- Podman can now ease the transition to Kubernetes and CRI-O

11.4. STARTING CONTAINERS AND PODS IN OPENSHIFT ENVIRONMENT

You can use the **oc create** command to create pods and containers in the OpenShift environment.

Procedure

- Create a pod from the YAML file in the OpenShift environment:

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

NOTE

In the Kubernetes environment, you can use the **kubectl create** command with the same flags.

11.5. MANUALLY RUNNING CONTAINERS AND PODS USING PODMAN

The following procedure shows how to manually create a WordPress content management system paired with a MariaDB database using Podman.

Suppose the following directory layout:

```
├── mariadb-conf
│   ├── Containerfile
│   ├── my.cnf
```

Procedure

1. Display the **mariadb-conf/Containerfile** file:

```
$ cat mariadb-conf/Containerfile
FROM docker.io/library/mariadb
COPY my.cnf /etc/mysql/my.cnf
```

2. Display the **mariadb-conf/my.cnf** file:

```
[client-server]
# Port or socket location where to connect
port = 3306
socket = /run/mysqld/mysqld.sock

# Import all .cnf files from the configuration directory
[mariadbd]
skip-host-cache
skip-name-resolve
bind-address = 127.0.0.1
```

```
!includedir /etc/mysql/mariadb.conf.d/
!includedir /etc/mysql/conf.d/
```

3. Build the **docker.io/library/mariadb** image using **mariadb-conf/Containerfile**:

```
$ cd mariadb-conf
$ podman build -t mariadb-conf .
$ cd ..
STEP 1: FROM docker.io/library/mariadb
Trying to pull docker.io/library/mariadb:latest...
Getting image source signatures
Copying blob 7b1a6ab2e44d done
...
Storing signatures
STEP 2: COPY my.cnf /etc/mysql/my.cnf
STEP 3: COMMIT mariadb-conf
--> ffae584aa6e
Successfully tagged localhost/mariadb-conf:latest
ffae584aa6e733ee1cdf89c053337502e1089d1620ff05680b6818a96eec3c17
```

4. Optional. List all images:

```
$ podman images
LIST IMAGES
REPOSITORY                                  TAG         IMAGE ID      CREATED
SIZE
localhost/mariadb-conf                      latest      b66fa0fa0ef2  57 seconds ago
416 MB
```

5. Create the pod named **wordpresspod** and configure port mappings between the container and the host system:

```
$ podman pod create --name wordpresspod -p 8080:80
```

6. Create the **mydb** container inside the **wordpresspod** pod:

```
$ podman run --detach --pod wordpresspod \ -e MYSQL_ROOT_PASSWORD=1234 \ -e
MYSQL_DATABASE=mywpdb \ -e MYSQL_USER=mywpuser \ -e
MYSQL_PASSWORD=1234 \ --name mydb localhost/mariadb-conf
```

7. Create the **myweb** container inside the **wordpresspod** pod:

```
$ podman run --detach --pod wordpresspod \ -e WORDPRESS_DB_HOST=127.0.0.1 \ -e
WORDPRESS_DB_NAME=mywpdb \ -e WORDPRESS_DB_USER=mywpuser \ -e
WORDPRESS_DB_PASSWORD=1234 \ --name myweb docker.io/wordpress
```

8. Optional. List all pods and containers associated with them:

```
$ podman ps --pod -a
CONTAINER ID  IMAGE                          COMMAND             CREATED
STATUS                  PORTS               NAMES               POD ID        PODNAME
9ea56f771915  k8s.gcr.io/pause:3.5                               Less than a second ago  Up Less
than a second ago  0.0.0.0:8080->80/tcp  4b7f054a6f01-infra  4b7f054a6f01  wordpresspod
```

```
60e8dbbabac5  localhost/mariadb-conf:latest       mariadbd             Less than a second ago
Up Less than a second ago  0.0.0.0:8080->80/tcp  mydb                 4b7f054a6f01
wordpresspod
045d3d506e50  docker.io/library/wordpress:latest  apache2-foregroun...  Less than a second
ago  Up Less than a second ago  0.0.0.0:8080->80/tcp  myweb              4b7f054a6f01
wordpresspod
```

Verification

- Verify that the pod is running: Visit the http://localhost:8080/wp-admin/install.php page or use the **curl** command:

```
$ curl http://localhost:8080/wp-admin/install.php
<!DOCTYPE html>
<html lang="en-US" xml:lang="en-US">
<head>
...
</head>
<body class="wp-core-ui">
<p id="logo">WordPress</p>
   <h1>Welcome</h1>
...
```

Additional resources

- Build Kubernetes pods with Podman play kube
- **podman-play-kube** man page

11.6. GENERATING A YAML FILE USING PODMAN

You can generate a Kubernetes YAML file using the **podman generate kube** command.

Prerequisites

- The pod named **wordpresspod** has been created. For details, see section Creating pods.

Procedure

1. List all pods and containers associated with them:

```
$ podman ps --pod -a
CONTAINER ID  IMAGE                           COMMAND               CREATED
STATUS                   PORTS               NAMES               POD ID        PODNAME
9ea56f771915  k8s.gcr.io/pause:3.5                                  Less than a second ago  Up Less
than a second ago  0.0.0.0:8080->80/tcp  4b7f054a6f01-infra  4b7f054a6f01  wordpresspod
60e8dbbabac5  localhost/mariadb-conf:latest       mariadbd             Less than a second ago
Up Less than a second ago  0.0.0.0:8080->80/tcp  mydb                 4b7f054a6f01
wordpresspod
045d3d506e50  docker.io/library/wordpress:latest  apache2-foregroun...  Less than a second
ago  Up Less than a second ago  0.0.0.0:8080->80/tcp  myweb              4b7f054a6f01
wordpresspod
```

2. Use the pod name or ID to generate the Kubernetes YAML file:

```
$ podman generate kube wordpresspod >> wordpresspod.yaml
```

Verification

- Display the **wordpresspod.yaml** file:

```
$ cat wordpresspod.yaml
...
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: "2021-12-09T15:09:30Z"
  labels:
    app: wordpresspod
  name: wordpresspod
spec:
  containers:
  - args:
        value: podman
      - name: MYSQL_PASSWORD
        value: "1234"
      - name: MYSQL_MAJOR
        value: "8.0"
      - name: MYSQL_VERSION
        value: 8.0.27-1debian10
      - name: MYSQL_ROOT_PASSWORD
        value: "1234"
      - name: MYSQL_DATABASE
        value: mywpdb
      - name: MYSQL_USER
        value: mywpuser
        image: mariadb
            name: mydb
            ports:
            - containerPort: 80
              hostPort: 8080
              protocol: TCP
  - args:
    - name: WORDPRESS_DB_NAME
      value: mywpdb
    - name: WORDPRESS_DB_PASSWORD
      value: "1234"
    - name: WORDPRESS_DB_HOST
      value: 127.0.0.1
    - name: WORDPRESS_DB_USER
      value: mywpuser
      image: docker.io/library/wordpress:latest
      name: myweb
```

Additional resources

- Build Kubernetes pods with Podman play kube
- **podman-play-kube** man page

11.7. AUTOMATICALLY RUNNING CONTAINERS AND PODS USING PODMAN

You can use the **podman play kube** command to test the creation of pods and containers on your local system before you transfer the generated YAML files to the Kubernetes or OpenShift environment.

The **podman play kube** command can also automatically build and run multiple pods with multiple containers in the pod using the YAML file similarly to the docker compose command. The images are automatically built if the following conditions are met:

1. a directory with the same name as the image used in YAML file exists
2. that directory contains a Containerfile

Prerequisites

- The pod named **wordpresspod** has been created. For details, see section Manually running containers and pods using Podman.
- The YAML file has been generated. For details, see section Generating a YAML file using Podman.
- To repeat the whole scenario from the beginning, delete locally stored images:

```
$ podman rmi localhost/mariadb-conf
$ podman rmi docker.io/library/wordpress
$ podman rmi docker.io/library/mysql
```

Procedure

1. Create the wordpress pod using the **wordpress.yaml** file:

```
STEP 1/2: FROM docker.io/library/mariadb
STEP 2/2: COPY my.cnf /etc/mysql/my.cnf
COMMIT localhost/mariadb-conf:latest
--> 428832c45d0
Successfully tagged localhost/mariadb-conf:latest
428832c45d07d78bb9cb34e0296a7dc205026c2fe4d636c54912c3d6bab7f399
Trying to pull docker.io/library/wordpress:latest...
Getting image source signatures
Copying blob 99c3c1c4d556 done
...
Storing signatures
Pod:
3e391d091d190756e655219a34de55583eed3ef59470aadd214c1fc48cae92ac
Containers:
6c59ebe968467d7fdb961c74a175c88cb5257fed7fb3d375c002899ea855ae1f
29717878452ff56299531f79832723d3a620a403f4a996090ea987233df0bc3d
```

The **podman play kube** command:

- Automatically build the **localhost/mariadb-conf:latest** image based on **docker.io/library/mariadb** image.
- Pull the **docker.io/library/wordpress:latest** image.

- Create a pod named **wordpresspod** with two containers named **wordpresspod-mydb** and **wordpresspod-myweb**.

2. List all containers and pods:

```
$ podman ps --pod -a
CONTAINER ID  IMAGE                                COMMAND               CREATED         STATUS
PORTS                 NAMES               POD ID        PODNAME
a1dbf7b5606c  k8s.gcr.io/pause:3.5                                      3 minutes ago   Up 2 minutes ago
0.0.0.0:8080->80/tcp  3e391d091d19-infra  3e391d091d19  wordpresspod
6c59ebe96846  localhost/mariadb-conf:latest        mariadbd              2 minutes ago   Exited (1)
2 minutes ago  0.0.0.0:8080->80/tcp  wordpresspod-mydb   3e391d091d19  wordpresspod
29717878452f  docker.io/library/wordpress:latest  apache2-foregroun...  2 minutes ago   Up 2
minutes ago           0.0.0.0:8080->80/tcp  wordpresspod-myweb  3e391d091d19
wordpresspod
```

Verification

- Verify that the pod is running: Visit the http://localhost:8080/wp-admin/install.php page or use the **curl** command:

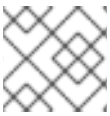
```
$ curl http://localhost:8080/wp-admin/install.php
<!DOCTYPE html>
<html lang="en-US" xml:lang="en-US">
<head>
...
</head>
<body class="wp-core-ui">
<p id="logo">WordPress</p>
   <h1>Welcome</h1>
...
```

Additional resources

- Build Kubernetes pods with Podman play kube
- **podman-play-kube** man page

11.8. AUTOMATICALLY STOPPING AND REMOVING PODS USING PODMAN

The **podman play kube --down** command stops and removes all pods and their containers.



NOTE

If a volume is in use, it is not removed.

Prerequisites

- The pod named **wordpresspod** has been created. For details, see section Manually running containers and pods using Podman.

- The YAML file has been generated. For details, see section Generating a YAML file using Podman.
- The pod is running. For details, see section Automatically running containers and pods using Podman.

Procedure

- Remove all pods and containers created by the **wordpresspod.yaml** file:

```
$ podman play kube --down wordpresspod.yaml
Pods stopped:
3e391d091d190756e655219a34de55583eed3ef59470aadd214c1fc48cae92ac
Pods removed:
3e391d091d190756e655219a34de55583eed3ef59470aadd214c1fc48cae92ac
```

Verification

- Verify that all pods and containers created by the **wordpresspod.yaml** file were removed:

```
$ podman ps --pod -a
CONTAINER ID  IMAGE                         COMMAND           CREATED
STATUS                  PORTS             NAMES             POD ID       PODNAME
```

Additional resources

- Build Kubernetes pods with Podman play kube
- **podman-play-kube** man page

CHAPTER 12. PORTING CONTAINERS TO SYSTEMD USING PODMAN

Podman (Pod Manager) is a fully featured container engine that is a simple daemonless tool. Podman provides a Docker-CLI comparable command line that eases the transition from other container engines and allows the management of pods, containers and images.

Podman was not originally designed to bring up an entire Linux system or manage services for such things as start-up order, dependency checking, and failed service recovery. That is the job of a full-blown initialization system like systemd. Red Hat has become a leader in integrating containers with systemd, so that OCI and Docker-formatted containers built by Podman can be managed in the same way that other services and features are managed in a Linux system. You can use the systemd initialization service to work with pods and containers. You can use the **podman generate systemd** command to generate a systemd unit file for containers and pods.

With systemd unit files, you can:

- Set up a container or pod to start as a systemd service.
- Define the order in which the containerized service runs and check for dependencies (for example making sure another service is running, a file is available or a resource is mounted).
- Control the state of the systemd system using the **systemctl** command.

This chapter provides you with information on how to generate portable descriptions of containers and pods using systemd unit files.

12.1. ENABLING SYSTEMD SERVICES

When enabling the service, you have different options.

Procedure

- Enable the service:
  - To enable a service at system start, no matter if user is logged in or not, enter:

    ```
    # systemctl enable <service>
    ```

    You have to copy the systemd unit files to the **/etc/systemd/system** directory.
  - To start a service at user login and stop it at user logout, enter:

    ```
    $ systemctl --user enable <service>
    ```

    You have to copy the systemd unit files to the **$HOME/.config/systemd/user** directory.
  - To enable users to start a service at system start and persist over logouts, enter:

    ```
    # loginctl enable-linger <username>
    ```

Additional resources

- **systemctl** man page

- **loginctl** man page
- Making systemd services start at boot time

12.2. GENERATING A SYSTEMD UNIT FILE USING PODMAN

Podman allows systemd to control and manage container processes. You can generate a systemd unit file for the existing containers and pods using **podman generate systemd** command. It is recommended to use **podman generate systemd** because the generated units files change frequently (via updates to Podman) and the **podman generate systemd** ensures that you get the latest version of unit files.

Procedure

1. Create a container (for example **myubi**):

```
$ podman create --name myubi registry.access.redhat.com/ubi8:latest sleep infinity
0280afe98bb75a5c5e713b28de4b7c5cb49f156f1cce4a208f13fee2f75cb453
```

2. Use the container name or ID to generate the systemd unit file and direct it into the **~/.config/systemd/user/container-myubi.service** file:

```
$ podman generate systemd --name myubi > ~/.config/systemd/user/container-myubi.service
```

Verification steps

- Display the content of generated systemd unit file:

```
$ cat ~/.config/systemd/user/container-myubi.service
# container-myubi.service
# autogenerated by Podman 3.3.1
# Wed Sep  8 20:34:46 CEST 2021

[Unit]
Description=Podman container-myubi.service
Documentation=man:podman-generate-systemd(1)
Wants=network-online.target
After=network-online.target
RequiresMountsFor=/run/user/1000/containers

[Service]
Environment=PODMAN_SYSTEMD_UNIT=%n
Restart=on-failure
TimeoutStopSec=70
ExecStart=/usr/bin/podman start myubi
ExecStop=/usr/bin/podman stop -t 10 myubi
ExecStopPost=/usr/bin/podman stop -t 10 myubi
PIDFile=/run/user/1000/containers/overlay-
containers/9683103f58a32192c84801f0be93446cb33c1ee7d9cdda225b78049d7c5deea4/user
data/conmon.pid
Type=forking
```

```
[Install]
WantedBy=multi-user.target default.target
```

- The **Restart=on-failure** line sets the restart policy and instructs systemd to restart when the service cannot be started or stopped cleanly, or when the process exits non-zero.
- The **ExecStart** line describes how we start the container.
- The **ExecStop** line describes how we stop and remove the container.

Additional resources

- Running containers with Podman and shareable systemd services

12.3. AUTO-GENERATING A SYSTEMD UNIT FILE USING PODMAN

By default, Podman generates a unit file for existing containers or pods. You can generate more portable systemd unit files using the **podman generate systemd --new**. The **--new** flag instructs Podman to generate unit files that create, start and remove containers.

Procedure

1. Pull the image you want to use on your system. For example, to pull the **httpd-24** image:

```
# podman pull registry.access.redhat.com/ubi8/httpd-24
```

2. Optional. List all images available on your system:

```
# podman images
REPOSITORY                                TAG     IMAGE ID      CREATED       SIZE
registry.access.redhat.com/ubi8/httpd-24  latest  8594be0a0b57  2 weeks ago   462
MB
```

3. Create the **httpd** container:

```
# podman create --name httpd -p 8080:8080 registry.access.redhat.com/ubi8/httpd-24
cdb9f981cf143021b1679599d860026b13a77187f75e46cc0eac85293710a4b1
```

4. Optional. Verify the container has been created:

```
# podman ps -a
CONTAINER ID  IMAGE                                      COMMAND            CREATED
STATUS      PORTS                 NAMES
cdb9f981cf14  registry.access.redhat.com/ubi8/httpd-24:latest  /usr/bin/run-http...  5 minutes
ago  Created     0.0.0.0:8080->8080/tcp  httpd
```

5. Generate a systemd unit file for the **httpd** container:

```
# podman generate systemd --new --files --name httpd
/root/container-httpd.service
```

6. Display the content of the generated **container-httpd.service** systemd unit file:

```
# cat /root/container-httpd.service
# container-httpd.service
# autogenerated by Podman 3.3.1
# Wed Sep  8 20:41:44 CEST 2021

[Unit]
Description=Podman container-httpd.service
Documentation=man:podman-generate-systemd(1)
Wants=network-online.target
After=network-online.target
RequiresMountsFor=%t/containers

[Service]
Environment=PODMAN_SYSTEMD_UNIT=%n
Restart=on-failure
TimeoutStopSec=70
ExecStartPre=/bin/rm -f %t/%n.ctr-id
ExecStart=/usr/bin/podman run --cidfile=%t/%n.ctr-id --sdnotify=conmon --cgroups=no-conmon --rm -d --replace --name httpd -p 8080:8080 registry.access.redhat.com/ubi8/httpd-24
ExecStop=/usr/bin/podman stop --ignore --cidfile=%t/%n.ctr-id
ExecStopPost=/usr/bin/podman rm -f --ignore --cidfile=%t/%n.ctr-id
Type=notify
NotifyAccess=all

[Install]
WantedBy=multi-user.target default.target
```

NOTE

Unit files generated using the **--new** option do not expect containers and pods to exist. Therefore, they perform the **podman run** command when starting the service (see the **ExecStart** line) instead of the **podman start** command. For example, see section Generating a systemd unit file using Podman .

- The **podman run** command uses the following command-line options:
  - The **--conmon-pidfile** option points to a path to store the process ID for the **conmon** process running on the host. The **conmon** process terminates with the same exit status as the container, which allows systemd to report the correct service status and restart the container if needed.
  - The **--cidfile** option points to the path that stores the container ID.
  - The **%t** is the path to the run time directory root, for example **/run/user/$UserID**.
  - The **%n** is the full name of the service.

7. Copy unit files to **/usr/lib/systemd/system** for installing them as a root user:

```
# cp -Z container-httpd.service /etc/systemd/system
```

8. Enable and start the **container-httpd.service**:

```
# systemctl daemon-reload
# systemctl enable --now container-httpd.service
```

```
Created symlink /etc/systemd/system/multi-user.target.wants/container-httpd.service →
/etc/systemd/system/container-httpd.service.
Created symlink /etc/systemd/system/default.target.wants/container-httpd.service →
/etc/systemd/system/container-httpd.service.
```

Verification steps

- Check the status of the **container-httpd.service**:

```
# systemctl status container-httpd.service
   ● container-httpd.service - Podman container-httpd.service
      Loaded: loaded (/etc/systemd/system/container-httpd.service; enabled; vendor preset:
disabled)
      Active: active (running) since Tue 2021-08-24 09:53:40 EDT; 1min 5s ago
        Docs: man:podman-generate-systemd(1)
     Process: 493317 ExecStart=/usr/bin/podman run --conmon-pidfile /run/container-
httpd.pid --cidfile /run/container-httpd.ctr-id --cgroups=no-conmon -d --repla>
     Process: 493315 ExecStartPre=/bin/rm -f /run/container-httpd.pid /run/container-httpd.ctr-
id (code=exited, status=0/SUCCESS)
    Main PID: 493435 (conmon)
   ...
```

Additional resources

- Improved Systemd Integration with Podman 2.0
- Making systemd services start at boot time

12.4. AUTO-STARTING CONTAINERS USING SYSTEMD

You can control the state of the systemd system and service manager using the **systemctl** command. This section shows the general procedure on how to enable, start, stop the service as a non-root user. To install the service as a root user, omit the **--user** option.

Procedure

1. Reload systemd manager configuration:

```
# systemctl --user daemon-reload
```

2. Enable the service **container.service** and start it at boot time:

```
# systemctl --user enable container.service
```

3. Start the service immediately:

```
# systemctl --user start container.service
```

4. Check the status of the service:

```
$ systemctl --user status container.service
● container.service - Podman container.service
   Loaded: loaded (/home/user/.config/systemd/user/container.service; enabled; vendor
```

```
preset: enabled)
   Active: active (running) since Wed 2020-09-16 11:56:57 CEST; 8s ago
     Docs: man:podman-generate-systemd(1)
  Process: 80602 ExecStart=/usr/bin/podman run --conmon-pidfile
//run/user/1000/container.service-pid --cidfile //run/user/1000/container.service-cid -d ubi8-
minimal:>
  Process: 80601 ExecStartPre=/usr/bin/rm -f //run/user/1000/container.service-pid
//run/user/1000/container.service-cid (code=exited, status=0/SUCCESS)
 Main PID: 80617 (conmon)
   CGroup: /user.slice/user-1000.slice/user@1000.service/container.service
           ├─ 2870 /usr/bin/podman
           ├─80612 /usr/bin/slirp4netns --disable-host-loopback --mtu 65520 --enable-sandbox -
-enable-seccomp -c -e 3 -r 4 --netns-type=path /run/user/1000/netns/cni->
           ├─80614 /usr/bin/fuse-overlayfs -o
lowerdir=/home/user/.local/share/containers/storage/overlay/l/YJSPGXM2OCDZPLMLXJOW3N
RF6Q:/home/user/.local/share/contain>
           ├─80617 /usr/bin/conmon --api-version 1 -c
cbc75d6031508dfd3d78a74a03e4ace1732b51223e72a2ce4aa3bfe10a78e4fa -u
cbc75d6031508dfd3d78a74a03e4ace1732b51223e72>
           └─cbc75d6031508dfd3d78a74a03e4ace1732b51223e72a2ce4aa3bfe10a78e4fa
             └─80626 /usr/bin/coreutils --coreutils-prog-shebang=sleep /usr/bin/sleep 1d
```

You can check if the service is enabled using the **systemctl is-enabled container.service** command.

Verification steps

- List containers that are running or have exited:

```
# podman ps
CONTAINER ID  IMAGE                                  COMMAND  CREATED        STATUS
PORTS  NAMES
f20988d59920  registry.access.redhat.com/ubi8-minimal:latest  top      12 seconds ago  Up 11
seconds ago         funny_zhukovsky
```

NOTE

To stop **container.service**, enter:

```
# systemctl --user stop container.service
```

Additional resources

- **systemctl** man page
- Running containers with Podman and shareable systemd services
- Managing services with systemd

12.5. AUTO-STARTING PODS USING SYSTEMD

You can start multiple containers as systemd services. Note that the **systemctl** command should only be used on the pod and you should not start or stop containers individually via **systemctl**, as they are managed by the pod service along with the internal infra-container.

Procedure

1. Create an empty pod, for example named **systemd-pod**:

```
$ podman pod create --name systemd-pod
11d4646ba41b1fffa51c108cbdf97cfab3213f7bd9b3e1ca52fe81b90fed5577
```

2. Optional. List all pods:

```
$ podman pod ps
POD ID        NAME         STATUS   CREATED         # OF CONTAINERS  INFRA ID
11d4646ba41b  systemd-pod  Created  40 seconds ago  1                8a428b257111
11d4646ba41b1fffa51c108cbdf97cfab3213f7bd9b3e1ca52fe81b90fed5577
```

3. Create two containers in the empty pod. For example, to create **container0** and **container1** in **systemd-pod**:

```
$ podman create --pod systemd-pod --name container0
registry.access.redhat.com/ubi8 top
$ podman create --pod systemd-pod --name container1
registry.access.redhat.com/ubi8 top
```

4. Optional. List all pods and containers associated with them:

```
$ podman ps -a --pod
CONTAINER ID  IMAGE                                COMMAND  CREATED        STATUS
PORTS   NAMES              POD ID        PODNAME
24666f47d9b2  registry.access.redhat.com/ubi8:latest  top      3 minutes ago  Created
container0         3130f724e229  systemd-pod
56eb1bf0cdfe  k8s.gcr.io/pause:3.2                          4 minutes ago  Created
3130f724e229-infra  3130f724e229  systemd-pod
62118d170e43  registry.access.redhat.com/ubi8:latest  top      3 seconds ago  Created
container1         3130f724e229  systemd-pod
```

5. Generate the systemd unit file for the new pod:

```
$ podman generate systemd --files --name systemd-pod
/home/user1/pod-systemd-pod.service
/home/user1/container-container0.service
/home/user1/container-container1.service
```

Note that three systemd unit files are generated, one for the **systemd-pod** pod and two for the containers **container0** and **container1**.

6. Display **pod-systemd-pod.service** unit file:

```
$ cat pod-systemd-pod.service
# pod-systemd-pod.service
# autogenerated by Podman 3.3.1
# Wed Sep  8 20:49:17 CEST 2021

[Unit]
Description=Podman pod-systemd-pod.service
Documentation=man:podman-generate-systemd(1)
```

```
Wants=network-online.target
After=network-online.target
RequiresMountsFor=
Requires=container-container0.service container-container1.service
Before=container-container0.service container-container1.service

[Service]
Environment=PODMAN_SYSTEMD_UNIT=%n
Restart=on-failure
TimeoutStopSec=70
ExecStart=/usr/bin/podman start bcb128965b8e-infra
ExecStop=/usr/bin/podman stop -t 10 bcb128965b8e-infra
ExecStopPost=/usr/bin/podman stop -t 10 bcb128965b8e-infra
PIDFile=/run/user/1000/containers/overlay-
containers/1dfdcf20e35043939ea3f80f002c65c00d560e47223685dbc3230e26fe001b29/userda
ta/conmon.pid
Type=forking

[Install]
WantedBy=multi-user.target default.target
```

- The **Requires** line in the **[Unit]** section defines dependencies on **container-container0.service** and **container-container1.service** unit files. Both unit files will be activated.

- The **ExecStart** and **ExecStop** lines in the **[Service]** section start and stop the infra-container, respectively.

7. Display **container-container0.service** unit file:

```
$ cat container-container0.service
# container-container0.service
# autogenerated by Podman 3.3.1
# Wed Sep  8 20:49:17 CEST 2021

[Unit]
Description=Podman container-container0.service
Documentation=man:podman-generate-systemd(1)
Wants=network-online.target
After=network-online.target
RequiresMountsFor=/run/user/1000/containers
BindsTo=pod-systemd-pod.service
After=pod-systemd-pod.service

[Service]
Environment=PODMAN_SYSTEMD_UNIT=%n
Restart=on-failure
TimeoutStopSec=70
ExecStart=/usr/bin/podman start container0
ExecStop=/usr/bin/podman stop -t 10 container0
ExecStopPost=/usr/bin/podman stop -t 10 container0
PIDFile=/run/user/1000/containers/overlay-
containers/4bccd7c8616ae5909b05317df4066fa90a64a067375af5996fdef9152f6d51f5/userdat
a/conmon.pid
Type=forking
```

```
[Install]
WantedBy=multi-user.target default.target
```

- The **BindsTo** line line in the **[Unit]** section defines the dependency on the **pod-systemd-pod.service** unit file
- The **ExecStart** and **ExecStop** lines in the **[Service]** section start and stop the **container0** respectively.

8. Display **container-container1.service** unit file:

```
$ cat container-container1.service
```

9. Copy all the generated files to **$HOME/.config/systemd/user** for installing as a non-root user:

```
$ cp pod-systemd-pod.service container-container0.service container-container1.service $HOME/.config/systemd/user
```

10. Enable the service and start at user login:

```
$ systemctl enable --user pod-systemd-pod.service
Created symlink /home/user1/.config/systemd/user/multi-user.target.wants/pod-systemd-pod.service → /home/user1/.config/systemd/user/pod-systemd-pod.service.
Created symlink /home/user1/.config/systemd/user/default.target.wants/pod-systemd-pod.service → /home/user1/.config/systemd/user/pod-systemd-pod.service.
```

Note that the service stops at user logout.

Verification steps

- Check if the service is enabled:

```
$ systemctl is-enabled pod-systemd-pod.service
enabled
```

Additional resources

- **podman-create** man page
- **podman-generate-systemd** man page
- **systemctl** man page
- Running containers with Podman and shareable systemd services
- Making systemd services start at boot time

12.6. AUTO-UPDATING CONTAINERS USING PODMAN

The **podman auto-update** command allows you to automatically update containers according to their auto-update policy. The **podman auto-update** command updates services when the container image is updated on the registry. To use auto-updates, containers must be created with the **--label**

"io.containers.autoupdate=image" label and run in a systemd unit generated by **podman generate systemd --new** command.

Podman searches for running containers with the **"io.containers.autoupdate"** label set to **"image"** and communicates to the container registry. If the image has changed, Podman restarts the corresponding systemd unit to stop the old container and create a new one with the new image. As a result, the container, its environment, and all dependencies, are restarted.

Prerequisites

- The **container-tools** module is installed.

```
# yum module install -y container-tools
```

Procedure

1. Start a **myubi** container based on the **registry.access.redhat.com/ubi8/ubi-init** image:

```
# podman run --label "io.containers.autoupdate=image" \ --name myubi -dt
registry.access.redhat.com/ubi8/ubi-init top
bc219740a210455fa27deacc96d50a9e20516492f1417507c13ce1533dbdcd9d
```

2. Optional: List containers that are running or have exited:

```
# podman ps -a
CONTAINER ID  IMAGE                                COMMAND  CREATED         STATUS
PORTS   NAMES
76465a5e2933  registry.access.redhat.com/8/ubi-init:latest  top      24 seconds ago  Up 23
seconds ago          myubi
```

3. Generate a systemd unit file for the **myubi** container:

```
# podman generate systemd --new --files --name myubi /root/container-myubi.service
```

4. Copy unit files to **/usr/lib/systemd/system** for installing it as a root user:

```
# cp -Z ~/container-myubi.service /usr/lib/systemd/system
```

5. Reload systemd manager configuration:

```
# systemctl daemon-reload
```

6. Start and check the status of a container:

```
# systemctl start container-myubi.service
# systemctl status container-myubi.service
```

7. Auto-update the container:

```
# podman auto-update
```

Additional resources

- Improved Systemd Integration with Podman 2.0
- Running containers with Podman and shareable systemd services
- Making systemd services start at boot time

12.7. AUTO-UPDATING CONTAINERS USING SYSTEMD

As mentioned in section Auto-updating containers using Podman, you can update the container using the **podman auto-update** command. It integrates into custom scripts and can be invoked when needed. Another way to auto update the containers is to use the pre-installed **podman-auto-update.timer** and **podman-auto-update.service** systemd service. The **podman-auto-update.timer** can be configured to trigger auto updates at a specific date or time. The **podman-auto-update.service** can further be started by the **systemctl** command or be used as a dependency by other systemd services. As a result, auto updates based on time and events can be triggered in various ways to meet individual needs and use cases.

Prerequisites

- The **container-tools** module is installed.

```
# yum module install -y container-tools
```

Procedure

1. Display the **podman-auto-update.service** unit file:

```
# cat /usr/lib/systemd/system/podman-auto-update.service

[Unit]
Description=Podman auto-update service
Documentation=man:podman-auto-update(1)
Wants=network.target
After=network-online.target

[Service]
Type=oneshot
ExecStart=/usr/bin/podman auto-update

[Install]
WantedBy=multi-user.target default.target
```

2. Display the **podman-auto-update.timer** unit file:

```
# cat /usr/lib/systemd/system/podman-auto-update.timer

[Unit]
Description=Podman auto-update timer

[Timer]
OnCalendar=daily
Persistent=true
```

```
[Install]
WantedBy=timers.target
```

In this example, the **podman auto-update** command is launched daily at midnight.

3. Enable the **podman-auto-update.timer** service at system start:

```
# systemctl enable podman-auto-update.timer
```

4. Start the systemd service:

```
# systemctl start podman-auto-update.timer
```

5. Optional: List all timers:

```
# systemctl list-timers --all
NEXT                         LEFT      LAST                         PASSED       UNIT
ACTIVATES
Wed 2020-12-09 00:00:00 CET  9h left   n/a                          n/a          podman-auto-
update.timer     podman-auto-update.service
```

You can see that **podman-auto-update.timer** activates the **podman-auto-update.service**.

Additional resources

- Improved Systemd Integration with Podman 2.0
- Running containers with Podman and shareable systemd services
- Making systemd services start at boot time

CHAPTER 13. RUNNING SKOPEO, BUILDAH, AND PODMAN IN A CONTAINER

This chapter describes how you can run Skopeo, Buildah, and Podman in a container.

With Skopeo, you can inspect images on a remote registry without having to download the entire image with all its layers. You can also use Skopeo for copying images, signing images, syncing images, and converting images across different formats and layer compressions.

Buildah facilitates building OCI container images. With Buildah, you can create a working container, either from scratch or using an image as a starting point. You can create an image either from a working container or using the instructions in a **Containerfile**. You can mount and unmount a working container's root filesystem.

With Podman, you can manage containers and images, volumes mounted into those containers, and pods made from groups of containers. Podman is based on a **libpod** library for container lifecycle management. The **libpod** library provides APIs for managing containers, pods, container images, and volumes.

Reasons to run Buildah, Skopeo, and Podman in a container:

- **CI/CD system**:
  - **Podman and Skopeo**: You can run a CI/CD system inside of Kubernetes or use OpenShift to build your container images, and possibly distribute those images across different container registries. To integrate Skopeo into a Kubernetes workflow, you need to run it in a container.
  - **Buildah**: You want to build OCI/container images within a Kubernetes or OpenShift CI/CD systems that are constantly building images. Previously, people used a Docker socket to connect to the container engine and perform a **docker build** command. This was the equivalent of giving root access to the system without requiring a password which is not secure. For this reason, Red Hatrecommends using Buildah in a container.
- **Different versions**:
  - **All**: You are running an older OS on the host but you want to run the latest version of Skopeo, Buildah, or Podman. The solution is to run the container tools in a container. For example, this is useful for running the latest version of the container tools provided in Red Hat Enterprise Linux 8 on a Red Hat Enterprise Linux 7 container host which does not have access to the newest versions natively.
- **HPC environment**:
  - **All**: A common restriction in HPC environments is that non-root users are not allowed to install packages on the host. When you run Skopeo, Buildah, or Podman in a container, you can perform these specific tasks as a non-root user.

13.1. RUNNING SKOPEO IN A CONTAINER

This procedure demonstrates how to inspect a remote container image using Skopeo. Running Skopeo in a container means that the container root filesystem is isolated from the host root filesystem. To share or copy files between the host and container, you have to mount files and directories.

Prerequisites

- The **container-tools** module is installed.

```
# yum module install -y container-tools
```

Procedure

1. Log in to the registry.redhat.io registry:

```
$ podman login registry.redhat.io
Username: myuser@mycompany.com
Password: <password>
Login Succeeded!
```

2. Get the **registry.redhat.io/rhel8/skopeo** container image:

```
$ podman pull registry.redhat.io/rhel8/skopeo
```

3. Inspect a remote container image **registry.access.redhat.com/ubi8/ubi** using Skopeo:

```
$ podman run --rm registry.redhat.io/rhel8/skopeo \ skopeo inspect
docker://registry.access.redhat.com/ubi8/ubi
{
    "Name": "registry.access.redhat.com/ubi8/ubi",
    ...
    "Labels": {
        "architecture": "x86_64",
        ...
        "name": "ubi8",
        ...
        "summary": "Provides the latest release of Red Hat Universal Base Image 8.",
        "url":
"https://access.redhat.com/containers/#/registry.access.redhat.com/ubi8/images/8.2-347",
        ...
    },
    "Architecture": "amd64",
    "Os": "linux",
    "Layers": [
    ...
    ],
    "Env": [
        "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
        "container=oci"
    ]
}
```

The **--rm** option removes the **registry.redhat.io/rhel8/skopeo** image after the container exits.

Additional resources

- How to run skopeo in a container

13.2. RUNNING SKOPEO IN A CONTAINER USING CREDENTIALS

Working with container registries requires an authentication to access and alter data. Skopeo supports various ways to specify credentials.

With this approach you can specify credentials on the command line using the **--cred USERNAME[:PASSWORD]** option.

Prerequisites

- The **container-tools** module is installed.

```
# yum module install -y container-tools
```

Procedure

- Inspect a remote container image using Skopeo against a locked registry:

```
$ podman run --rm registry.redhat.io/rhel8/skopeo inspect --creds
$USER:$PASSWORD docker://$IMAGE
```

Additional resources

- How to run skopeo in a container

13.3. RUNNING SKOPEO IN A CONTAINER USING AUTHFILES

You can use an authentication file (authfile) to specify credentials. The **skopeo login** command logs into the specific registry and stores the authentication token in the authfile. The advantage of using authfiles is preventing the need to repeatedly enter credentials.

When running on the same host, all container tools such as Skopeo, Buildah, and Podman share the same authfile. When running Skopeo in a container, you have to either share the authfile on the host by volume-mounting the authfile in the container, or you have to reauthenticate within the container.

Prerequisites

- The **container-tools** module is installed.

```
# yum module install -y container-tools
```

Procedure

- Inspect a remote container image using Skopeo against a locked registry:

```
$ podman run --rm -v $AUTHFILE:/auth.json registry.redhat.io/rhel8/skopeo inspect
docker://$IMAGE
```

  The **-v $AUTHFILE:/auth.json** option volume-mounts an authfile at /auth.json within the container. Skopeo can now access the authentication tokens in the authfile on the host and get secure access to the registry.

Other Skopeo commands work similarly, for example:

- Use the **skopeo-copy** command to specify credentials on the command line for the source and destination image using the **--source-creds** and **--dest-creds** options. It also reads the **/auth.json** authfile.
- If you want to specify separate authfiles for the source and destination image, use the **--source-authfile** and **--dest-authfile** options and volume-mount those authfiles from the host into the container.

Additional resources

- How to run skopeo in a container

13.4. COPYING CONTAINER IMAGES TO OR FROM THE HOST

Skopeo, Buildah, and Podman share the same local container-image storage. If you want to copy containers to or from the host container storage, you need to mount it into the Skopeo container.

NOTE

The path to the host container storage differs between root (**/var/lib/containers/storage**) and non-root users (**$HOME/.local/share/containers/storage**).

Prerequisites

- The **container-tools** module is installed.

```
# yum module install -y container-tools
```

Procedure

1. Copy the **registry.access.redhat.com/ubi8/ubi** image into your local container storage:

```
$ podman run --privileged --rm -v
$HOME/.local/share/containers/storage:/var/lib/containers/storage \
registry.redhat.io/rhel8/skopeo skopeo copy \
docker://registry.access.redhat.com/ubi8/ubi containers-
storage:registry.access.redhat.com/ubi8/ubi
```

   - The **--privileged** option disables all security mechanisms. Red Hat recommends only using this option in trusted environments.
   - To avoid disabling security mechanisms, export the images to a tarball or any other path-based image transport and mount them in the Skopeo container:
     - **$ podman save --format oci-archive -o oci.tar $IMAGE**
     - **$ podman run --rm -v oci.tar:/oci.tar registry.redhat.io/rhel8/skopeo copy oci-archive:/oci.tar $DESTINATION**

2. Optional: List images in local storage:

```
$ podman images
REPOSITORY                              TAG     IMAGE ID      CREATED       SIZE
registry.access.redhat.com/ubi8/ubi     latest  ecbc6f53bba0  8 weeks ago   211 MB
```

Additional resources

- How to run skopeo in a container

13.5. RUNNING BUILDAH IN A CONTAINER

The procedure demonstrates how to run Buildah in a container and create a working container based on an image.

Prerequisites

- The **container-tools** module is installed.

```
# yum module install -y container-tools
```

Procedure

1. Log in to the registry.redhat.io registry:

```
$ podman login registry.redhat.io
Username: myuser@mycompany.com
Password: <password>
Login Succeeded!
```

2. Pull and run the **registry.redhat.io/rhel8/buildah** image:

```
# podman run --rm --device /dev/fuse -it \ registry.redhat.io/rhel8/buildah /bin/bash
```

   - The **--rm** option removes the **registry.redhat.io/rhel8/buildah** image after the container exits.
   - The **--device** option adds a host device to the container.

3. Create a new container using a **registry.access.redhat.com/ubi8** image:

```
# buildah from registry.access.redhat.com/ubi8
...
ubi8-working-container
```

4. Run the **ls** / command inside the **ubi8-working-container** container:

```
# buildah run --isolation=chroot ubi8-working-container ls /
bin  boot  dev  etc  home  lib  lib64  lost+found  media  mnt  opt  proc  root  run  sbin  srv
```

5. Optional: List all images in a local storage:

```
# buildah images
REPOSITORY                      TAG      IMAGE ID       CREATED       SIZE
registry.access.redhat.com/ubi8   latest   ecbc6f53bba0   5 weeks ago   211 MB
```

6. Optional: List the working containers and their base images:

```
# buildah containers
CONTAINER ID  BUILDER  IMAGE ID     IMAGE NAME                       CONTAINER NAME
0aaba7192762     *     ecbc6f53bba0 registry.access.redhat.com/ub... ubi8-working-container
```

7. Optional: Push the **registry.access.redhat.com/ubi8** image to the a local registry located on **registry.example.com**:

```
# buildah push ecbc6f53bba0 registry.example.com:5000/ubi8/ubi
```

Additional resources

- Best practices for running Buildah in a container

13.6. PRIVILEGED AND UNPRIVILEGED PODMAN CONTAINERS

By default, Podman containers are unprivileged and cannot, for example, modify parts of the operating system on the host. This is because by default a container is only allowed limited access to devices.

The following list emphasizes important properties of privileged containers. You can run the privileged container using the **podman run --privileged *<image_name>*** command.

- A privileged container is given the same access to devices as the user launching the container.
- A privileged container disables the security features that isolate the container from the host. Dropped Capabilities, limited devices, read-only mount points, Apparmor/SELinux separation, and Seccomp filters are all disabled.
- A privileged container cannot have more privileges than the account that launched them.

Additional resources

- How to use the --privileged flag with container engines
- **podman-run** man page

13.7. RUNNING PODMAN WITH EXTENDED PRIVILEGES

If you cannot run your workloads in a rootless environment, you need to run these workloads as a root user. Running a container with extended privileges should be done judiciously, because it disables all security features.

Prerequisites

- The **container-tools** module is installed.

```
# yum module install -y container-tools
```

Procedure

- Run the Podman container in the Podman container:

```
$ podman run --privileged --name=privileged_podman \
registry.access.redhat.com//podman podman run ubi8 echo hello
Resolved "ubi8" as an alias (/etc/containers/registries.conf.d/001-rhel-shortnames.conf)
Trying to pull registry.access.redhat.com/ubi8:latest...
...
Storing signatures
hello
```

- Run the outer container named **privileged_podman** based on the **registry.access.redhat.com/ubi8/podman** image.
- The **--privileged** option disables the security features that isolate the container from the host.
- Run **podman run ubi8 echo hello** command to create the inner container based on the **ubi8** image.
- Notice that the **ubi8** short image name was resolved as an alias. As a result, the **registry.access.redhat.com/ubi8:latest** image is pulled.

Verification

- List all containers:

```
$ podman ps -a
CONTAINER ID  IMAGE                                COMMAND               CREATED         STATUS
PORTS   NAMES
52537876caf4  registry.access.redhat.com/ubi8/podman  podman run ubi8 e...  30
seconds ago     Exited (0) 13 seconds ago               privileged_podman
```

Additional resources

- How to use Podman inside of a container
- **podman-run** man page

13.8. RUNNING PODMAN WITH LESS PRIVILEGES

You can run two nested Podman containers without the **--privileged** option. Running the container without the **--privileged** option is a more secure option.

This can be useful when you want to try out different versions of Podman in the most secure way possible.

Prerequisites

- The **container-tools** module is installed.

```
# yum module install -y container-tools
```

Procedure

- Run two nested containers:

```
$ podman run --name=unprivileged_podman --security-opt label=disable \ --user
podman --device /dev/fuse \ registry.access.redhat.com/ubi8/podman \ podman run
ubi8 echo hello
```

- Run the outer container named **unprivileged_podman** based on the **registry.access.redhat.com/ubi8/podman** image.
- The **--security-opt label=disable** option disables SELinux separation on the host Podman. SELinux does not allow containerized processes to mount all of the file systems required to run inside a container.
- The **--user podman** option automatically causes the Podman inside the outer container to run within the user namespace.
- The **--device /dev/fuse** option uses the **fuse-overlayfs** package inside the container. This option adds **/dev/fuse** to the outer container, so that Podman inside the container can use it.
- Run **podman run ubi8 echo hello** command to create the inner container based on the **ubi8** image.
- Notice that the ubi8 short image name was resolved as an alias. As a result, the **registry.access.redhat.com/ubi8:latest** image is pulled.

Verification

- List all containers:

```
$ podman ps -a
CONTAINER ID  IMAGE                         COMMAND           CREATED          STATUS
PORTS   NAMES
a47b26290f43  podman run ubi8 e...  30 seconds ago    Exited (0) 13 seconds ago
unprivileged_podman
```

13.9. BUILDING A CONTAINER INSIDE A PODMAN CONTAINER

This procedure shows how to run a container in a container using Podman. This example shows how to use Podman to build and run another container from within this container. The container will run "Moon-buggy", a simple text-based game.

Prerequisites

- The **container-tools** module is installed.

```
# yum module install -y container-tools
```

- You are logged in to the registry.redhat.io registry:

```
# podman login registry.redhat.io
```

Procedure

1. Run the container based on **registry.redhat.io/rhel8/podman** image:

```
# podman run --privileged --name podman_container -it \
registry.redhat.io/rhel8/podman /bin/bash
```

   - Run the outer container named **podman_container** based on the **registry.redhat.io/rhel8/podman** image.
   - The **--it** option specifies that you want to run an interactive bash shell within a container.
   - The **--privileged** option disables the security features that isolate the container from the host.

2. Create a **Containerfile** inside the **podman_container** container:

```
# vi Containerfile
FROM registry.access.redhat.com/ubi8/ubi
RUN yum install -y https://dl.fedoraproject.org/pub/epel/epel-release-latest-8.noarch.rpm
RUN yum -y install moon-buggy && yum clean all
CMD ["/usr/bin/moon-buggy"]
```

   The commands in the **Containerfile** cause the following build command to:

   - Build a container from the **registry.access.redhat.com/ubi8/ubi** image.
   - Install the **epel-release-latest-8.noarch.rpm** package.
   - Install the **moon-buggy** package.
   - Set the container command.

3. Build a new container image named **moon-buggy** using the **Containerfile**:

```
# podman build -t moon-buggy .
```

4. Optional: List all images:

```
# podman images
REPOSITORY                 TAG      IMAGE ID      CREATED        SIZE
localhost/moon-buggy  latest  c97c58abb564  13 seconds ago  1.67 GB
registry.access.redhat.com/ubi8/ubi latest 4199acc83c6a  132seconds ago 213 MB
```

5. Run a new container based on a **moon-buggy** container:

```
# podman run -it --name moon moon-buggy
```

6. Optional: Tag the **moon-buggy** image:

```
# podman tag moon-buggy registry.example.com/moon-buggy
```

7. Optional: Push the **moon-buggy** image to the registry:

```
# podman push registry.example.com/moon-buggy
```

Additional resources

- Technology preview: Running a container inside a container

CHAPTER 14. BUILDING CONTAINER IMAGES WITH BUILDAH

Buildah facilitates building OCI container images that meet the OCI Runtime Specification. With Buildah, you can create a working container, either from scratch or using an image as a starting point. You can create an image either from a working container or using the instructions in a **Containerfile**. You can mount and unmount a working container's root filesystem.

14.1. THE BUILDAH TOOL

Using Buildah is different from building images with the docker command in the following ways:

No Daemon

Buildah requires no container runtime.

Base image or scratch

You can build an image based on another container or start with an empty image (scratch).

Build tools are external

Buildah does not include build tools within the image itself. As a result, Buildah:

- Reduces the size of built images.
- Increases security of images by excluding software (for example gcc, make, and yum) from the resulting image.
- Allows to transport the images using fewer resources because of the reduced image size.

Compatibility

Buildah supports building container images with Dockerfiles allowing for an easy migration from Docker to Buildah.

NOTE

The default location Buildah uses for container storage is the same as the location the CRI-O container engine uses for storing local copies of images. As a result, the images pulled from a registry by either CRI-O or Buildah, or committed by the buildah command, are stored in the same directory structure. However, even though CRI-O and Buildah are currently able to share images, they cannot share containers.

Additional resources

- Buildah - a tool that facilitates building Open Container Initiative (OCI) container images
- Buildah Tutorial 1: Building OCI container images
- Buildah Tutorial 2: Using Buildah with container registries
- Building with Buildah: Dockerfiles, command line, or scripts
- How rootless Buildah works: Building containers in unprivileged environments

14.2. INSTALLING BUILDAH

Install the Buildah tool using the **yum** command.

Procedure

- Install the Buildah tool:

```
# yum -y install buildah
```

Verification

- Display the help message:

```
# buildah -h
```

14.3. GETTING IMAGES WITH BUILDAH

Use the **buildah from** command to create a new working container from scratch or based on a specified image as a starting point.

Procedure

- Create a new working container based on the **registry.redhat.io/ubi8/ubi** image:

```
# buildah from registry.access.redhat.com/ubi8/ubi
Getting image source signatures
Copying blob…
Writing manifest to image destination
Storing signatures
ubi-working-container
```

Verification

1. List all images in local storage:

```
# buildah images
REPOSITORY                              TAG      IMAGE ID       CREATED       SIZE
registry.access.redhat.com/ubi8/ubi     latest   272209ff0ae5   2 weeks ago   234 MB
```

2. List the working containers and their base images:

```
# buildah containers
CONTAINER ID  BUILDER  IMAGE ID     IMAGE NAME                       CONTAINER NAME
01eab9588ae1     *     272209ff0ae5 registry.access.redhat.com/ub... ubi-working-container
```

Additional resources

- **buildah-from** man page
- **buildah-images** man page
- **buildah-containers** man page

14.4. RUNNING COMMANDS INSIDE OF THE CONTAINER

Use the **buildah run** command to execute a command from the container.

Prerequisites

- A pulled image is available on the local system.

Procedure

- Display the operating system version:

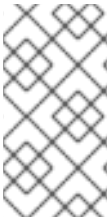
```
# buildah run ubi-working-container cat /etc/redhat-release
Red Hat Enterprise Linux release 8.4 (Ootpa)
```

Additional resources

- **buildah-run** man page

14.5. BUILDING AN IMAGE FROM A CONTAINERFILE WITH BUILDAH

Use the **buildah bud** command to build an image using instructions from a **Containerfile**.



NOTE

The **buildah bud** command uses a **Containerfile** if found in the context directory, if it is not found the **buildah bud** command uses a **Dockerfile**; otherwise any file can be specified with the **--file** option. The available commands that are usable inside a **Containerfile** and a **Dockerfile** are equivalent.

Procedure

1. Create a **Containerfile**:

```
# cat Containerfile
FROM registry.access.redhat.com/ubi8/ubi
ADD myecho /usr/local/bin
ENTRYPOINT "/usr/local/bin/myecho"
```

2. Create a **myecho** script:

```
# cat myecho
echo "This container works!"
```

3. Change the access permissions of **myecho** script:

```
# chmod 755 myecho
```

4. Build the **myecho** image using **Containerfile** in the current directory:

```
# buildah bud -t myecho .
STEP 1: FROM registry.access.redhat.com/ubi8/ubi
STEP 2: ADD myecho /usr/local/bin
STEP 3: ENTRYPOINT "/usr/local/bin/myecho"
```

```
STEP 4: COMMIT myecho
...
Storing signatures
```

Verification

1. List all images:

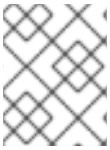
```
# buildah images
REPOSITORY                TAG      IMAGE ID       CREATED              SIZE
localhost/myecho          latest   b28cd00741b3   About a minute ago   234 MB
```

2. Run the **myecho** container based on the **localhost/myecho** image:

```
# podman run --name=myecho localhost/myecho
This container works!
```

3. List all containers:

```
# podman ps -a
0d97517428d  localhost/myecho                    12 seconds ago  Exited (0) 13
seconds ago          myecho
```



NOTE

You can use the **podman history** command to display the information about each layer used in the image.

Additional resources

- **buildah-bud** man page

14.6. INSPECTING CONTAINERS AND IMAGES WITH BUILDAH

Use the **buildah inspect** command to display information about a container or image.

Prerequisites

- An image was built using instructions from Containerfile. For details, see section Building an image from a Containerfile with Buildah.

Procedure

- Inspect the image:
  - To inspect the myecho image, enter:

```
# buildah inspect localhost/myecho
 {
    "Type": "buildah 0.0.1",
    "FromImage": "localhost/myecho:latest",
    "FromImageID":
"b28cd00741b38c92382ee806e1653eae0a56402bcd2c8d31bdcd36521bc267a4",
```

```
    "FromImageDigest":
"sha256:0f5b06cbd51b464fabe93ce4fe852a9038cdd7c7b7661cd7efef8f9ae8a59585",
    "Config":
    ...
     "Entrypoint": [
                "/bin/sh",
                "-c",
                "\"/usr/local/bin/myecho\""
            ],
    ...
}
```

- To inspect the working container from the **myecho** image:

  i. Create a working container based on the **localhost/myecho** image:

  ```
  # buildah from localhost/myecho
  ```

  ii. Inspect the **myecho-working-container** container:

  ```
  # buildah inspect ubi-working-container
  {
      "Type": "buildah 0.0.1",
      "FromImage": "registry.access.redhat.com/ubi8/ubi:latest",
      "FromImageID":
  "272209ff0ae5fe54c119b9c32a25887e13625c9035a1599feba654aa7638262d",
      "FromImageDigest":
  "sha256:77623387101abefbf83161c7d5a0378379d0424b2244009282acb39d42f1fe13
  ",
      "Config":
      ...
  "Container": "ubi-working-container",
  "ContainerID":
  "01eab9588ae1523746bb706479063ba103f6281ebaeeccb5dc42b70e450d5ad0",
  "ProcessLabel": "system_u:system_r:container_t:s0:c162,c1000",
  "MountLabel": "system_u:object_r:container_file_t:s0:c162,c1000",
  ...
  }
  ```

Additional resources

- **buildah-inspect** man page

14.7. MODIFYING A CONTAINER USING BUILDAH MOUNT

Use the **buildah inspect** command to display information about a container or image.

Prerequisites

- An image built using instructions from Containerfile. For details, see section Building an image from a Containerfile with Buildah.

Procedure

1. Create a working container based on the **registry.access.redhat.com/ubi8/ubi** image and save the name of the container to the **mycontainer** variable:

```
# mycontainer=$(buildah from localhost/myecho)

# echo $mycontainer
myecho-working-container
```

2. Mount the **myecho-working-container** container and save the mount point path to the **mymount** variable:

```
# mymount=$(buildah mount $mycontainer)

# echo $mymount
/var/lib/containers/storage/overlay/c1709df40031dda7c49e93575d9c8eebcaa5d8129033a58e5
b6a95019684cc25/merged
```

3. Modify the **myecho** script and make it executable:

```
# echo 'echo "We modified this container."' >> $mymount/usr/local/bin/myecho
# chmod +x $mymount/usr/local/bin/myecho
```

4. Create the **myecho2** image from the **myecho-working-container** container:

```
# buildah commit $mycontainer containers-storage:myecho2
```

Verification

1. List all images in local storage:

```
# buildah images
REPOSITORY                    TAG      IMAGE ID       CREATED          SIZE
docker.io/library/myecho2     latest   4547d2c3e436   4 minutes ago    234 MB
localhost/myecho              latest   b28cd00741b3   56 minutes ago   234 MB
```

2. Run the **myecho2** container based on the **docker.io/library/myecho2** image:

```
# podman run --name=myecho2 docker.io/library/myecho2
This container works!
We even modified it.
```

Additional resources

- **buildah-mount** man page
- **buildah-commit** man page

14.8. MODIFYING A CONTAINER USING BUILDAH COPY AND BUILDAH CONFIG

Use **buildah copy** command to copy files to a container without mounting it. You can then configure the container using the **buildah config** command to run the script you created by default.

Prerequisites

- An image built using instructions from Containerfile. For details, see section Building an image from a Containerfile with Buildah.

Procedure

1. Create a script named **newecho** and make it executable:

```
# cat newecho
echo "I changed this container"
# chmod 755 newecho
```

2. Create a new working container:

```
# buildah from myecho:latest
myecho-working-container-2
```

3. Copy the newecho script to **/usr/local/bin** directory inside the container:

```
# buildah copy myecho-working-container-2 newecho /usr/local/bin
```

4. Change the configuration to use the **newecho** script as the new entrypoint:

```
# buildah config --entrypoint "/bin/sh -c /usr/local/bin/newecho" myecho-working-container-2
```

5. Optional. Run the **myecho-working-container-2** container whixh triggers the **newecho** script to be executed:

```
# buildah run myecho-working-container-2 -- sh -c '/usr/local/bin/newecho'
I changed this container
```

6. Commit the **myecho-working-container-2** container to a new image called **mynewecho**:

```
# buildah commit myecho-working-container-2 containers-storage:mynewecho
```

Verification

- List all images in local storage:

```
# buildah images
REPOSITORY                      TAG      IMAGE ID       CREATED          SIZE
docker.io/library/mynewecho     latest   fa2091a7d8b6   8 seconds ago   234 MB
```

Additional resources

- **buildah-copy** man page
- **buildah-config** man page
- **buildah-commit** man page

- **buildah-run** man page

14.9. CREATING IMAGES FROM SCRATCH WITH BUILDAH

Instead of starting with a base image, you can create a new container that holds only a minimal amount of container metadata.

When creating an image from scratch container, consider:

- You can copy the executable with no dependencies into the scratch image and make a few configuration settings to get a minimal container to work.
- You must initialize an RPM database and add a release package in the container to use tools like **yum** or **rpm**.
- If you add a lot of packages, consider using the standard UBI or minimal UBI images instead of scratch images.

Procedure

This procedure adds a web service httpd to a container and configures it to run.

1. Create an empty container:

```
# buildah from scratch
working-container
```

2. Mount the **working-container** container and save the mount point path to the **scratchmnt** variable:

```
# scratchmnt=$(buildah mount working-container)


# echo $scratchmnt
/var/lib/containers/storage/overlay/be2eaecf9f74b6acfe4d0017dd5534fde06b2fa8de9ed875691f6ccc791c1836/merged
```

3. Initialize an RPM database within the scratch image and add the **redhat-release** package:

```
# yum install -y --releasever=8 --installroot=$scratchmnt redhat-release
```

4. Install the **httpd** service to the **scratch** directory:

```
# yum install -y --setopt=reposdir=/etc/yum.repos.d \ --installroot=$scratchmnt \ --setopt=cachedir=/var/cache/dnf httpd
```

5. Create the **$scratchmnt/var/www/html/index.html** file:

```
# mkdir -p $scratchmnt/var/www/html
# echo "Your httpd container from scratch works!" > $scratchmnt/var/www/html/index.html
```

6. Configure **working-container** to run the **httpd** daemon directly from the container:

```
# buildah config --cmd "/usr/sbin/httpd -DFOREGROUND" working-container
# buildah config --port 80/tcp working-container
# buildah commit working-container localhost/myhttpd:latest
```

Verification

1. List all images in local storage:

```
# podman images
REPOSITORY                  TAG     IMAGE ID      CREATED         SIZE
localhost/myhttpd           latest  08da72792f60  2 minutes ago   121 MB
```

2. Run the **localhost/myhttpd** image and configure port mappings between the container and the host system:

```
# podman run -p 8080:80 -d --name myhttpd 08da72792f60
```

3. Test the web server:

```
# curl localhost:8080
Your httpd container from scratch works!
```

Additional resources

- **buildah-config** man page
- **buildah-commit** man page

14.10. PUSHING CONTAINERS TO A PRIVATE REGISTRY

Use **buildah push** command to push an image from local storage to a public or private repository.

Prerequisites

- An image was built using instructions from Containerfile. For details, see section Building an image from a Containerfile with Buildah.

Procedure

1. Create the local registry on your machine:

```
# podman run -d -p 5000:5000 registry:2
```

2. Push the **myecho:latest** image to the **localhost** registry:

```
#  buildah push --tls-verify=false myecho:latest localhost:5000/myecho:latest
Getting image source signatures
Copying blob sha256:e4efd0...
...
Writing manifest to image destination
Storing signatures
```

Verification

1. List all images in the **localhost** repository:

```
# curl http://localhost:5000/v2/_catalog
{"repositories":["myecho2]}


# curl http://localhost:5000/v2/myecho2/tags/list
{"name":"myecho","tags":["latest"]}
```

2. Inspect the **docker://localhost:5000/myecho:latest** image:

```
# skopeo inspect --tls-verify=false docker://localhost:5000/myecho:latest | less
{
    "Name": "localhost:5000/myecho",
    "Digest": "sha256:8999ff6050...",
    "RepoTags": [
        "latest"
    ],
    "Created": "2021-06-28T14:44:05.919583964Z",
    "DockerVersion": "",
    "Labels": {
        "architecture": "x86_64",
        "authoritative-source-url": "registry.redhat.io",
    ...
}
```

3. Pull the **localhost:5000/myecho** image:

```
# podman pull --tls-verify=false localhost:5000/myecho2
# podman run localhost:5000/myecho2
This container works!
```

Additional resources

- **buildah-push** man page

14.11. PUSHING CONTAINERS TO THE DOCKER HUB

Use your Docker Hub credentials to push and pull images from the Docker Hub with the **buildah** command.

Prerequisites

- An image built using instructions from Containerfile. For details, see section Building an image from a Containerfile with Buildah.

Procedure

1. Push the **docker.io/library/myecho:latest** to your Docker Hub. Replace ***username*** and ***password*** with your Docker Hub credentials:

```
# buildah push --creds username:password \ docker.io/library/myecho:latest
docker://testaccountXX/myecho:latest
```

Verification

- Get and run the **docker.io/testaccountXX/myecho:latest** image:
  - Using Podman tool:

```
# podman run docker.io/testaccountXX/myecho:latest
This container works!
```

  - Using Buildah and Podman tools:

```
# buildah from docker.io/testaccountXX/myecho:latest
myecho2-working-container-2
# podman run myecho-working-container-2
```

Additional resources

- **buildah-push** man page

14.12. REMOVING IMAGES WITH BUILDAH

Use the **buildah rmi** command to remove locally stored container images. You can remove an image by its ID or name.

Procedure

1. List all images on your local system:

```
# *buildah images
REPOSITORY                                TAG      IMAGE ID       CREATED          SIZE
localhost/johndoe/webserver *             latest   dc5fcc610313   46 minutes ago   263 MB
docker.io/library/mynewecho               latest   fa2091a7d8b6   17 hours ago     234 MB
docker.io/library/myecho2                 latest   4547d2c3e436   6 days ago       234 MB
localhost/myecho                          latest   b28cd00741b3   6 days ago       234 MB
localhost/ubi-micro-httpd                 latest   c6a7678c4139   12 days ago      152 MB
registry.access.redhat.com/ubi8/ubi       latest   272209ff0ae5   3 weeks ago      234 MB
```

2. Remove the **localhost/myecho** image:

```
# buildah rmi localhost/myecho
```

- To remove multiple images:

```
# buildah rmi docker.io/library/mynewecho docker.io/library/myecho2
```

- To remove all images from your system:

```
# buildah rmi -a
```

- To remove images that have multiple names (tags) associated with them, add the **-f** option to remove them:

```
# buildah rmi -f localhost/ubi-micro-httpd
```

Verification

- Ensure that images were removed:

```
# buildah images
```

Additional resources

- **buildah-rmi** man page

14.13. REMOVING CONTAINERS WITH BUILDAH

Use the **buildah rm** command to remove containers. You can specify containers for removal with the container ID or name.

Prerequisites

- At least one container has been stopped.

Procedure

1. List all containers:

```
# buildah containers
CONTAINER ID  BUILDER  IMAGE ID     IMAGE NAME                       CONTAINER NAME
05387e29ab93     *     c37e14066ac7 docker.io/library/myecho:latest  myecho-working-
container
```

2. Remove the myecho-working-container container:

```
# buildah rm myecho-working-container
05387e29ab93151cf52e9c85c573f3e8ab64af1592b1ff9315db8a10a77d7c22
```

Verification

- Ensure that containers were removed:

```
# buildah containers
```

Additional resources

- **buildah-rm** man page

CHAPTER 15. MONITORING CONTAINERS

This chapter focuses on useful Podman commands that allow you to manage a Podman environment, including determining the health of the container, displaying system and pod information, and monitoring Podman events.

15.1. USING A HEALTH CHECK ON A CONTAINER

You can use the health check to determine the health or readiness of the process running inside the container.

If the health check succeeds, the container is marked as "healthy"; otherwise, it is "unhealthy". You can compare a health check with running the **podman exec** command and examining the exit code. The zero exit value means that the container is "healthy".

Health checks can be set when building an image using the **HEALTHCHECK** instruction in the **Containerfile** or when creating the container on the command line. You can display the health-check status of a container using the **podman inspect** or **podman ps** commands.

A health check consists of six basic components:

- Command
- Retries
- Interval
- Start-period
- Timeout
- Container recovery

The description of health check components follows:

Command (--health-cmd option)

Podman executes the command inside the target container and waits for the exit code.

The other five components are related to the scheduling of the health check and they are optional.

Retries (--health-retries option)

Defines the number of consecutive failed health checks that need to occur before the container is marked as "unhealthy". A successful health check resets the retry counter.

Interval (--health-interval option)

Describes the time between running the health check command. Note that small intervals cause your system to spend a lot of time running health checks. The large intervals cause struggles with catching time outs.

Start-period (--health-start-period option)

Describes the time between when the container starts and when you want to ignore health check failures.

Timeout (--health-timeout option)

Describes the period of time the health check must complete before being considered unsuccessful.

NOTE

The values of the Retries, Interval, and Start-period components are time durations, for example “30s” or “1h15m”. Valid time units are "ns," "us," or "µs", "ms," "s," "m," and "h".

Container recovery (--health-on-failure option)

Determines which actions to perform when the status of a container is unhealthy. When the application fails, Podman restarts it automatically to provide robustness. The **--health-on-failure** option supports four actions:

- **none**: Take no action, this is the default action.
- **kill**: Kill the container.
- **restart**: Restart the container.
- **stop**: Stop the container.

NOTE

The **--health-on-failure** option is available in Podman version 4.2 and later.

WARNING

Do not combine the **restart** action with the **--restart** option. When running inside of a systemd unit, consider using the **kill** or **stop** action instead, to make use of systemd’s restart policy.

Health checks run inside the container. Health checks only make sense if you know what the health state of the service is and can differentiate between a successful and unsuccessful health check.

Additional resources

- **podman-healthcheck** man page
- **podman-run** man page
- Podman at the edge: Keeping services alive with custom healthcheck actions
- Monitoring container vitality and availability with Podman

15.2. PERFORMING A HEALTH CHECK USING THE COMMAND LINE

You can set a health check when creating the container on the command line.

Procedure

1. Define a health check:

```
$ podman run -dt --name=hc-container -p 8080:8080 --health-cmd='curl
http://localhost:8080 || exit 1' --health-interval=0
registry.access.redhat.com/ubi8/httpd-24
```

- The **--health-cmd** option sets a health check command for the container.
- The **--health-interval=0** option with 0 value indicates that you want to run the health check manually.

2. Check the health status of the **hc-container** container:
   - Using the **podman inspect** command:

```
$ podman inspect --format='{{json .State.Health.Status}}' hc-container
healthy
```

   - Using the **podman ps** command:

```
$ podman ps
CONTAINER ID  IMAGE                            COMMAND               CREATED      STATUS
PORTS       NAMES
a680c6919fe  localhost/hc-container:latest  /usr/bin/run-http...  2 minutes ago  Up 2
minutes (healthy) hc-container
```

   - Using the **podman healthcheck run** command:

```
$ podman healthcheck run hc-container
healthy
```

Additional resources

- **podman-healthcheck** man page
- **podman-run** man page
- Podman at the edge: Keeping services alive with custom healthcheck actions
- Monitoring container vitality and availability with Podman

15.3. PERFORMING A HEALTH CHECK USING A CONTAINERFILE

You can set a health check by using the **HEALTHCHECK** instruction in the **Containerfile**.

Procedure

1. Create a **Containerfile**:

```
$ cat Containerfile
FROM registry.access.redhat.com/ubi8/httpd-24
EXPOSE 8080
HEALTHCHECK CMD curl http://localhost:8080 || exit 1
```

NOTE

The **HEALTHCHECK** instruction is supported only for the **docker** image format. For the **oci** image format, the instruction is ignored.

2. Build the container and add an image name:

```
$ podman build --format=docker -t hc-container .
STEP 1/3: FROM registry.access.redhat.com/ubi8/httpd-24
STEP 2/3: EXPOSE 8080
--> 5aea97430fd
STEP 3/3: HEALTHCHECK CMD curl http://localhost:8080 || exit 1
COMMIT health-check
Successfully tagged localhost/health-check:latest
a680c6919fe6bf1a79219a1b3d6216550d5a8f83570c36d0dadfee1bb74b924e
```

3. Run the container:

```
$ podman run -dt --name=hc-container localhost/hc-container
```

4. Check the health status of the **hc-container** container:

- Using the **podman inspect** command:

```
$ podman inspect --format='{{json .State.Health.Status}}' hc-container
healthy
```

- Using the **podman ps** command:

```
$ podman ps
CONTAINER ID  IMAGE                COMMAND               CREATED      STATUS
PORTS       NAMES
a680c6919fe  localhost/hc-container:latest  /usr/bin/run-http...  2 minutes ago  Up 2
minutes (healthy) hc-container
```

- Using the **podman healthcheck run** command:

```
$ podman healthcheck run hc-container
healthy
```

Additional resources

- **podman-healthcheck** man page
- **podman-run** man page
- Podman at the edge: Keeping services alive with custom healthcheck actions
- Monitoring container vitality and availability with Podman

15.4. DISPLAYING PODMAN SYSTEM INFORMATION

The **podman system** command allows you to manage the Podman systems. This section provides information on how to display Podman system information.

Procedure

- Display Podman system information:
  - To show Podman disk usage, enter:

```
$ podman system df
TYPE           TOTAL       ACTIVE      SIZE        RECLAIMABLE
Images         3           2           1.085GB     233.4MB (0%)
Containers     2           0           28.17kB     28.17kB (100%)
Local Volumes  3           0           0B          0B (0%)
```

  - To show detailed information on space usage, enter:

```
$ podman system df -v
Images space usage:

REPOSITORY                            TAG         IMAGE ID      CREATED     SIZE
SHARED SIZE  UNIQUE SIZE  CONTAINERS
registry.access.redhat.com/ubi8       latest      b1e63aaae5cf  13 days     233.4MB
233.4MB      0B           0
registry.access.redhat.com/ubi8/httpd-24  latest      0d04740850e8  13 days     461.5MB
0B           461.5MB      1
registry.redhat.io/rhel8/podman       latest      dce10f591a2d  13 days     390.6MB
233.4MB      157.2MB      1

Containers space usage:

CONTAINER ID  IMAGE         COMMAND                  LOCAL VOLUMES  SIZE
CREATED     STATUS      NAMES
311180ab99fb  0d04740850e8  /usr/bin/run-httpd       0              28.17kB     16 hours
exited      hc1
bedb6c287ed6  dce10f591a2d  podman run ubi8 echo hello  0           0B          11 hours
configured  dazzling_tu

Local Volumes space usage:

VOLUME NAME                                                 LINKS       SIZE
76de0efa83a3dae1a388b9e9e67161d28187e093955df185ea228ad0b3e435d0  0
0B
8a1b4658aecc9ff38711a2c7f2da6de192c5b1e753bb7e3b25e9bf3bb7da8b13  0
0B
d9cab4f6ccbcf2ac3cd750d2efff9d2b0f29411d430a119210dd242e8be20e26  0           0B
```

  - To display information about the host, current storage stats, and build of Podman, enter:

```
$ podman system info
host:
  arch: amd64
  buildahVersion: 1.22.3
  cgroupControllers: []
  cgroupManager: cgroupfs
```

```
  cgroupVersion: v1
  conmon:
    package: conmon-2.0.29-1.module+el8.5.0+12381+e822eb26.x86_64
    path: /usr/bin/conmon
    version: 'conmon version 2.0.29, commit:
7d0fa63455025991c2fc641da85922fde889c91b'
  cpus: 2
  distribution:
    distribution: '"rhel"'
    version: "8.5"
  eventLogger: file
  hostname: localhost.localdomain
  idMappings:
    gidmap:
    - container_id: 0
      host_id: 1000
      size: 1
    - container_id: 1
      host_id: 100000
      size: 65536
    uidmap:
    - container_id: 0
      host_id: 1000
      size: 1
    - container_id: 1
      host_id: 100000
      size: 65536
  kernel: 4.18.0-323.el8.x86_64
  linkmode: dynamic
  memFree: 352288768
  memTotal: 2819129344
  ociRuntime:
    name: runc
    package: runc-1.0.2-1.module+el8.5.0+12381+e822eb26.x86_64
    path: /usr/bin/runc
    version: |-
      runc version 1.0.2
      spec: 1.0.2-dev
      go: go1.16.7
      libseccomp: 2.5.1
  os: linux
  remoteSocket:
    path: /run/user/1000/podman/podman.sock
  security:
    apparmorEnabled: false
    capabilities:
CAP_NET_RAW,CAP_CHOWN,CAP_DAC_OVERRIDE,CAP_FOWNER,CAP_FSETID,C
AP_KILL,CAP_NET_BIND_SERVICE,CAP_SETFCAP,CAP_SETGID,CAP_SETPCAP,CA
P_SETUID,CAP_SYS_CHROOT
    rootless: true
    seccompEnabled: true
    seccompProfilePath: /usr/share/containers/seccomp.json
    selinuxEnabled: true
  serviceIsRemote: false
  slirp4netns:
    executable: /usr/bin/slirp4netns
```

```
    package: slirp4netns-1.1.8-1.module+el8.5.0+12381+e822eb26.x86_64
    version: |-
      slirp4netns version 1.1.8
      commit: d361001f495417b880f20329121e3aa431a8f90f
      libslirp: 4.4.0
      SLIRP_CONFIG_VERSION_MAX: 3
      libseccomp: 2.5.1
  swapFree: 3113668608
  swapTotal: 3124752384
  uptime: 11h 24m 12.52s (Approximately 0.46 days)
registries:
  search:
  - registry.fedoraproject.org
  - registry.access.redhat.com
  - registry.centos.org
  - docker.io
store:
  configFile: /home/user/.config/containers/storage.conf
  containerStore:
    number: 2
    paused: 0
    running: 0
    stopped: 2
  graphDriverName: overlay
  graphOptions:
    overlay.mount_program:
      Executable: /usr/bin/fuse-overlayfs
      Package: fuse-overlayfs-1.7.1-1.module+el8.5.0+12381+e822eb26.x86_64
      Version: |-
        fusermount3 version: 3.2.1
        fuse-overlayfs: version 1.7.1
        FUSE library version 3.2.1
        using FUSE kernel interface version 7.26
  graphRoot: /home/user/.local/share/containers/storage
  graphStatus:
    Backing Filesystem: xfs
    Native Overlay Diff: "false"
    Supports d_type: "true"
    Using metacopy: "false"
  imageStore:
    number: 3
  runRoot: /run/user/1000/containers
  volumePath: /home/user/.local/share/containers/storage/volumes
version:
  APIVersion: 3.3.1
  Built: 1630360721
  BuiltTime: Mon Aug 30 23:58:41 2021
  GitCommit: ""
  GoVersion: go1.16.7
  OsArch: linux/amd64
  Version: 3.3.1
```

- To remove all unused containers, images and volume data, enter:

```
$ podman system prune
WARNING! This will remove:
```

```
        - all stopped containers
        - all stopped pods
        - all dangling images
        - all build cache
Are you sure you want to continue? [y/N] y
```

- The **podman system prune** command removes all unused containers (both dangling and unreferenced), pods and optionally, volumes from local storage.
- Use the **--all** option to delete all unused images. Unused images are dangling images and any image that does not have any containers based on it.
- Use the **--volume** option to prune volumes. By default, volumes are not removed to prevent important data from being deleted if there is currently no container using the volume.

Additional resources

- **podman-system-df** man page
- **podman-system-info** man page
- **podman-system-prune** man page

15.5. PODMAN EVENT TYPES

You can monitor events that occur in Podman. Several event types exist and each event type reports different statuses.

The *container* event type reports the following statuses:

- attach
- checkpoint
- cleanup
- commit
- create
- exec
- export
- import
- init
- kill
- mount
- pause
- prune

- remove
- restart
- restore
- start
- stop
- sync
- unmount
- unpause

The *pod* event type reports the following statuses:

- create
- kill
- pause
- remove
- start
- stop
- unpause

The *image* event type reports the following statuses:

- prune
- push
- pull
- save
- remove
- tag
- untag

The *system* type reports the following statuses:

- refresh
- renumber

The *volume* type reports the following statuses:

- create

- prune
- remove

Additional resources

- **podman-events** man page

15.6. MONITORING PODMAN EVENTS

You can monitor and print events that occur in Podman. Each event will include a timestamp, a type, a status, name (if applicable), and image (if applicable).

Procedure

- Show Podman events:
  - To show all Podman events, enter:

```
$ podman events
2020-05-14 10:33:42.312377447 -0600 CST container create 34503c192940
(image=registry.access.redhat.com/ubi8/ubi:latest, name=keen_colden)
2020-05-14 10:33:46.958768077 -0600 CST container init 34503c192940
(image=registry.access.redhat.com/ubi8/ubi:latest, name=keen_colden)
2020-05-14 10:33:46.973661968 -0600 CST container start 34503c192940
(image=registry.access.redhat.com/ubi8/ubi:latest, name=keen_colden)
2020-05-14 10:33:50.833761479 -0600 CST container stop 34503c192940
(image=registry.access.redhat.com/ubi8/ubi:latest, name=keen_colden)
2020-05-14 10:33:51.047104966 -0600 CST container cleanup 34503c192940
(image=registry.access.redhat.com/ubi8/ubi:latest, name=keen_colden)
```

    To exit logging, press CTRL+c.

  - To show only Podman create events, enter:

```
$ podman events --filter event=create
2020-05-14 10:36:01.375685062 -0600 CST container create 20dc581f6fbf
(image=registry.access.redhat.com/ubi8/ubi:latest)
2019-03-02 10:36:08.561188337 -0600 CST container create 58e7e002344c
(image=registry.access.redhat.com/ubi8/ubi-minimal:latest)
2019-03-02 10:36:29.978806894 -0600 CST container create d81e30f1310f
(image=registry.access.redhat.com/ubi8/ubi-init:latest)
```

Additional resources

- **podman-events** man page

CHAPTER 16. CREATING AND RESTORING CONTAINER CHECKPOINTS

Checkpoint/Restore In Userspace (CRIU) is a software that enables you to set a checkpoint on a running container or an individual application and store its state to disk. You can use data saved to restore the container after a reboot at the same point in time it was checkpointed.

16.1. CREATING AND RESTORING A CONTAINER CHECKPOINT LOCALLY

This example is based on a Python based web server which returns a single integer which is incremented after each request.

Procedure

1. Create a Python based server:

```
# cat counter.py
#!/usr/bin/python3

import http.server

counter = 0

class handler(http.server.BaseHTTPRequestHandler):
    def do_GET(s):
        global counter
            s.send_response(200)
            s.send_header('Content-type', 'text/html')
            s.end_headers()
            s.wfile.write(b'%d\n' % counter)
            counter += 1


server = http.server.HTTPServer(('', 8088), handler)
server.serve_forever()
```

2. Create a container with the following definition:

```
# cat Containerfile
FROM registry.access.redhat.com/ubi8/ubi

COPY counter.py /home/counter.py

RUN useradd -ms /bin/bash counter

RUN yum -y install python3 && chmod 755 /home/counter.py

USER counter
ENTRYPOINT /home/counter.py
```

The container is based on the Universal Base Image (UBI 8) and uses a Python based server.

3. Build the container:

```
# podman build . --tag counter
```

Files **counter.py** and **Containerfile** are the input for the container build process (**podman build**). The built image is stored locally and tagged with the tag **counter**.

4. Start the container as root:

```
# podman run --name criu-test --detach counter
```

5. To list all running containers, enter:

```
# podman ps
CONTAINER ID  IMAGE  COMMAND  CREATED   STATUS  PORTS NAMES
e4f82fd84d48  localhost/counter:latest  5 seconds ago  Up 4 seconds ago  criu-test
```

6. Display IP address of the container:

```
# podman inspect criu-test --format "{{.NetworkSettings.IPAddress}}"
10.88.0.247
```

7. Send requests to the container:

```
# curl 10.88.0.247:8080
0
# curl 10.88.0.247:8080
1
```

8. Create a checkpoint for the container:

```
# podman container checkpoint criu-test
```

9. Reboot the system.

10. Restore the container:

```
# podman container restore --keep criu-test
```

11. Send requests to the container:

```
# curl 10.88.0.247:8080
2
# curl 10.88.0.247:8080
3
# curl 10.88.0.247:8080
4
```

The result now does not start at **0** again, but continues at the previous value.

This way you can easily save the complete container state through a reboot.

Additional resources

- Adding checkpoint/restore support to Podman

16.2. REDUCING STARTUP TIME USING CONTAINER RESTORE

You can use container migration to reduce startup time of containers which require a certain time to initialize. Using a checkpoint, you can restore the container multiple times on the same host or on different hosts. This example is based on the container from the Creating and restoring a container checkpoint locally.

Procedure

1. Create a checkpoint of the container, and export the checkpoint image to a **tar.gz** file:

```
# podman container checkpoint criu-test --export /tmp/chkpt.tar.gz
```

2. Restore the container from the **tar.gz** file:

```
# podman container restore --import /tmp/chkpt.tar.gz --name counter1
# podman container restore --import /tmp/chkpt.tar.gz --name counter2
# podman container restore --import /tmp/chkpt.tar.gz --name counter3
```

   The **--name** (**-n**) option specifies a new name for containers restored from the exported checkpoint.

3. Display ID and name of each container:

```
# podman ps -a --format "{{.ID}} {{.Names}}"
a8b2e50d463c counter3
faabc5c27362 counter2
2ce648af11e5 counter1
```

4. Display IP address of each container:

```
#  podman inspect counter1 --format "{{.NetworkSettings.IPAddress}}"
10.88.0.248

#  podman inspect counter2 --format "{{.NetworkSettings.IPAddress}}"
10.88.0.249

#  podman inspect counter3 --format "{{.NetworkSettings.IPAddress}}"
10.88.0.250
```

5. Send requests to each container:

```
#  curl 10.88.0.248:8080
4
#  curl 10.88.0.249:8080
4
#  curl 10.88.0.250:8080
4
```

   Note, that the result is **4** in all cases, because you are working with different containers restored from the same checkpoint.

Using this approach, you can quickly start up stateful replicas of the initially checkpointed container.

Additional resources

- Container migration with Podman on RHEL

16.3. MIGRATING CONTAINERS AMONG SYSTEMS

This procedure shows the migration of running containers from one system to another, without losing the state of the applications running in the container. This example is based on the container from the Creating and restoring a container checkpoint locally .

section tagged with **counter**.

Prerequisites

The following steps are not necessary if the container is pushed to a registry as Podman will automatically download the container from a registry if it is not available locally. This example does not use a registry, you have to export previously built and tagged container (see Creating and restoring a container checkpoint locally. section).

- Export previously built container:

```
# podman save --output counter.tar counter
```

- Copy exported container image to the destination system (***other_host***):

```
# scp counter.tar other_host:
```

- Import exported container on the destination system:

```
# ssh other_host podman load --input counter.tar
```

Now the destination system of this container migration has the same container image stored in its local container storage.

Procedure

1. Start the container as root:

```
# podman run --name criu-test --detach counter
```

2. Display IP address of the container:

```
# podman inspect criu-test --format "{{.NetworkSettings.IPAddress}}"
10.88.0.247
```

3. Send requests to the container:

```
# curl 10.88.0.247:8080
0
# curl 10.88.0.247:8080
1
```

4. Create a checkpoint of the container, and export the checkpoint image to a **tar.gz** file:

```
# podman container checkpoint criu-test --export /tmp/chkpt.tar.gz
```

5. Copy the checkpoint archive to the destination host:

```
# scp /tmp/chkpt.tar.gz other_host:/tmp/
```

6. Restore the checkpoint on the destination host (***other_host***):

```
# podman container restore --import /tmp/chkpt.tar.gz
```

7. Send a request to the container on the destination host (***other_host***):

```
# *curl 10.88.0.247:8080*
2
```

As a result, the stateful container has been migrated from one system to another without losing its state.

Additional resources

- Container migration with Podman on RHEL

CHAPTER 17. USING PODMAN IN HPC ENVIRONMENT

You can use Podman with Open MPI (Message Passing Interface) to run containers in a High Performance Computing (HPC) environment.

17.1. USING PODMAN WITH MPI

The example is based on the ring.c program taken from Open MPI. In this example, a value is passed around by all processes in a ring-like fashion. Each time the message passes rank 0, the value is decremented. When each process receives the 0 message, it passes it on to the next process and then quits. By passing the 0 first, every process gets the 0 message and can quit normally.

Procedure

1. Install Open MPI:

```
# yum install openmpi
```

2. To activate the environment modules, type:

```
$ . /etc/profile.d/modules.sh
```

3. Load the **mpi/openmpi-x86_64** module:

```
$ module load mpi/openmpi-x86_64
```

   Optionally, to automatically load **mpi/openmpi-x86_64** module, add this line to the **.bashrc** file:

```
$ echo "module load mpi/openmpi-x86_64" >> .bashrc
```

4. To combine **mpirun** and **podman**, create a container with the following definition:

```
$ cat Containerfile
FROM registry.access.redhat.com/ubi8/ubi

RUN yum -y install openmpi-devel wget && \
    yum clean all

RUN wget https://raw.githubusercontent.com/open-mpi/ompi/master/test/simple/ring.c && \
    /usr/lib64/openmpi/bin/mpicc ring.c -o /home/ring && \
    rm -f ring.c
```

5. Build the container:

```
$ podman build --tag=mpi-ring .
```

6. Start the container. On a system with 4 CPUs this command starts 4 containers:

```
$ mpirun \ --mca orte_tmpdir_base /tmp/podman-mpirun \ podman run --env-host \ -v
/tmp/podman-mpirun:/tmp/podman-mpirun \ --userns=keep-id \ --net=host --pid=host --
ipc=host \ mpi-ring /home/ring
Rank 2 has cleared MPI_Init
```

```
Rank 2 has completed ring
Rank 2 has completed MPI_Barrier
Rank 3 has cleared MPI_Init
Rank 3 has completed ring
Rank 3 has completed MPI_Barrier
Rank 1 has cleared MPI_Init
Rank 1 has completed ring
Rank 1 has completed MPI_Barrier
Rank 0 has cleared MPI_Init
Rank 0 has completed ring
Rank 0 has completed MPI_Barrier
```

As a result, **mpirun** starts up 4 Podman containers and each container is running one instance of the **ring** binary. All 4 processes are communicating over MPI with each other.

Additional resources

- Podman in HPC environments

17.2. THE MPIRUN OPTIONS

The following **mpirun** options are used to start the container:

- **--mca orte_tmpdir_base /tmp/podman-mpirun** line tells Open MPI to create all its temporary files in **/tmp/podman-mpirun** and not in **/tmp**. If using more than one node this directory will be named differently on other nodes. This requires mounting the complete **/tmp** directory into the container which is more complicated.

The **mpirun** command specifies the command to start, the **podman** command. The following **podman** options are used to start the container:

- **run** command runs a container.
- **--env-host** option copies all environment variables from the host into the container.
- **-v /tmp/podman-mpirun:/tmp/podman-mpirun** line tells Podman to mount the directory where Open MPI creates its temporary directories and files to be available in the container.
- **--userns=keep-id** line ensures the user ID mapping inside and outside the container.
- **--net=host --pid=host --ipc=host** line sets the same network, PID and IPC namespaces.
- **mpi-ring** is the name of the container.
- **/home/ring** is the MPI program in the container.

Additional resources

- Podman in HPC environments

CHAPTER 18. RUNNING SPECIAL CONTAINER IMAGES

This chapter provides information about some special types of container images. Some container images have built in labels called *runlabels* that allow you to run those containers with preset options and arguments. The **podman container runlabel <label>** command, allows you to execute the command defined in the **<label>** for the container image. Supported labels are **install**, **run** and **uninstall**.

18.1. OPENING PRIVILEGES TO THE HOST

There are several differences between privileged and non-privileged containers. For example, the toolbox container is a privileged container. Here are examples of privileges that may or may not be open to the host from a container:

- **Privileges**: A privileged container disables the security features that isolate the container from the host. You can run a privileged container using the **podman run --privileged *<image_name>*** command. You can, for example, delete files and directories mounted from the host that are owned by the root user.
- **Process tables**: You can use the **podman run --privileged --pid=host *<image_name>*** command to use the host PID namespace for the container. Then you can use the **ps -e** command within a privileged container to list all processes running on the host. You can pass a process ID from the host to commands that run in the privileged container (for example, **kill <PID>**).
- **Network interfaces**: By default, a container has only one external network interface and one loopback network interface. You can use the **podman run --net=host *<image_name>*** command to access host network interfaces directly from within the container.
- **Inter-process communications**: The IPC facility on the host is accessible from within the privileged container. You can run commands such as **ipcs** to see information about active message queues, shared memory segments, and semaphore sets on the host.

18.2. CONTAINER IMAGES WITH RUNLABELS

Some Red Hat images include labels that provide pre-set command lines for working with those images. Using the **podman container runlabel <label>** command, you can use the **podman** command to execute the command defined in the **<label>** for the image.

Existing runlabels include:

- **install**: Sets up the host system before executing the image. Typically, this results in creating files and directories on the host that the container can access when it is run later.
- **run**: Identifies podman command line options to use when running the container. Typically, the options will open privileges on the host and mount the host content the container needs to remain permanently on the host.
- **uninstall**: Cleans up the host system after you finish running the container.

18.3. RUNNING RSYSLOG WITH RUNLABELS

The **rhel8/rsyslog** container image is made to run a containerized version of the **rsyslogd** daemon. The **rsyslog** image contains the following runlabels: **install**, **run** and **uninstall**. The following procedure steps you through installing, running, and uninstalling the **rsyslog** image:

Procedure

1. Pull the **rsyslog** image:

```
# podman pull registry.redhat.io/rhel8/rsyslog
```

2. Display the **install** runlabel for **rsyslog**:

```
# podman container runlabel install --display rhel8/rsyslog
command: podman run --rm --privileged -v /:/host -e HOST=/host -e
IMAGE=registry.redhat.io/rhel8/rsyslog:latest -e NAME=rsyslog
registry.redhat.io/rhel8/rsyslog:latest /bin/install.sh
```

This shows that the command will open privileges to the host, mount the host root filesystem on **/host** in the container, and run an **install.sh** script.

3. Run the **install** runlabel for **rsyslog**:

```
# podman container runlabel install rhel8/rsyslog
command: podman run --rm --privileged -v /:/host -e HOST=/host -e
IMAGE=registry.redhat.io/rhel8/rsyslog:latest -e NAME=rsyslog
registry.redhat.io/rhel8/rsyslog:latest /bin/install.sh
Creating directory at /host//etc/pki/rsyslog
Creating directory at /host//etc/rsyslog.d
Installing file at /host//etc/rsyslog.conf
Installing file at /host//etc/sysconfig/rsyslog
Installing file at /host//etc/logrotate.d/syslog
```

This creates files on the host system that the **rsyslog** image will use later.

4. Display the **run** runlabel for **rsyslog**:

```
# podman container runlabel run --display rhel8/rsyslog
command: podman run -d --privileged --name rsyslog --net=host --pid=host -v
/etc/pki/rsyslog:/etc/pki/rsyslog -v /etc/rsyslog.conf:/etc/rsyslog.conf -v
/etc/sysconfig/rsyslog:/etc/sysconfig/rsyslog -v /etc/rsyslog.d:/etc/rsyslog.d -v /var/log:/var/log
-v /var/lib/rsyslog:/var/lib/rsyslog -v /run:/run -v /etc/machine-id:/etc/machine-id -v
/etc/localtime:/etc/localtime -e IMAGE=registry.redhat.io/rhel8/rsyslog:latest -e
NAME=rsyslog --restart=always registry.redhat.io/rhel8/rsyslog:latest /bin/rsyslog.sh
```

This shows that the command opens privileges to the host and mount specific files and directories from the host inside the container, when it launches the **rsyslog** container to run the **rsyslogd** daemon.

5. Execute the **run** runlabel for **rsyslog**:

```
# podman container runlabel run rhel8/rsyslog
command: podman run -d --privileged --name rsyslog --net=host --pid=host -v
/etc/pki/rsyslog:/etc/pki/rsyslog -v /etc/rsyslog.conf:/etc/rsyslog.conf -v
/etc/sysconfig/rsyslog:/etc/sysconfig/rsyslog -v /etc/rsyslog.d:/etc/rsyslog.d -v /var/log:/var/log
-v /var/lib/rsyslog:/var/lib/rsyslog -v /run:/run -v /etc/machine-id:/etc/machine-id -v
/etc/localtime:/etc/localtime -e IMAGE=registry.redhat.io/rhel8/rsyslog:latest -e
NAME=rsyslog --restart=always registry.redhat.io/rhel8/rsyslog:latest /bin/rsyslog.sh
28a0d719ff179adcea81eb63cc90fcd09f1755d5edb121399068a4ea59bd0f53
```

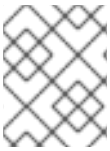
The **rsyslog** container opens privileges, mounts what it needs from the host, and runs the **rsyslogd** daemon in the background (**-d**). The **rsyslogd** daemon begins gathering log messages and directing messages to files in the **/var/log** directory.

6. Display the **uninstall** runlabel for **rsyslog**:

```
# podman container runlabel uninstall --display rhel8/rsyslog
command: podman run --rm --privileged -v /:/host -e HOST=/host -e
IMAGE=registry.redhat.io/rhel8/rsyslog:latest -e NAME=rsyslog
registry.redhat.io/rhel8/rsyslog:latest /bin/uninstall.sh
```

7. Run the **uninstall** runlabel for **rsyslog**:

```
# podman container runlabel uninstall rhel8/rsyslog
command: podman run --rm --privileged -v /:/host -e HOST=/host -e
IMAGE=registry.redhat.io/rhel8/rsyslog:latest -e NAME=rsyslog
registry.redhat.io/rhel8/rsyslog:latest /bin/uninstall.sh
```



NOTE

In this case, the **uninstall.sh** script just removes the **/etc/logrotate.d/syslog** file. It does not clean up the configuration files.

CHAPTER 19. USING THE CONTAINER-TOOLS API

The new REST based Podman 2.0 API replaces the old remote API for Podman that used the varlink library. The new API works in both a rootful and a rootless environment.

The Podman v2.0 RESTful API consists of the Libpod API providing support for Podman, and Docker-compatible API. With this new REST API, you can call Podman from platforms such as cURL, Postman, Google's Advanced REST client, and many others.

19.1. ENABLING THE PODMAN API USING SYSTEMD IN ROOT MODE

This procedure shows how to do the following:

1. Use systemd to activate the Podman API socket.
2. Use a Podman client to perform basic commands.

Prerequisities

- The **podman-remote** package is installed.

```
# yum install podman-remote
```

Procedure

1. Start the service immediately:

```
# systemctl enable --now podman.socket
```

2. To enable the link to **var/lib/docker.sock** using the **docker-podman** package:

```
# yum install podman-docker
```

Verification steps

1. Display system information of Podman:

```
# podman-remote info
```

2. Verify the link:

```
# ls -al /var/run/docker.sock
lrwxrwxrwx. 1 root root 23 Nov  4 10:19 /var/run/docker.sock -> /run/podman/podman.sock
```

Additional resources

- Podman v2.0 RESTful API
- A First Look At Podman 2.0 API
- Sneak peek: Podman's new REST API

19.2. ENABLING THE PODMAN API USING SYSTEMD IN ROOTLESS MODE

This procedure shows how to use systemd to activate the Podman API socket and podman API service.

Prerequisites

- The **podman-remote** package is installed.

```
# yum install podman-remote
```

Procedure

1. Enable and start the service immediately:

```
$ systemctl --user enable --now podman.socket
```

2. Optional. To enable programs using Docker to interact with the rootless Podman socket:

```
$ export DOCKER_HOST=unix:///run/user/<uid>/podman//podman.sock
```

Verification steps

1. Check the status of the socket:

```
$ systemctl --user status podman.socket
● podman.socket - Podman API Socket
 Loaded: loaded (/usr/lib/systemd/user/podman.socket; enabled; vendor preset: enabled)
Active: active (listening) since Mon 2021-08-23 10:37:25 CEST; 9min ago
Docs: man:podman-system-service(1)
Listen: /run/user/1000/podman/podman.sock (Stream)
CGroup: /user.slice/user-1000.slice/user@1000.service/podman.socket
```

   The **podman.socket** is active and is listening at **/run/user/*<uid>*/podman.podman.sock**, where ***<uid>*** is the user's ID.

2. Display system information of Podman:

```
$ podman-remote info
```

Additional resources

- Podman v2.0 RESTful API
- A First Look At Podman 2.0 API
- Sneak peek: Podman's new REST API
- Exploring Podman RESTful API using Python and Bash

19.3. RUNNING THE PODMAN API MANUALLY

This procedure describes how to run the Podman API. This is useful for debugging API calls, especially when using the Docker compatibility layer.

Prerequisities

- The **podman-remote** package is installed.

```
# yum install podman-remote
```

Procedure

1. Run the service for the REST API:

```
# podman system service -t 0 --log-level=debug
```

   - The value of 0 means no timeout. The default endpoint for a rootful service is **unix:/run/podman/podman.sock**.
   - The **--log-level <level>** option sets the logging level. The standard logging levels are **debug**, **info**, **warn**, **error**, **fatal**, and **panic**.

2. In another terminal, display system information of Podman. The **podman-remote** command, unlike the regular **podman** command, communicates through the Podman socket:

```
# podman-remote info
```

3. To troubleshoot the Podman API and display request and responses, use the **curl** comman. To get the information about the Podman installation on the Linux server in JSON format:

```
# curl -s --unix-socket /run/podman/podman.sock http://d/v1.0.0/libpod/info | jq
   {
  "host": {
    "arch": "amd64",
    "buildahVersion": "1.15.0",
    "cgroupVersion": "v1",
    "conmon": {
      "package": "conmon-2.0.18-1.module+el8.3.0+7084+c16098dd.x86_64",
      "path": "/usr/bin/conmon",
      "version": "conmon version 2.0.18, commit:
7fd3f71a218f8d3a7202e464252aeb1e942d17eb"
    },
    …
  "version": {
    "APIVersion": 1,
    "Version": "2.0.0",
    "GoVersion": "go1.14.2",
    "GitCommit": "",
    "BuiltTime": "Thu Jan  1 01:00:00 1970",
    "Built": 0,
    "OsArch": "linux/amd64"
  }
}
```

   A **jq** utility is a command-line JSON processor.

4. Pull the **registry.access.redhat.com/ubi8/ubi** container image:

```
# curl -XPOST --unix-socket /run/podman/podman.sock -v
'http://d/v1.0.0/images/create?fromImage=registry.access.redhat.com%2Fubi8%2Fubi'
*   Trying /run/podman/podman.sock...
* Connected to d (/run/podman/podman.sock) port 80 (#0)
> POST /v1.0.0/images/create?fromImage=registry.access.redhat.com%2Fubi8%2Fubi
HTTP/1.1
> Host: d
> User-Agent: curl/7.61.1
> Accept: /
>
< HTTP/1.1 200 OK
< Content-Type: application/json
< Date: Tue, 20 Oct 2020 13:58:37 GMT
< Content-Length: 231
<
{"status":"pulling image () from registry.access.redhat.com/ubi8/ubi:latest,
registry.redhat.io/ubi8/ubi:latest","error":"","progress":"","progressDetail":
{},"id":"ecbc6f53bba0d1923ca9e92b3f747da8353a070fccbae93625bd8b47dbee772e"}
* Connection #0 to host d left intact
```

5. Display the pulled image:

```
# curl --unix-socket /run/podman/podman.sock -v 'http://d/v1.0.0/libpod/images/json' |
jq
*   Trying /run/podman/podman.sock...
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0* Connected to d
(/run/podman/podman.sock) port 80 (0) > GET /v1.0.0/libpod/images/json HTTP/1.1 > Host: d
> User-Agent: curl/7.61.1 > Accept: / > < HTTP/1.1 200 OK < Content-Type: application/json
< Date: Tue, 20 Oct 2020 13:59:55 GMT < Transfer-Encoding: chunked < { [12498 bytes
data] 100 12485 0 12485 0 0 2032k 0 --:--:-- --:--:-- --:--:-- 2438k * Connection #0 to host d
left intact [ { "Id":
"ecbc6f53bba0d1923ca9e92b3f747da8353a070fccbae93625bd8b47dbee772e",
"RepoTags": [ "registry.access.redhat.com/ubi8/ubi:latest", "registry.redhat.io/ubi8/ubi:latest"
], "Created": "2020-09-01T19:44:12.470032Z", "Size": 210838671, "Labels": { "architecture":
"x86_64", "build-date": "2020-09-01T19:43:46.041620", "com.redhat.build-host": "cpt-
1008.osbs.prod.upshift.rdu2.redhat.com", ... "maintainer": "Red Hat, Inc.", "name": "ubi8", ...
"summary": "Provides the latest release of Red Hat Universal Base Image 8.", "url":
"https://access.redhat.com/containers//registry.access.redhat.com/ubi8/images/8.2-347",
      ...
    },
    "Names": [
      "registry.access.redhat.com/ubi8/ubi:latest",
      "registry.redhat.io/ubi8/ubi:latest"
    ],
    ...
    ]
  }
]
```

Additional resources

- Podman v2.0 RESTful API
- Sneak peek: Podman's new REST API
- Exploring Podman RESTful API using Python and Bash
- **podman-system-service** man page