Red Hat Enterprise Linux 10

Building, running, and managing containers

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

Last Updated: 2026-02-04

Red Hat Enterprise Linux 10 Building, running, and managing containers

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

Abstract

Red Hat Enterprise Linux (RHEL) provides a number of command-line tools for working with container images. You can manage pods and container images using Podman. To build, update, and manage container images you can use Buildah. To copy and inspect images in remote repositories, you can use Skopeo. In addition to command-line tools, you can also use Podman Desktop, a graphical interface to manage containers, images, pods, and registries.

Table of Contents

PROVIDING FEEDBACK ON RED HAT DOCUMENTATION

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

Submitting feedback through Jira (account required)

1. Log in to the Jira website.
2. Click **Create** in the top navigation bar.
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 **Create** at the bottom of the dialogue.

CHAPTER 1. INTRODUCTION TO CONTAINERS

Linux containers combine lightweight application isolation with the flexibility of image-based deployment methods. They use RHEL core technologies to reduce security risks and provide an enterprise-quality environment.

RHEL implements Linux containers by using 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 Red Hat Enterprise Linux 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
- **crun** - an optional runtime that can be configured and gives greater flexibility, control, and security for rootless containers

Besides these tools, you can also use Podman Desktop, that is, a GUI-based application for container management. Podman Desktop is built on top of Podman, making it easy to create, manage, and run containerized applications visually.

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. INFORMATION ABOUT CONTAINERS COMMAND-LINE TOOLS

Red Hat OpenShift provides powerful command-line and web UI tools for building, managing, and running containers in units referred to as pods. You can build and manage individual containers and container images outside of OpenShift.

You can use the following tools to perform these tasks directly on RHEL systems. Unlike other container tools implementations, the tools provided by OpenShift 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

Besides these tools, you can also use Podman Desktop, that is, a GUI-based application for container management. Podman Desktop is built on top of Podman, making it easy to create, manage, and run containerized applications visually.

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 Container Platform and Using the CRI-O Container Engine for details.

1.2. CHARACTERISTICS OF PODMAN, PODMAN DESKTOP, BUILDAH, AND SKOPEO

The Podman, Podman Desktop, 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, Podman Desktop, 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, Podman Desktop, 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 Using the container-tools API chapter.
- Podman Desktop provides a simple and intuitive interface to run your application workload on a Podman engine.

1.3. RUNNING CONTAINERS WITHOUT DOCKER

You can run containerized applications on RHEL by using Podman to eliminate dependencies on a central daemon and improve system security through rootless operation.

The **podman-docker** package also provides a compatible environment for managing Docker-formatted images while ensuring a smaller system footprint and better isolation.

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 10.

- 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

- Migrating from Docker to Podman Desktop

1.4. SUPPORTED ARCHITECTURES FOR CONTAINER DEPLOYMENT IN RHEL

Deploy containerized applications on supported hardware architectures in RHEL to ensure workload compatibility across diverse environments. Identifying the appropriate platform enables you to maintain consistent performance and portability for your container images.

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. INSTALLING CONTAINER TOOLS

Install the **container-tools** meta-package to get Podman, Buildah, Skopeo, CRIU, Udica, and all required libraries. This provides a complete environment for working with containers on RHEL.

NOTE

The stable streams are not available on RHEL 10. To receive stable access to Podman, Buildah, Skopeo, and others, use the RHEL EUS subscription.

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. Install the **container-tools** meta-package:

   ```
   # dnf install container-tools
   ```

   You can also install **podman**, **buildah**, and **skopeo** individually if you prefer.

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

   ```
   # dnf install podman-docker
   ```

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

1.6. INSTALLING PODMAN DESKTOP

Install Podman Desktop to visualize and manage your development environment, including pods and containers. This GUI tool runs on the Podman engine and simplifies tasks such as building images and deploying to Kubernetes.

Podman Desktop helps you perform development tasks and visualize your development environment, such as the number of pods or containers running. You can run this tool on three different operating systems; macOS, Windows, and Linux. Podman Desktop runs your workloads on a Podman engine and therefore, provides you Podman-native capabilities to interact with your containerized applications. As a developer, you can:

- Create and manage containers or pods
- Manage container images
- Deploy containers or pods to Kubernetes using Kind, Lima, Minikube or OpenShift
- Manage Docker compatibility to run your Docker workloads on a Podman engine
- Integrate your tools using extensions

For installation, use the subscription manager package on a RHEL 10 machine.

Prerequisites

- You have a RHEL 10 machine.
- You have registered with the **subscription-manager**.

Procedure

1. Open a terminal, and enable the RHEL extensions repository:

   ```
   # subscription-manager repos --enable rhel-10-for-$(arch)-extensions-rpms
   ```

2. Enter your password when prompted.
3. Install Podman Desktop:

   ```
   # dnf install podman-desktop
   ```

4. Enter **y** to confirm the installed size.
5. Enter **y** to import the GPG key and complete the installation.

Verification

1. Enter Podman Desktop in the search box at the top of your home screen, and click the Podman Desktop application to open it.
2. Follow the prompts to complete a quick onboarding process with the application.

NOTE

Podman is included with a RHEL subscription, and the application automatically detects and runs it.

Next steps

- Perform basic tasks, such as:

- Start a container
- Create a pod
- Working with container images
- Deploying a pod or container to Kubernetes
- Viewing container logs
- Containers and Kubernetes development with Podman Desktop

Additional resources

- Podman Desktop documentation
- Tutorials
- Blogs

1.7. SPECIAL CONSIDERATIONS FOR ROOTLESS CONTAINERS

Understand the configuration differences and limitations when running containers as a non-root user. Rootless containers use different storage paths and cannot modify certain system features or bind to privileged ports without configuration.

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 docker.io/ntpd/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
  ```

1.8. USING MODULES FOR ADVANCED PODMAN CONFIGURATION

You can use Podman modules to load a predetermined set of configurations. Podman modules are **containers.conf** files in the Tom's Obvious Minimal Language (TOML) format.

These modules are located in the following directories, or their subdirectories:

- For rootless users: **$HOME/.config/containers/containers.conf.modules**
- For root users: **/etc/containers/containers.conf.modules**, or **/usr/share/containers/containers.conf.modules**

You can load the modules on-demand with the **podman --module <your_module_name>** command to override the system and user configuration files. Working with modules involve the following facts:

- You can specify modules multiple times by using the **--module** option.
- If **<your_module_name>** is the absolute path, the configuration file will be loaded directly.
- The relative paths are resolved relative to the three module directories mentioned previously.
- Modules in **$HOME** override those in the **/etc/** and **/usr/share/** directories.

For more information, see the **containers.conf(5)** man page on your system.

1.9. ADDITIONAL RESOURCES

- A Practical Introduction to Container Terminology

CHAPTER 2. TYPES OF CONTAINER IMAGES

Identify standard and multi architecture image formats to determine the optimal deployment strategy for your workloads on RHEL. Selecting the appropriate image type helps ensure application compatibility and optimized delivery across diverse hardware platforms by using tools such as Podman.

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 enable sharing and deployment of containerized applications across various platforms, including non-Red Hat ones. They serve as a foundation for cloud-native and web applications.

2.1. GENERAL CHARACTERISTICS OF RHEL CONTAINER IMAGES

Review the shared characteristics of RHEL base images and UBI images. These supported, cataloged, and updated images provide a secure foundation for your applications.

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.

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

2.2. CHARACTERISTICS OF UBI IMAGES

Use Universal Base Images (UBI) to build redistributable container images based on RHEL software. UBI offers various image types, such as micro and standard, to suit different application needs.

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 DNF repositories** DNF repositories include RPM packages and updates that allow you to add application dependencies and rebuild UBI container images.
    - The **ubi-10-baseos** repository holds the redistributable subset of RHEL packages you can include in your container.
    - The **ubi-10-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 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

- (Re)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

2.3. UNDERSTANDING THE UBI STANDARD IMAGES

You can use the UBI standard images as a reliable, high-performance foundation for your containerized applications on RHEL. With these freely redistributable images you can build and share secure workloads that benefit from the stability and extensive package library of the RHEL ecosystem.

The standard images (named **ubi**) are designed for any application that runs on RHEL. The key features of Red Hat Universal Base Image (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 services automatically, such as a Web server (**httpd**) or FTP server (**vsftpd**).

- **dnf**: You have access to free DNF repositories for adding and updating software. You can use the standard set of **dnf** commands (**dnf**, **dnf-config-manager**, **dnfdownloader**, 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 contains more content than minimal images but less than standard images.

Because the **ubi10-init** image builds on top of the **ubi10** image, their contents are mostly the same. However, there are a few critical differences:

- **ubi10-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 **ubi10-init** ignores normal signals to exit (**SIGTERM** and **SIGKILL**), but will terminate if it receives **SIGRTMIN+3**
- **ubi10**:
  - 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 **dnf** 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 runtime 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.
  - 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 want to park on a specific dot release, see the Extended Update Support article for more information.

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

You can also use **ubi-micro** 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 REGISTRIES

Configure container image registries to store and retrieve container images and artifacts. The system-wide **/etc/containers/registries.conf** file manages which registries tools like Podman and Buildah use.

If the container image given to a container tool is not fully qualified, then the container tool references the **registries.conf** file. Within the **registries.conf** file, you can specify aliases for short names, granting administrators full control over where images are pulled from when not fully qualified.

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/ubi10/ubi** container image is identified by:

- Registry server (**registry.redhat.io**)
- Namespace (**ubi10**)
- 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 **ubi10/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**

NOTE

Use 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.

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

Configure container registries on RHEL to control how Podman searches for and retrieves images. Defining trusted sources and search priorities ensures secure, predictable application deployment across your infrastructure.

You can display the container registries by using the **podman info --format** command:

```
$ podman info -f json | jq '.registries["search"]'
[
  "registry.access.redhat.com",
  "registry.redhat.io",
  "docker.io"
]
```

NOTE

The **podman info** command is available in Podman 4.0.0 or later.

You can edit 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.access.redhat.com", "registry.redhat.io", "docker.io"]
short-name-mode = "enforcing"
```

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 by 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

The **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"
```

For more information, see the **podman-pull(1)** and **podman-info(1)** man pages on your system.

Additional resources

- Setting up container registries using Podman Desktop

3.3. SEARCHING FOR CONTAINER IMAGES

You can search for images across container registries by using the **podman search** command. You can also search for images at Red Hat Container Catalog . The catalog 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.

To search for the **postgresql-10** images in the quay.io registry, follow the steps.

Prerequisites

- The **container-tools** meta-package is installed.
- 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/rhel10/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.

   For more information, see the **podman-search(1)** man page on your system.

3.4. CONFIGURING SHORT-NAME ALIASES

Configure short-name aliases in **registries.conf** to map short names, such as ubi10, to fully qualified image names. This provides control over image sources and helps prevent spoofing risks associated with short names.

Always pull an image by its fully-qualified name. However, it is customary to pull images by short names. For example, you can use **ubi10** instead of **registry.access.redhat.com/ubi10: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 table in the form **"name" = "value"**.

You can see the lists of aliases in the **/etc/containers/registries.conf.d** directory. Red Hat provides a set of aliases in this directory. For example, **podman pull ubi10** directly resolves to the right image, that is **registry.access.redhat.com/ubi10: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.

CHAPTER 4. WORKING WITH CONTAINER IMAGES

Manage container images by using the Podman tool. You can use this tool to pull the image, inspect, tag, save, load, redistribute, and define the image signature.

4.1. PULLING IMAGES FROM REGISTRIES

Download container images from remote registries to your local system by using the **podman pull** command. This makes the image available for creating and running containers.

Prerequisites

- The **container-tools** meta-package is installed.

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/ubi10/ubi** container image:

```
$ podman pull registry.redhat.io/ubi10/ubi
```

Verification

- List all images pulled to your local system:

```
$ podman images
REPOSITORY                     TAG     IMAGE ID      CREATED      SIZE
registry.redhat.io/ubi10/ubi    latest  3269c37eae33  7 weeks ago  208 MB
```

For more information, see the **podman-pull(1)** man page on your system.

Additional resources

- Pulling an image using Podman Desktop

4.2. 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.

Prerequisites

- The **container-tools** meta-package is installed.

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
  ```

4.3. LISTING IMAGES

You can list locally stored container images on by using Podman to verify version availability and manage system storage. Viewing the image list ensures you select the appropriate foundation for your workloads while maintaining visibility into your local environment.

Prerequisites

- The **container-tools** meta-package is installed.
- 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/ubi10/ubi  latest  3269c37eae33  6 weeks ago  208 MB
```

For more information, see the **podman-images(1)** man page on your system.

4.4. 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 about containers and images identified by name or ID.

Prerequisites

- The **container-tools** meta-package is installed.
- A pulled image is available on the local system.

Procedure

- Inspect the **registry.redhat.io/ubi10/ubi** image:

```
$ podman inspect registry.redhat.io/ubi10/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": "ubi10-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 ubi10/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.

4.5. 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. This reveals details such as the default command, environment variables, and architecture.

Prerequisites

- The **container-tools** meta-package is installed.

Procedure

- Inspect the **registry.redhat.io/ubi10/ubi-init** image:

```
# skopeo inspect docker://registry.redhat.io/ubi10/ubi-init
{
    "Name": "registry.redhat.io/ubi10/ubi10-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": "ubi10-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
        ...
    }
}
```

For more information, see the **skopeo-inspect(1)** man page on your system.

4.6. 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.

Prerequisites

- The **container-tools** meta-package is installed.

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
```

Refer to **skopeo-copy(1)** man page on your system for more information.

4.7. COPYING IMAGE LAYERS TO A LOCAL DIRECTORY

Copying container image layers to a local directory by using Skopeo to audit image contents or troubleshoot file system changes. Storing layers locally enables you to inspect the image structure and verify security compliance without the need to deploy the container.

Prerequisites

- The **container-tools** meta-package is installed.

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
```

Refer to **skopeo-copy(1)** man page on your system for more information.

4.8. TAGGING IMAGES

Assign additional names or tags to local images by using the **podman tag** command. Tagging helps organize images and prepare them for pushing to specific registries.

This additional name can consist of several parts: *<registryhost>/<username>/<name>:<tag>*.

Prerequisites

- The **container-tools** meta-package is installed.
- 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/ubi10/ubi   latest  3269c37eae33  7 weeks ago  208 MB
```

2. Assign the **myubi** name to the **registry.redhat.io/ubi10/ubi** image using one of the following options:

   - The image name:

     ```
     $ podman tag registry.redhat.io/ubi10/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/ubi10/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 **10** tag to the **registry.redhat.io/ubi10/ubi** image using either:

   - The image name:

     ```
     $ podman tag registry.redhat.io/ubi10/ubi myubi:10
     ```

   - The image ID:

     ```
     $ podman tag 3269c37eae33 myubi:10
     ```

     Both commands give you the same result.

Verification

1. List all images:

   ```
   $ podman images
   REPOSITORY                     TAG    IMAGE ID     CREATED      SIZE
   registry.redhat.io/ubi10/ubi   latest 3269c37eae33 2 months ago 208 MB
   localhost/myubi                latest 3269c37eae33 2 months ago 208 MB
   localhost/myubi                10     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.

2. After tagging the **registry.redhat.io/ubi10/ubi** image, you have three options to run the container:

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

- by name (**localhost/myubi:10**)
  For more information, see **podman-tag(1)** man page on your system.

4.9. BUILDING MULTI-ARCHITECTURE IMAGES

Build multi-architecture container images by using Podman on RHEL to ensure your applications run consistently across diverse hardware platforms. This approach allows a single image manifest to support multiple architectures, without the requirement of unique builds for each environment.

Prerequisites

- The **container-tools** meta-package is installed.

Procedure

1. Create **Containerfiles** for each architecture you want to support.
2. Build images for each architecture. For example:

   ```
   $ podman build --platform linux/arm64,linux/amd64 --manifest <registry>/<image> .
   ```

   - The **--platform linux/arm64,linux/amd64** option specifies the target platforms for which the container image is being built.
   - The **--manifest *<registry>/<image>*** option creates a manifest list with the specified name, that is ***<registry>/<image>***, and adds the newly-built images to them. A manifest list is a collection of image manifests, each one targeting a different architecture.
3. Push the manifest list to the registry:

   ```
   $ podman manifest push <registry>/<image>
   ```

   This manifest list acts as a single entry point for pulling the multi-architecture container.

   As a result, you can pull the appropriate container image for your platform, based on a single manifest list.

   You can also remove items from the manifest list by using the **podman manifest remove *<manifest_list> <digest_ID>*** command, where ***<digest_ID>*** is the SHA-256 checksum of the container image. For example: **podman manifest remove *<registry>/<image>* sha256:cb8a924afdf…**.

Verification

- Display the manifest list:

  ```
  $ podman manifest inspect <registry>/<image>
  ```

  Refer to the **podman-build(1)** and **podman-manifest(1)** man page on your system for more information.

4.10. 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 the **--format** option to specify the archive format. The supported formats are:

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

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

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

Prerequisites

- The **container-tools** meta-package is installed.
- A pulled image is available on the local system.

Procedure

1. Save the **registry.redhat.io/rhel10/support-tools** image as a tarball:
   - In the default **docker-archive** format:

     ```
     $ podman save -o mysupport-tools.tar registry.redhat.io/rhel10/support-tools:latest
     ```

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

     ```
     $ podman save -o mysupport-tools-oci.tar --format=oci-archive
     registry.redhat.io/rhel10/support-tools
     ```

     The **mysupport-tools.tar** and **mysupport-tools-oci.tar** archives are stored in your current directory. The next steps are performed with the **mysupport-tools.tar** tarball.

2. Check the file type of **mysupport-tools.tar**:

   ```
   $ file mysupport-tools.tar
   mysupport-tools.tar: POSIX tar archive
   ```

3. To load the **registry.redhat.io/rhel10/support-tools:latest** image from the **mysupport-tools.tar**:

   ```
   $ podman load -i mysupport-tools.tar
   ...
   Loaded image(s): registry.redhat.io/rhel10/support-tools:latest
   ```

For more information, see the **podman-save(1)** and **podman-load(1)** man pages on your system.

4.11. REDISTRIBUTING UBI IMAGES

Share your custom UBI-based images by pushing them to a registry with the **podman push** command. This enables others to download and use your modified images.

Prerequisites

- The **container-tools** meta-package is installed.
- 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/ubi10/ubi registry.example.com:5000/ubi10/ubi
```

2. Push the **registry.example.com:5000/ubi10/ubi** image from your local storage to a registry:

```
# podman push registry.example.com:5000/ubi10/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.

4.12. REMOVING IMAGES

Delete unused container images from local storage using the **podman rmi** command. Removing old images frees up disk space on your system.

Prerequisites

- The **container-tools** meta-package is installed.

Procedure

1. List all images on your local system:

```
$ podman images
REPOSITORY                           TAG     IMAGE ID      CREATED       SIZE
registry.redhat.io/rhel10/support-tools   latest  4b32d14201de  7 weeks ago  228 MB
registry.redhat.io/ubi10/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/rhel10/support-tools:latest  usr/bin/bash  6 seconds ago  Up
5 seconds ago          my-support-tools
```

To remove the **registry.redhat.io/rhel10/support-tools** image, you have to stop all containers running from this image by using the **podman stop** command. You can stop a container by its ID or name.

3. Stop the **my-support-tools** container:

```
$ podman stop my-support-tools
7ccd6001166e9720c47fbeb077e0afd0bb635e74a1b0ede3fd34d09eaf5a52e9
```

4. Remove the **registry.redhat.io/rhel10/support-tools** image:

```
$ podman rmi registry.redhat.io/rhel10/support-tools
```

- To remove multiple images:

```
$ podman rmi registry.redhat.io/rhel10/support-tools registry.redhat.io/ubi10/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...
```

Verification

- List all images by using the **podman images** command to verify that container images were removed.

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

Start a new container from an image by using the **podman run** command. This command pulls the image if necessary and launches the container process with its own isolated file system and network.

The **podman run** command runs a process in a new container based on a container image. If the container image is not already loaded, **podman run** pulls the image and all its dependencies from the repository, as if running **podman pull *image***, before starting the container from that image. The container process has its own file system, networking, and an 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

Enter commands within a container directly from the host by using **podman run**. With this command, you can inspect the container environment or run utilities without entering an interactive shell.

Prerequisites

- The **container-tools** meta-package is installed.

Procedure

1. Display the type of operating system of the container based on the **registry.access.redhat.com/ubi10/ubi** container image using the **cat /etc/os-release** command:

   ```
   $ podman run --rm registry.access.redhat.com/ubi10/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 10"
...
```

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.

5.3. RUNNING COMMANDS INSIDE THE CONTAINER

Start an interactive session inside a container by using **podman run -it**. After you use this command, you can work within the container's shell to enter commands and troubleshoot.

Prerequisites

- The **container-tools** meta-package is installed.

Procedure

1. Run the container named **myubi** based on the **registry.redhat.io/ubi10/ubi** image:

```
$ podman run --name=myubi -it registry.access.redhat.com/ubi10/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 /]# dnf 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 -a
CONTAINER ID  IMAGE                          COMMAND    CREATED        STATUS
PORTS   NAMES
1984555a2c27  registry.redhat.io/ubi10/ubi:latest  /bin/bash  21 minutes ago  Exited (0) 21
minutes ago         myubi
```

   You can see that the container is in Exited status.

5.4. BUILDING A CONTAINER

Build new container images by using Buildah or **podman build**. You can define image instructions in a Containerfile to automate software installation and configuration.

Prerequisites

1. The **container-tools** meta-package is installed.

Procedure

1. Install Container Tools: Ensure the necessary container tools are installed on your RHEL system. The container-tools module provides Buildah, Podman, and Skopeo.

```
$ sudo dnf install container-tools
```

2. Create a Containerfile: A Containerfile defines the instructions for building your container image. This file specifies the base image, any software to install, configurations to apply, and the application to run. For example:

```
FROM registry.redhat.io/ubi10/ubi-minimal
RUN microdnf -y update && microdnf -y install
COPY index.html /var/www/html/
EXPOSE 80
CMD ["httpd", "-DFOREGROUND"]
```

3. Build the container image with Buildah: Use **buildah bud** (or **podman build**) to build the image after you navigate to the directory containing your Container file.

```
$ cd /<path_to_container_file>

$ buildah bud -t your_image_name:tag .
```

   - **your_image_name**: The name for your image.
   - **tag**: The tag for your image (e.g., latest, 1.0).
   - **.**: Indicates that the Containerfile is in the current directory.

4. Run the container: After you build the image, you can run a container from it using the **podman run** command.

```
$ podman run -d -p 8080:80 my-web-app
```

- **-d**: Runs the container in detached mode (in the background).
- **-p 8080:80**: Maps port 8080 on the host to port 80 inside the container.
- **my-web-app**: The name of the image to run.

The heredocs syntax in container buildings

You can use the **heredoc** syntax in Containerfile, with a Red Hat Enterprise Linux base image, ensuring you enable **BuildKit**. If the commands contain **heredoc** syntax, the Containerfile considers the next lines, until the line only contains a heredoc delimiter, as part of the same command. You can embed multi-line strings directly within instructions like **RUN** or **COPY** in Containerfile using **heredocs**. This is especially useful with RHEL-based images, as it removes the need to create separate script files for simple tasks and thus improves readability and maintainability.
For Example, a common use case is running multiple shell commands in a single **RUN** instruction to create a single image layer, avoiding the **&&** \ syntax:

```
# syntax=container/containerfile:1.4
FROM registry.redhat.io/ubi10/ubi-minimal
# Use a heredoc to perform a multi-line RUN command:
RUN <<EOF
microdnf -y update
microdnf -y install nginx
microdnf clean all
echo "Nginx installed and packages updated"
EOF
```

- **RUN <<EOF**: The **<<** signals the start of the heredoc, and **EOF** is the user-defined delimiter.
- The lines between the **<<EOF** and the final **EOF** are treated as a single script executed by the shell.
- The entire block is a single **RUN** instruction, which is more efficient and easier to read.

5.5. LISTING CONTAINERS

View the status of containers on your system by using the **podman ps** command. You can list currently running containers or use the **-a** option to see all containers, including stopped ones.

Prerequisites

- The **container-tools** meta-package is installed.

Procedure

1. Run the container based the on **registry.redhat.io/rhel10/support-tools** image:

```
$ podman run -dt registry.redhat.io/rhel10/support-tools
```

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

```
$ podman ps
CONTAINER ID IMAGE            COMMAND        CREATED      STATUS
PORTS NAMES
74b1da000a11 rhel10/support-tools /usr/bin/bash 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 ubi10/ubi      /bin/bash  3 secs ago Exited (0) 5 secs ago peaceful_hopper
  false
  74b1da000a11 rhel10/support-tools /usr/bin/bash 2 mins ago Up About a minute
  musing_brown    false
  ```

  If you do not remove containers that are not running, by using the **--rm** option, you can restart them.

  For more information, see the **podman-images(1)** man page on your system.

5.6. 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 **container-tools** meta-package is installed.
- 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 **-i** (**--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
   ```

For more information, see the **podman-start(1)** man page on your system.

Additional resources

- Starting a container using Podman Desktop

5.7. 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 **container-tools** meta-package is installed.

Procedure

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

```
$ podman inspect 64ad95327c74
[
    {
        "Id":
"64ad95327c740ad9de468d551c50b6d906344027a0e645927256cd061049f681",
        "Created": "2025-05-05T18:15:56.743553292+02:00",
          "Path": "/usr/bin/bash",
          "Args": [
               "/usr/bin/bash"
          ],
          "State": {
               "OciVersion": "1.2.0",
            "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
  For more information, see the **podman-inspect(1)** man page on your system.

5.8. MOUNTING DIRECTORY ON LOCALHOST TO THE CONTAINER

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 **container-tools** meta-package is installed.

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/ubi10/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.

5.9. MOUNTING A CONTAINER FILESYSTEM

Share host directories or devices with a container by mounting them. For example, mounting **/dev/log** allows the container to write logs directly to the host’s system journal.

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

Prerequisites

- The **container-tools** meta-package is installed.

Procedure

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

```
# podman run -dt --name=mysyslog registry.redhat.io/rhel10/support-tools
```

2. Optional: List all containers:

```
# podman ps -a
CONTAINER ID  IMAGE                                    COMMAND        CREATED         STATUS
PORTS   NAMES
c56ef6a256f8  registry.redhat.io/rhel10/support-tools:latest  usr/bin/bash  20 minutes ago  Up
20 minutes ago                mysyslog
```

3. Mount the **mysyslog** container:

```
# podman mount mysyslog
/var/lib/containers/storage/overlay/990b5c6ddcdeed4bde7b245885ce4544c553d108310e2b797
d7be46750894719/merged
```

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

```
# ls
/var/lib/containers/storage/overlay/990b5c6ddcdeed4bde7b245885ce4544c553d108310
e2b797d7be46750894719/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/990b5c6ddcdeed4bde7b245885ce4544c553d108310
e2b797d7be46750894719/merged/etc/os-release
NAME="Red Hat Enterprise Linux"
VERSION="10 (Ootpa)"
ID="rhel"
ID_LIKE="fedora"
...
```

For more information, see the **podman-mount(1)** man page on your system.

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

Access a container's root file system from the host by using the **podman mount** command. With this command, you can inspect or modify files within the container directly from the host.

The following example runs the **support-tools** 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 **container-tools** meta-package is installed.

Procedure

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

```
# podman run -d --ip=10.88.0.44 registry.redhat.io/rhel10/support-tools
efde5f0a8c723f70dd5cb5dc3d5039df3b962fae65575b08662e0d5b5f9fbe85
```

2. Check that the IP address is set properly:

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

For more information, see the **podman-inspect(1)** and **podman-run(1)** man pages on your system.

5.11. 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 **container-tools** meta-package is installed.
- The container is running.

Procedure

1. Enter the **rpm -qa** command inside the **my-support-tools** container to list all installed packages:

   ```
   $ podman exec -it my-support-tools rpm -qa
   gpg-pubkey-fd431d51-4ae0493b
   gpg-pubkey-5a6340b3-6229229e
   libgcc-11.5.0-2.el9.x86_64
   setup-2.13.7-10.el9.noarch
   ...
   ```

2. Enter a **/bin/bash** command in the **my-support-tools** container:

   ```
   $ podman exec -it my-support-tools /bin/bash
   ```

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

   ```
   # dnf 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           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
     tmpfs           6.3G  448K  6.3G   1% /etc/hosts
     shm              63M     0   63M   0% /dev/shm
     overlay         953G   76G  877G   8% /
     ```

```
tmpfs           64M     0   64M   0% /dev
devtmpfs        4.0M     0  4.0M   0% /dev/tty
...
```

- To display system information:

```
# uname -r
6.12.0-124.15.1.el10_1.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
```

5.12. 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 **container-tools** meta-package is installed.

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/ubi10/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/ubi10/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
```

For more information, see the **podman-volume(1)** man page on your system.

5.13. EXPORTING AND IMPORTING CONTAINERS

You can use the **podman export** command to export the file system of a running container to a tar file 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 to revert back to it later.

Also, you can use the **podman import** command to import a tar file 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 **container-tools** meta-package is installed.

Procedure

1. Run the **myubi** container based on the **registry.access.redhat.com/ubi10/ubi** image:

```
$ podman run -dt --name=myubi registry.access.redhat.com/10/ubi
```

2. Optional: List all containers:

```
$ podman ps -a
CONTAINER ID  IMAGE                              COMMAND         CREATED     STATUS
PORTS   NAMES
a6a6d4896142  registry.access.redhat.com/10: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 myubi-imported cat testfile
hello
```

For more information, see the **podman-export (1)** and **podman-import(1)** man pages on your system.

5.14. 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 **container-tools** meta-package is installed.
- At least one container is running.

Procedure

- Stop the **myubi** container:
  - By using the container name:

    ```
    $ podman stop myubi
    ```

  - By 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. The following example command sends a SIGHUP signal to a container. If supported by the application, the signal causes the application to re-read its configuration files:

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

For more information, see the **podman-stop(1)** and **podman-kill(1)** man pages on your system.

Additional resources

- Managing containers and pods by using Podman Desktop

5.15. REMOVING CONTAINERS

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

Prerequisites

- The **container-tools** meta-package is installed.
- 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 ubi10/ubi      /bin/bash  3 secs ago Exited (0) 5 secs ago peaceful_hopper
false
74b1da000a11 rhel10/support-tools usr/bin/bash 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
     ```

   - To remove multiple containers:

     ```
     $ podman rm clever_yonath furious_shockley
     ```

- To remove all containers from your local system:

  ```
  $ podman rm -a
  ```

Verification

- List all images by using the **podman ps -a** command to verify that containers were removed.

5.16. CREATING SELINUX POLICIES FOR CONTAINERS

To generate SELinux policies for containers, use the Udica tool. For more information, see the Creating SELinux policies for containers chapter in the RHEL Using SELinux document.

5.17. CONFIGURING PRE-EXECUTION HOOKS IN PODMAN

You can create plugin scripts to define a fine-control over container operations, especially blocking unauthorized actions, for example pulling, running, or listing container images.

NOTE

The file **/etc/containers/podman_preexec_hooks.txt** must be created by an administrator and can be empty. If the **/etc/containers/podman_preexec_hooks.txt** does not exist, the plugin scripts will not be executed.

The following rules apply to the plugin scripts:

- Have to be root-owned and not writable.
- Have to be located in the **/usr/libexec/podman/pre-exec-hooks** and **/etc/containers/pre-exec-hooks** directories.
- Execute in sequentially and alphanumeric order.
- If all plugin scripts return zero value, then the **podman** command is executed.
- If any of the plugin scripts return a non-zero value, it indicates a failure. The **podman** command exits and returns the non-zero value of the first-failed script.
- Red Hat recommends to use the following naming convention to execute the scripts in the correct order: ***DDD_name.lang***, where:
  - The ***DDD*** is the decimal number indicating the order of script execution. Use one or two leading zeros if necessary.
  - The ***name*** is the name of the plugin script.
  - The ***lang*** (optional) is the file extension for the given programming language. For example, the name of the plugin script can be: **001-check-groups.sh**.

NOTE

The plugin scripts are valid at the time of creation. Containers created before plugin scripts are not affected.

Prerequisites

- The **container-tools** meta-package is installed.

Procedure

- Create the script plugin named **001-check-groups.sh**. For example:

  ```
  #!/bin/bash
  if id -nG "$USER" 2> /dev/null | grep -qw "$GROUP" 2> /dev/null ; then
      exit 0
  else
      exit 1
  fi
  ```

  - The script checks if a user is in a specified group.
  - The **USER** and **GROUP** are environment variables set by Podman.
  - Exit code provided by the **001-check-groups.sh** script would be provided to the **podman** binary.
  - The **podman** command exits and returns the non-zero value of the first-failed script.

Verification

- Check if the **001-check-groups.sh** script works correctly:

  ```
  $ podman run image
  ...
  ```

  If the user is not in the correct group, the following error appears:

  ```
  external preexec hook /etc/containers/pre-exec-hooks/001-check-groups.sh failed
  ```

5.18. DEBUGGING APPLICATIONS IN CONTAINERS

You can use various command-line tools tailored to different aspects of troubleshooting. For more information, see Debugging applications in containers.

CHAPTER 6. INTRODUCTION TO REPRODUCIBLE CONTAINER BUILDS

Create identical container images from the same source code by using reproducible builds with Podman and Buildah. This consistency helps ensure supply chain security and reduces storage and bandwidth usage.

Fewer changes between images can decrease the amount of data you need to pull from a registry when moving from one version of an image to a newer version of that image. Reproducible container builds are crucial for ensuring supply chain security, facilitating reliable software deployment, and facilitating effective debugging.

Previously, the escalating size of container images and increased customer apprehension regarding update payload sizes amplify the existing challenges and limitations associated with tarball creation. In systems such as Konflux, each Git commit generates new tarballs, which necessitates a complete re-download of the entire image. Factors such as mtime changes mean that even when users install the same RPMs, storage requirements double, despite no alteration in the underlying data. This not only burdens registry storage but also forces clients to pull a new layer, even if no changes have occurred. This situation further exacerbates problems in environments such as **rhel-bootc** and RHEL AI, which prioritize faster updates.

6.1. BENEFITS OF REPRODUCIBLE CONTAINER BUILDS

Reproducible builds for RHEL containers reduce registry storage, create smaller update payloads, and enable faster downloads by ensuring the image layers remain consistent when the source code and build environment are unchanged.

This process maximizes the efficiency of layer caching, preventing the redundant storage and transfer of identical data. The notable benefits of reproducible container builds are:

- Reduced registry storage: When an image is updated, only changed layers are stored. Reproducible builds ensure identical images from the same source code by preventing non-deterministic factors (timestamps, file order, metadata) from causing changes, thus avoiding additional storage.
- Efficient and smaller update payloads: For container minor updates, for example, security patches, only the changed layer needs to be downloaded, not the whole image. Reproducible builds also ensure that only affected layers change with source updates, unlike non-reproducible builds where small code changes can alter multiple layers.
- Faster downloads: Container reproducible builds optimize both build systems and end users by enabling faster downloads through efficient caching, and reduced network traffic.

6.2. IMPACT OF REPRODUCIBLE CONTAINER BUILDS ON DIFFERENT ENVIRONMENTS

Ensure consistent, identical container images across environments such as Konflux and bootc with reproducible builds. This verification helps maintain supply chain integrity and simplifies compliance and debugging.

Reproducible container builds on RHEL ensure consistent, identical container images regardless of build time or location. The impact of reproducible container builds on different environments are:

Konflux

- Enhanced software supply chain integrity: Reproducible container builds enhance Konflux's mission of delivering a secure and transparent software supply chain. Konflux uses reproducible builds to verify that a built container image derives exactly from its source code. Any third party can rebuild the container from the same inputs and verify that the output is bit-for-bit identical. Also, RHEL reproducible container builds protect against "in-transit" vulnerabilities, where an attacker can compromise a distribution mirror or inject malicious code into the build process. Konflux can prove that the released binary matches its source, mitigating attacks on its own build infrastructure.
- Improved compliance and transparency: Konflux enforces SLSA security policies. It verifies the origin and provenance of reproducible RHEL images, simplifying compliance. Konflux uses Tekton Chains to create an immutable, signed attestation that documents the entire build process. RHEL's reproducible container builds add a foundational layer of trust to this attestation by ensuring the base image is trustworthy and verifiably built.
- Development and security workflows: Reproducible builds guarantee consistent container image digests across multiple runs, simplifying testing and debugging. Konflux leverages this to efficiently scan and update vulnerable packages in RHEL containers. Konflux uses verified attestations to automatically block noncompliant builds and enforce security policies without reducing flexibility.

Bootc

- Verifiable supply chain and enhanced security: Reproducible RHEL container builds enhance rhel-bootc by creating a more secure, reliable, and transparent build process for bootable OS images. You can verify that a specific bootc image was built from its claimed source code, which makes it more difficult for attackers to inject malicious code into a container image by compromising a build pipeline.
- Streamlined CI/CD and GitOps Workflows: You can use reproducibility to manage their entire OS configuration and application stack using Git-based workflows (GitOps). A change to the Containerfile, guarantees a consistent bootable image across all environments. Reproducible builds form a cornerstone of automated CI/CD pipelines.

RHEL AI

- Reproducible container builds are critical for RHEL AI because they provide the foundational consistency, security, and efficiency AI model development and deployment need. RHEL AI delivers a bootable container image, which means it manages the operating system itself as a container artifact. Reproducibility ensures that this base AI environment is always consistent and trustworthy.

6.3. ACHIEVING REPRODUCIBILITY IN RHEL CONTAINER TOOLS

Achieve build reproducibility by using RHEL container tools such as Buildah, Podman, and Skopeo. These tools offer options to control timestamps and other non-deterministic factors during the build process.

The RHEL container tools provide a standardized, daemonless, and scriptable workflow to achieve reproducibility. This approach ensures that a container built once can run consistently anywhere, addressing potential issues with dependencies, environments, and versioning.

Buildah

RHEL Buildah achieves reproducible container builds by providing granular control over the build process. It offers specific options to mitigate sources of non-determinism, such as unstable tags, filesystem metadata, and host-dependent data. Buildah uses fixed timestamps for reproducible builds. Standard timestamps cause major irreproducibility. By default, file creation and modification times reflect when someone adds a file to a container layer, which is never the same twice. Buildah allows you to zero these timestamps or set them to a specific, fixed value. You can use the following options for reproducibility:

- **-–rewrite-timestamp**: This option timestamps the contents of layers to be no later than the **--source-date-epoch**. Also, controls the created timestamp of an image and the timestamps of files within its layers, primarily to achieve deterministic builds.
- **--source-date-epoch**: This option is more flexible option than the older **--timestamp** option, allowing you to define a specific, reproducible timestamp for all files in the image layer. It affects creation and history dates in image metadata. You can set it by using CLI flag, environment variable, or as a build-arg. When the flag is set, declared ARGs are exposed in the environment for **RUN** instructions and get static hostname. Also, the container ID field is cleared in the committed image.

Podman

The **podman build** command, while the user-facing interface, delegates the actual image creation to the Buildah library. This means that Podman achieves reproducible container builds by leveraging the same core features as Buildah, with a focus on controlling sources of non-determinism during the build process.
The Podman commands also accept the **-–rewrite-timestamp** and **--source-date-epoch** options. Additionally, the **--no-cache** option instructs Podman to disregard its local cache and perform a fresh build. Using this option helps verify that your container image can be reliably ruced from scratch.

Skopeo

Skopeo achieves reproducible container builds by referencing immutable image digests instead of mutable tags. Skopeo primarily transports and manages images, while other tools like Buildah handle the actual reproducible image creation.

Using the **--source-date-epoch** and **--rewrite-timestamp** options can improve build reproducibility. However, complete reproducibility is not guaranteed. Content added from other images with the **COPY** instructions's **--from** option, accessed through the **RUN** instruction's **--mount=from=** option, or downloaded using the **ADD** instruction can change if you reference an image tag that later moves, or if the content at the specified URL changes.

6.4. WORKING WITH REPRODUCIBLE CONTAINER BUILDS

Build reproducible container images by specifying timestamp options in Podman or Buildah. By using the **-–source-date-epoch** and **-–rewrite-timestamp** options, you can create deterministic images that are identical across builds.

Procedure

- To set these options when running your **Buildah** command. For example, to build an image from a Containerfile and force all timestamps to a specific point in time:

```
Use a specific, immutable image.
FROM registry.access.redhat.com/ubi10/ubi:10.0 AS builder

# Set the SOURCE_DATE_EPOCH for deterministic timestamps
```

```
ARG SOURCE_DATE_EPOCH
ENV SOURCE_DATE_EPOCH=${SOURCE_DATE_EPOCH:-1}

# Build the image using the build-arg and rewrite-timestamp options
buildah bud --source-date-epoch=${SOURCE_DATE_EPOCH} \
  --rewrite-timestamp \
  -f Containerfile \
  -t my-reproducible-image .
```

- Run the **podman build** command with a consistent timestamp to create the reproducible image:

```
# Set a consistent timestamp using the last Git commit date
export SOURCE_DATE_EPOCH=$(git log -1 --pretty=%ct)
```

- Build the image with the specified timestamp:

```
podman build --source-date-epoch=${SOURCE_DATE_EPOCH} --rewrite-timestamp -t my-
reproducible-app .
```

Verification

- After building, run the reproducible command again. If the build is truly reproducible, the **buildah inspect** command should show the same image digest.

```
buildah bud --source-date-epoch=${SOURCE_DATE_EPOCH} \ --rewrite-timestamp \ -f
Containerfile \ -t my-reproducible-image-2 .
```

- Compare the digests:

```
buildah inspect --format '{{.Digest}}' my-reproducible-image
buildah inspect --format '{{.Digest}}' my-reproducible-image-2
```

CHAPTER 7. ADDING SOFTWARE TO A UBI CONTAINER

Enhance Universal Base Images (UBI) by installing additional software from Red Hat repositories. You can add packages to running containers by using **dnf** or **microdnf** depending on the image type.

Red Hat builds UBIs from a subset of RHEL content. UBIs also provide a subset of RHEL packages, which are freely available to install for use with UBI. You can use the DNF repositories, which include RPM packages and updates, to add or update software to a running container. UBIs provide a set of pre-built language runtime container images,for example, 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 **dnf** command
- On UBI minimal images, use the **microdnf** command

7.1. USING THE UBI INIT IMAGES

Build containers that run system services by using the UBI init image. This allows you to configure applications such as web servers to start automatically through systemd within the container.

You can build a container by 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 by 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.

Prerequisites

- The **container-tools** meta-package is installed.

Procedure

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

```
FROM registry.access.redhat.com/ubi10/ubi-init
RUN dnf -y install httpd; dnf 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

7.2. USING THE UBI MICRO IMAGES

Build ultra-lightweight containers by using the UBI micro image and Buildah. This process involves mounting the image and installing software into it from the host, as the image lacks a package manager.

Prerequisites

- The **container-tools** meta-package is installed.

Procedure

1. Pull and build the **registry.access.redhat.com/ubi10/ubi-micro** image:

```
# microcontainer=$(buildah from registry.access.redhat.com/ubi10/ubi-micro)
```

2. Mount a working container root filesystem:

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

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

```
# dnf install \
    --installroot $micromount \
    --releasever=/ \
    --setopt install_weak_deps=false \
    --setopt=reposdir=/etc/yum.repos.d/ \
    --nodocs -y \
    httpd
# dnf 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

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
```

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

Access full RHEL repositories inside UBI containers when running on a subscribed RHEL host. This allows you to install a wider range of packages beyond what is available in the standard UBI repositories.

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.

- Red Hat entitlements are passed from a subscribed Red Hat host as a secrets mount defined in **/usr/share/containers/mounts.conf** on the host running Podman.
  Verify the mounts configuration:

```
$ cat /usr/share/containers/mounts.conf
/usr/share/rhel/secrets:/run/secrets
```

- The **yum**, **dnf**, and **microdnf** commands should search for entitlement data at this path.

- If the path is not present, the commands cannot use Red Hat entitled content, such as the RHV repositories, because they lack the keys or content access the host has.
- This is applicable only for Red Hat shipped or provided Podman on a RHEL host.
- If you installed Podman not shipped by Red Hat, follow the instructions in How do I attach subscription data to containers running in Docker not provided by Red Hat? article.

7.4. ADDING SOFTWARE IN A STANDARD UBI CONTAINER

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

Prerequisites

- The **container-tools** meta-package is installed.

Procedure

1. Pull and run the **registry.access.redhat.com/ubi10/ubi** image:

```
$ podman run -it --name myubi registry.access.redhat.com/ubi10/ubi
```

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

   ```
   # dnf install --disablerepo=*--enablerepo=ubi-10-for-x86_64-appstream-rpms --enablerepo=ubi-10-for-x86_64-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:

   ```
   # dnf install zsh
   ```

   - To add a package that is in a different host repository, explicitly enable the repository you need. For example, to install the **python3-devel** package from the **codeready-builder-for-ubi-10-x86_64-rpms** repository:

   ```
   # dnf install --enablerepo=codeready-builder-for-ubi-10-x86_64-rpms python3-devel
   ```

Verification

1. List all enabled repositories inside the container:

```
# dnf 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.

7.5. ADDING SOFTWARE IN A MINIMAL UBI CONTAINER

Install software in minimal UBI containers by using the **microdnf** command. This command provides essential package management features while keeping the container size small.

Prerequisites

- The **container-tools** meta-package is installed.

Procedure

1. Pull and run the **registry.access.redhat.com/ubi10/ubi-minimal** image:

   ```
   $ podman run -it --name myubimin registry.access.redhat.com/ubi10/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 --setopt install_weak_deps=0
     ```

   - To add a package that is in a different host repository, explicitly enable the repository you need. For example, to install the **python3-devel** package from the **codeready-builder-for-rhel-10z-x86_64-rpms** repository:

     ```
     # microdnf install --enablerepo=codeready-builder-for-rhel-10-x86_64-rpms
     python3-devel --setopt install_weak_deps=0
     ```

     The **--setopt install_weak_deps=false** option disables the installation of weak dependencies. Weak dependencies include recommended or suggested packages that are not strictly required but are often installed by default.

Verification

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.

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

Add software to UBI containers on unsubscribed hosts by using the default UBI repositories. These repositories are freely available and do not require a RHEL subscription to access.

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

Prerequisites

- The **container-tools** meta-package is installed.

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 **dnf 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/ubi10/ubi
    # dnf install bzip2
    ```

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

    ```
    $ podman run -it --name myubimin registry.access.redhat.com/ubi10/ubi-minimal
    # microdnf install bzip2
    ```

Verification

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

     ```
     #  dnf 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.

7.7. BUILDING UBI-BASED IMAGES

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

NOTE

For UBI minimal images, use **microdnf** instead of **dnf**: **RUN microdnf update -y && rm -rf /var/cache/yum** and **RUN microdnf install httpd -y && microdnf clean all** commands.

Prerequisites

- The **container-tools** meta-package is installed.

Procedure

1. Create a **Containerfile**:

```
FROM registry.access.redhat.com/ubi10/ubi
USER root
LABEL maintainer="John Doe"
# Update image
RUN dnf update --disablerepo=* --enablerepo=ubi-10-appstream-rpms --enablerepo=ubi-10-
baseos-rpms -y && rm -rf /var/cache/yum
RUN dnf install --disablerepo=* --enablerepo=ubi-10-appstream-rpms --enablerepo=ubi-10-
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/ubi10/ubi:latest
STEP 2: USER root
STEP 3: LABEL maintainer="John Doe"
STEP 4: RUN dnf update --disablerepo=* --enablerepo=ubi-10-appstream-rpms --
enablerepo=ubi-10-baseos-rpms -y
...
Writing manifest to image destination
Storing signatures
--> f9874f27050
f9874f270500c255b950e751e53d37c6f8f6dba13425d42f30c2a8ef26b769f2
```

Verification

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
```

7.8. USING APPLICATION STREAM RUNTIME IMAGES

Use Application Stream runtime images as a foundation for your applications. These images provide pre-built environments for languages such as Python, Ruby, and Node.js.

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

7.9. GETTING UBI CONTAINER IMAGE SOURCE CODE

Access the source code for UBI images by downloading the corresponding source container image. Use Skopeo to retrieve and unpack these images to inspect the underlying source.

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 10 container **registry.access.redhat.com/ubi10:8.1-397** append **-source** to get the source container image (**registry.access.redhat.com/ubi10:8.1-397-source**).

Prerequisites

- The **container-tools** meta-package is installed.

Procedure

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

```
$ skopeo copy \
docker://registry.access.redhat.com/ubi10:8.1-397-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.

For more information, see the **skopeo-copy (1)** and **skopeo-inspect(1)** man pages on your system.

CHAPTER 8. WORKING WITH PODS

Group related containers together into a pod to share namespaces and resources. Pods allow you to manage multiple containers as a single unit.

Containers are the smallest manageable unit with Podman, Skopeo, and Buildah. A Podman pod, similar to a Kubernetes pod, groups one or more containers. Pods are the smallest compute units in OpenShift or Kubernetes. Each Podman pod includes an infra container, which manages namespaces and allows other containers to connect, start, and stop, keeping the pod running. The default infra container on the **registry.access.redhat.com/ubi10/pause** image.

8.1. CREATING PODS

Create a new pod using the **podman pod** create command. You can then add containers to this pod, allowing them to share network and storage resources.

Prerequisites

- The **container-tools** meta-package is installed.

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/ubi10/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/ubi10/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/ubi10/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/ubi10/ubi:latest  /bin/bash  Less than a second ago
   Up Less than a second ago        myubi              223df6b390b4
   3afdcd93de3e  registry.access.redhat.com/ubi10/pause                             Less than a
   second ago  Up Less than a second ago        223df6b390b4-infra  223df6b390b4
   ```

Additional resources

- Creating a pod using Podman Desktop

8.2. DISPLAYING POD INFORMATION

View details about your pods, including status and associated containers. This helps you monitor the health and configuration of your pod groups.

NOTE

Beginning with Podman v5.0.0, pod output is always a JSON array, regardless of the number of pods.

For more information, see the **podman-pod-top(1)**, **podman-pod-stats(1)**, and **podman-pod-inspect(1)** man pages on your system.

Prerequisites

- The **container-tools** meta-package is installed.
- 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 --format json <POD_ID_1> <POD_ID_2> <POD_ID_3>
  [
    {
        "CgroupParent": "/libpod_parent",
        "Containers": [
          {
            "ID": "...",
            "State": "..."
          }
        ],
        "Created": "2025-10-16T12:00:00.000000000Z",
        "ID": "673f326c9f69b0d24c0847f97a544c79532817d2a713917812f865f1e8e52a8a",
        "InfraContainerID": "...",
        "Labels": {},
        "Name": "web_pod",
        "State": "Running",
      },
      {
        "CgroupParent": "/libpod_parent",
        "Containers": [
          {
            "ID": "...",
            "State": "..."
          }
        ],
        "Created": "2025-10-16T12:01:00.000000000Z",
        "ID": "a1b2c3d4e5f678901234567890abcdef1234567890abcdef1234567890abcdef",
        "InfraContainerID": "...",
        "Labels": {},
        "Name": "db_pod",
        "State": "Running",
      }
  ]
  ```

  You can see information about containers in the pod.

8.3. STOPPING PODS

Stop an entire pod and all its contained containers by using the **podman pod stop** command. This helps ensure that all related services within the pod terminate together.

Prerequisites

- The **container-tools** meta-package is installed.

- 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/ubi10/ubi:latest  /bin/bash  About a minute ago  Exited (0) 7
seconds ago         myubi              223df6b390b4  mypod

3afdcd93de3e  registry.access.redhat.com/10/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

- Managing containers and pods using Podman Desktop

8.4. REMOVING PODS

Delete stopped pods by using the **podman pod rm** command. This removes the pod definition and all associated containers from your system.

Prerequisites

- The **container-tools** meta-package is installed.
- 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
```

For more information, see the **podman-pod-rm(1)** man page on your system.

Additional resources

- Managing containers and pods using Podman Desktop

CHAPTER 9. MANAGING A CONTAINER NETWORK

Manage container networking to control communication between containers and external systems. With Podman, you can create, inspect, and manage networks to isolate traffic and help maintain security.

Container networking is managed by configuring and controlling how containers communicate with each other and external systems. Tools such as Podman enable the creation and operation of lightweight, isolated containers.

9.1. LISTING CONTAINER NETWORKS

View available container networks by using the **podman network ls** command. This displays the network names, drivers, and other configuration details for both root and rootless environments.

Podman works with 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.

Prerequisites

The **container-tools** meta-package is installed.

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.
    For more information, see the **podman-network-ls(1)** man page on your system.

9.2. INSPECTING A NETWORK

View detailed configuration of a specific network by using the **podman network inspect** command. This reveals settings like subnets, gateways, and enabled plugins.

Prerequisites

- The **container-tools** meta-package is installed.

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.

For more information, see the **podman-network-inspect(1)** man page on your system.

9.3. CREATING A NETWORK

Create custom networks for your containers by using the **podman network create** command. You can configure external access or create isolated internal networks for secure container communication.

By default, Podman creates an external network. You can create an internal network in which containers can communicate with other containers on the host, but cannot connect to the network outside the host nor be reached from it.

Prerequisites

- The **container-tools** meta-package is installed.

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 by using the **podman network create** command.

Refer to **podman-network-create(1)** man page on your system for more information.

9.4. CONNECTING A CONTAINER TO A NETWORK

Attach a running container to an additional network by using the **podman network connect** command. This allows the container to communicate on multiple networks simultaneously.

Prerequisites

- The **container-tools** meta-package is installed.
- A network has been created by 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.

9.5. DISCONNECTING A CONTAINER FROM A NETWORK

Detach a container from a specific network by using the **podman network disconnect** command. This stops the container from communicating on that network without stopping the container itself.

Prerequisites

- The **container-tools** meta-package is installed.
- A network has been created by 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.

Refer to **podman-network-disconnect(1)** man page on your system for more information.

9.6. REMOVING A NETWORK

Delete unused networks by using the **podman network rm** command. Note that you cannot remove a network if it is currently in use by any containers.

Prerequisites

- The **container-tools** meta-package is installed.

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
```

For more information, see **podman-network-rm(1)** man page on your system.

9.7. 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.

Prerequisites

- The **container-tools** meta-package is installed.

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
```

For more information, see the **podman-network-prune(1)** man page on your system.

CHAPTER 10. COMMUNICATING AMONG CONTAINERS

Enable communication between containers, the host, and external applications. You can use methods such as port mapping, DNS resolution, and pod networking to manage connectivity.

10.1. THE NETWORK MODES AND LAYERS

Understand the available network modes in Podman, such as bridge, host, and slirp4netns. Choosing the right mode determines how your container interacts with the network stack.

Podman uses the following network modes and options:

- **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
- **pasta** - high performance replacement for **slirp4netns**. You can use **pasta** beginning with Podman v4.4.1.
- **none** - create a network namespace for the container but do not configure network interfaces for it. The container has no network connectivity.
- **ns:<path>** - path to a network namespace to join

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.

10.2. DIFFERENCES BETWEEN SLIRP4NETNS AND PASTA

Understand the key differences between the **pasta** and **slirp4netns** network modes.

Notable differences of **pasta** network mode compared to **slirp4netns** include:

- **pasta** supports IPv6 port forwarding.
- **pasta** is more efficient than **slirp4netns**.
- **pasta** copies IP addresses from the host, while slirp4netns uses a predefined IPv4 address.
- **pasta** uses an interface name from the host, while slirp4netns uses tap0 as interface name.
- **pasta** uses the gateway address from the host, while **slirp4netns** defines its own gateway address and uses NAT.

NOTE

The default network mode for rootless containers is **pasta**.

10.3. SETTING THE NETWORK MODE

Specify the network configuration for a container by using the **podman run** command with the **--network** option during startup. You can select modes such as host networking or attach to a custom bridge.

Prerequisites

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

Procedure

1. Optional: If you want to use the **pasta** network mode, install the **passt** package:

```
$ {PackageManager} install passt
```

2. Run the container based on the **registry.access.redhat.com/ubi10/ubi** image:

```
$ podman run --network=<netwok_mode> -d --name=myubi
registry.access.redhat.com/ubi10/ubi
```

The ***<netwok_mode>*** is the required network mode. Alternatively, you can use the **default_rootless_network_cmd** option in the **containers.conf** file to switch the default network mode.

NOTE

The default network mode for rootless containers is **pasta**.

Verification

- Verify the setting of the network mode:

```
$ podman inspect --format {{.HostConfig.NetworkMode}} myubi
<netwok_mode>
```

10.4. 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.

Prerequisites

- The **container-tools** meta-package is installed.

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>
```

10.5. COMMUNICATING BETWEEN A CONTAINER AND AN APPLICATION

Establish communication between a container and an external application or another container. This enables services to exchange data over the network.

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 by using these networks. By default, the web server listens on port 80. In this example procedure, the **myubi** container communicates with the **web-container** application.

Prerequisites

- The **container-tools** meta-package is installed.

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 ubi10/ubi curl 10.88.0.2:80
```

10.6. COMMUNICATING BETWEEN A CONTAINER AND A HOST

Communicate from the host to a container by using the container's IP address on the bridge network. This allows for accessing services running inside containers directly from the 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 **container-tools** meta-package is installed.
- The **web-container** is running. For more information, see Communicating between a container and an application.

Procedure

1. Verify that the bridge is configured:

   ```
   # podman network inspect podman | grep bridge

     "type": "bridge"
   ```

2. Display the host network configuration:

   ```
   # ip addr show cni-podman0

   6: 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 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 **podman0** subnet 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
   ```

10.7. COMMUNICATING BETWEEN CONTAINERS USING PORT MAPPING

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

Prerequisites

- The **container-tools** meta-package is installed.

Procedure

1. Run the unpublished container:

```
# podman run -dt --name=web1 ubi10/httpd-24
```

2. Run the automatically published container:

```
# podman run -dt --name=web2 -P ubi10/httpd-24
```

3. Run the manually published container and publish container port 8080:

```
# podman run -dt --name=web3 -p 8888:8080 ubi10/httpd-24
```

4. List all containers:

```
# podman ps

CONTAINER ID  IMAGE                                    COMMAND               CREATED
STATUS         PORTS                                    NAMES
db23e8dabc74  registry.access.redhat.com/ubi10/httpd-24:latest  /usr/bin/run-http...  23
seconds ago  Up 23 seconds  8080/tcp, 8443/tcp                      web1
1824b8f0a64b  registry.access.redhat.com/ubi10/httpd-24:latest  /usr/bin/run-http...  18
seconds ago  Up 18 seconds  0.0.0.0:33127->8080/tcp, 0.0.0.0:37679->8443/tcp  web2
39de784d917a  registry.access.redhat.com/ubi10/httpd-24:latest  /usr/bin/run-http...  5
seconds ago  Up 5 seconds  0.0.0.0:8888->8080/tcp, 8443/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/10/httpd-24** image has the **EXPOSE 8080** and **EXPOSE 8443** commands in the Containerfile.

- Container **web3** has a manually published port. The host port 8888 is mapped to the container port 8080.

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 10.88.0.2:8080
...
<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 10.88.0.4:8080
...
<title>Test Page for the HTTP Server on Red Hat Enterprise Linux</title>
...
```

10.8. COMMUNICATING BETWEEN CONTAINERS USING DNS

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

Prerequisites

- The **container-tools** meta-package is installed.
- A network with the enabled DNS plugin has been created by using the **podman network create** command.

Procedure

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

```
# podman run -d --net mynet --name receiver ubi10 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.

10.9. COMMUNICATING BETWEEN TWO CONTAINERS IN A POD

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

Prerequisites

- The **container-tools** meta-package is installed.

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
```

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

10.10. COMMUNICATING IN A POD

Expose services running inside a pod by publishing ports at the pod level. Containers within the pod share these exposed ports.

Prerequisites

- The **container-tools** meta-package is installed.

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
```

10.11. ATTACHING A POD TO THE CONTAINER NETWORK

Connect a pod to a specific network when creating it. All containers subsequently added to the pod automatically join that network.

Prerequisites

- The **container-tools** meta-package is installed.

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/ubi10/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
```

10.12. SETTING HTTP PROXY VARIABLES FOR PODMAN

To pull images behind a proxy server, you must set HTTP proxy variables for Podman. Podman reads the environment variable **HTTP_PROXY** to ascertain the HTTP Proxy information. HTTP proxy information can be configured as an environment variable or under **/etc/profile.d**.

Procedure

- Set proxy variables for Podman. For example:
  - Unauthenticated proxy:

    ```
    # cat /etc/profile.d/unauthenticated_http_proxy.sh
    export HTTP_PROXY=http://192.168.0.1:3128
    export HTTPS_PROXY=http://192.168.0.1:3128
    export NO_PROXY=example.com,172.5.0.0/16
    ```

  - Authenticated proxy:

    ```
    # cat /etc/profile.d/authenticated_http_proxy.sh
    export HTTP_PROXY=http://USERNAME:PASSWORD@192.168.0.1:3128
    export HTTPS_PROXY=http://USERNAME:PASSWORD@192.168.0.1:3128
    export NO_PROXY=example.com,172.5.0.0/16
    ```

CHAPTER 11. SETTING CONTAINER NETWORK MODES

You can configure container network modes by using Podman to define how your workloads communicate with the host and external networks. Selecting the appropriate networking environment ensures optimal performance and security isolation tailored to your specific application requirements.

11.1. RUNNING CONTAINERS WITH A STATIC IP

Assign a specific IP address to a container by using the **--ip** option. This helps ensure consistent network addressing for services that require fixed IPs.

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.

Prerequisites

- The **container-tools** meta-package is installed.

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/ubi10/ubi
efde5f0a8c723f70dd5cb5dc3d5039df3b962fae65575b08662e0d5b5f9fbe85
```

Verification

- Check that the IP address is set properly:

```
# podman inspect --format='{{.NetworkSettings.IPAddress}}' myubi
10.88.0.44
```

11.2. RUNNING THE DHCP PLUGIN FOR NETAVARK USING SYSTEMD

You can run the DHCP plugin for Netavark as a systemd service to enable dynamic IP addressing for your containers. By operating this plugin as a persistent service ensures that containerized workloads maintain consistent network connectivity automatically.

Prerequisites

- The **container-tools** meta-package is installed.

Procedure

1. Enable the DHCP proxy by using the systemd socket:

```
systemctl enable --now netavark-dhcp-proxy.socket
Created symlink /etc/systemd/system/sockets.target.wants/netavark-dhcp-proxy.socket →
/usr/lib/systemd/system/netavark-dhcp-proxy.socket.
```

2. Optional: Display the socket unit file:

```
# cat /usr/lib/systemd/system/netavark-dhcp-proxy.socket
[Unit]
Description=Netavark DHCP proxy socket

[Socket]
ListenStream=%t/podman/nv-proxy.sock
SocketMode=0660

[Install]
WantedBy=sockets.target
```

3. Create a macvlan network and specify your host interface with it. Typically, it is your external interface:

```
# podman network create -d macvlan --interface-name <LAN_INTERFACE> mv1
mv1
```

4. Run the container by using newly created network:

```
# podman run --rm --network mv1 -d --name test alpine top
894ae3b6b1081aca2a5d90a9855568eaa533c08a174874be59569d4656f9bc45
```

Verification

1. Confirm the container has an IP on your local subnet:

```
# podman exec test ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0@eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
UP qlen 1000
    link/ether 5a:30:72:bf:13:76 brd ff:ff:ff:ff:ff:ff
    inet 192.168.188.36/24 brd 192.168.188.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::5830:72ff:febf:1376/64 scope link
       valid_lft forever preferred_lft forever
```

2. Inspect the container to verify it uses correct IP addresses:

```
# podman container inspect test --format
{{.NetworkSettings.Networks.mv1.IPAddress}}
192.168.188.36
```

NOTE

When attempting to connect to this IP address, ensure the connection is made from a different host. Connections from the same host are not supported when using macvlan networking.

Additional resources

- Lease dynamic IPs with Netavark

11.3. THE MACVLAN PLUGIN

You can connect containers directly to a physical network interface by using the **macvlan** plugin. This allows containers to appear as physical devices on the network.

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. With the **macvlan** plugin, you can connect a container to the same network as the host. This only applies to rootfull containers. Rootless containers are not able to use the **macvlan** and **dhcp** plugins.

You can create a MacVLAN network by using the **podman network create --driver=macvlan** command.

CHAPTER 12. PORTING CONTAINERS TO SYSTEMD BY USING PODMAN

Manage containers as system services by using systemd integration. With this feature, you can auto-start containers, manage dependencies, and control their state by using standard **systemctl** commands.

Podman is a daemonless container engine with a Docker-CLI comparable command line, simplifying the transition from other container engines and managing pods, containers, and images.

Initially, Podman was not designed for full Linux system or service management. However, due to Red Hat's integration of containers with systemd, you can now manage OCI and Docker-formatted containers built by Podman like any other Linux service by using the systemd initialization service.

You can use systemd unit files to start containers or pods as systemd services, define their execution order and dependencies, and control their state by using the **systemctl** command.

12.1. AUTO-GENERATING A SYSTEMD UNIT FILE USING QUADLETS

Generate systemd service files automatically by using Quadlet. By creating a simple unit file describing the container, Quadlet handles the complexity of integrating with systemd.

With Quadlet, you describe how to run a container in a format that is very similar to regular systemd unit files. The container descriptions focus on the relevant container details and hide technical details of running containers under systemd. Create the ***<CTRNAME>*.container** unit file in one of the following directories:

- For root users: **/usr/share/containers/systemd/** or **/etc/containers/systemd/**
- For rootless users: **$HOME/.config/containers/systemd/**, **$XDG_CONFIG_HOME/containers/systemd/, /etc/containers/systemd/users/$(UID)**, or **/etc/containers/systemd/users/**

NOTE

Quadlet is available beginning with Podman v4.6.

Prerequisites

- The **container-tools** meta-package is installed.

Procedure

1. Create the **mysleep.container** unit file:

```
$ cat $HOME/.config/containers/systemd/mysleep.container
[Unit]
Description=The sleep container
After=local-fs.target

[Container]
Image=registry.access.redhat.com/ubi10-minimal:latest
Exec=sleep 1000

[Install]
```

```
# Start by default on boot
WantedBy=multi-user.target default.target
```

In the **[Container]** section you must specify:

- **Image** - container mage you want to tun
- **Exec** - the command you want to run inside the container
  This enables you to use all other fields specified in a **systemd** unit file.

2. Create the **mysleep.service** based on the **mysleep.container** file:

   ```
   $ systemctl --user daemon-reload
   ```

3. Optional: Check the status of the **mysleep.service**:

   ```
   $ systemctl --user status mysleep.service
   ○ mysleep.service - The sleep container
     Loaded: loaded (/home/username/.config/containers/systemd/mysleep.container;
   generated)
     Active: inactive (dead)
   ```

4. Start the **mysleep.service**:

   ```
   $ systemctl --user start mysleep.service
   ```

Verification

1. Check the status of the **mysleep.service**:

   ```
   $ systemctl --user status mysleep.service
   ● mysleep.service - The sleep container
     Loaded: loaded (/home/username/.config/containers/systemd/mysleep.container;
   generated)
     Active: active (running) since Thu 2023-02-09 18:07:23 EST; 2s ago
    Main PID: 265651 (conmon)
       Tasks: 3 (limit: 76815)
     Memory: 1.6M
        CPU: 94ms
     CGroup: ...
   ```

2. List all containers:

   ```
   $ podman ps -a
   CONTAINER ID  IMAGE                            COMMAND         CREATED         STATUS
   PORTS   NAMES
   421c8293fc1b  registry.access.redhat.com/ubi10-minimal:latest          sleep 1000  30
   seconds ago   Up 10 seconds ago systemd-mysleep
   ```

   Note that the name of the created container consists of the following elements:

   - a **systemd-** prefix

- a name of the **systemd** unit, that is **systemd-mysleep**
  This naming helps to distinguish common containers from containers running in **systemd** units. It also helps to determine which unit a container runs in. If you want to change the name of the container, use the **ContainerName** field in the **[Container]** section.

  For more information, see the **podman-systemd.unit(5)** man page on your system.

12.2. ENABLING SYSTEMD SERVICES

Enable your containerized services to start automatically at boot. For user services, you must enable lingering to help ensure they persist after logout.

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>
    ```

    Refer to **systemctl(1)** and **loginctl(1)** man pages on your system for more information.

Additional resources

- Managing system service with systemctl

12.3. AUTO-STARTING CONTAINERS BY USING SYSTEMD

You can control the state of the systemd system and service manager by using the **systemctl** command. You can enable, start, stop the service as a non-root user. To install the service as a root user, omit the **--user** option.

Prerequisites

- The **container-tools** meta-package is installed.

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 ubi10-
   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 by using the **systemctl is-enabled container.service** command.

5. Optional: To stop **container.service**, enter:

   ```
   # systemctl --user stop container.service
   ```

Verification

- List containers that are running or have exited:

  ```
  # podman ps
  CONTAINER ID  IMAGE                            COMMAND  CREATED        STATUS
  PORTS  NAMES
  f20988d59920  registry.access.redhat.com/ubi10-minimal:latest  top      12 seconds ago  Up
  11 seconds ago         funny_zhukovsky
  ```

Additional resources

- Managing system service with systemctl

12.4. ADVANTAGES OF USING QUADLETS OVER THE PODMAN GENERATE SYSTEMD COMMAND

You can use the Quadlets tool, which describes how to run a container in a format similar to regular **systemd** unit files.

> **NOTE**
>
> Quadlet is available starting from Podman v4.6.

Quadlets have many advantages over generating unit files by using the **podman generate systemd** command, such as:

- **Easy to maintain**: The container descriptions focus on the relevant container details and hide technical details of running containers under **systemd**.
- **Automatically updated**: Quadlets do not require manually regenerating unit files after an update. If a newer version of Podman is released, your service is automatically updated when the **systemclt daemon-reload** command is executed, for example, at boot time.
- **Simplified workflow**: Thanks to the simplified syntax, you can create Quadlet files from scratch and deploy them anywhere.
- **Support standard systemd options**: Quadlet extends the existing systemd-unit syntax with new tables, for example, a table to configure a container.

> **NOTE**
>
> Quadlet supports a subset of Kubernetes YAML capabilities. For more information, see the support matrix of supported YAML fields. You can generate the YAML files by using one of the following tools:
>
> - Podman: **podman generate kube** command
> - OpenShift: **oc generate** command with the **--dry-run** option
> - Kubernetes: **kubectl create** command with the **--dry-run** option

Quadlet supports these unit file types:

- **Container units**: Used to manage containers by running the **podman run** command.
  - File extension: **.container**
  - Section name: **[Container]**
  - Required fields: **Image** describing the container image the service runs
- **Kube units**: Used to manage containers defined in Kubernetes YAML files by running the **podman kube play** command.

- File extension: **.kube**
- Section name: **[Kube]**
- Required fields: **Yaml** defining the path to the Kubernetes YAML file

- **Network units**: Used to create Podman networks that may be referenced in **.container** or **.kube** files.
  - File extension: **.network**
  - Section name: **[Network]**
  - Required fields: None
- **Volume units**: Used to create Podman volumes that may be referenced in **.container** files.
  - File extension: **.volume**
  - Section name: **[Volume]**
  - Required fields: None

For more information, see the **podman-systemd.unit(5)** man page on your system.

Additional resources

- Podman Quadlets with Podman Desktop

12.5. GENERATING A SYSTEMD UNIT FILE USING PODMAN

Create systemd unit files for existing containers by using the **podman generate systemd** command. This allows Podman to manage running containers as services, though Quadlets are preferred for new configurations.

The **podman generate systemd** command also helps ensure that you get the latest version of unit files.

NOTE

Starting with Podman v4.6, you can use the Quadlets that describe how to run a container in a format similar to regular **systemd** unit files and hides the complexity of running containers under **systemd**.

Prerequisites

- The **container-tools** meta-package is installed.

Procedure

1. Create a container (for example **myubi**):

```
$ podman create --name myubi registry.access.redhat.com/ubi10: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

- 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.
    For more information, see **podman-generate-systemd(1)** man page on your system.

12.6. AUTOMATICALLY GENERATING A SYSTEMD UNIT FILE BY USING PODMAN

By default, Podman generates a unit file for existing containers or pods. You can generate more portable **systemd** unit files by using the **podman generate systemd --new**. The **--new** flag instructs Podman to generate unit files that create, start, and remove containers.

NOTE

Starting with Podman v4.6, you can use the Quadlets that describe how to run a container in a format similar to regular **systemd** unit files and hides the complexity of running containers under systemd.

Prerequisites

- The **container-tools** meta-package is installed.

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/ubi10/httpd-24
```

2. Optional: List all images available on your system:

```
# podman images
REPOSITORY                                  TAG         IMAGE ID      CREATED       SIZE
registry.access.redhat.com/ubi10/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/ubi10/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/ubi10/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/ubi10/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 by 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 **/etc/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

- 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

- Managing system service with systemctl

12.7. AUTOMATICALLY STARTING PODS USING SYSTEMD

Manage entire pods as systemd services to help ensure all containers within the pod start and stop together. This simplifies the management of multi-container applications.

Note that the **systemctl** command should only be used on a pod and you should not start or stop containers individually through **systemctl**, as they are managed by the pod service along with the internal infra-container.

NOTE

Starting with Podman v4.6, you can use the Quadlets that describe how to run a container in a format similar to regular **systemd** unit files and hides the complexity of running containers under **systemd**.

Prerequisites

- The **container-tools** meta-package is installed.

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/ubi*10
top
$ *podman create --pod systemd-pod --name container1 registry.access.redhat.com/ubi*10
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/ubi10: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/ubi10: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

- Check if the service is enabled:

```
$ systemctl is-enabled pod-systemd-pod.service
enabled
```

For more information, see the **podman-create(1)**, **podman-generate-systemd(1)**, and **systemctl(1)** man pages on your system.

Additional resources

- Managing system service with systemctl

12.8. AUTOMATICALLY UPDATING CONTAINERS USING PODMAN

With the **podman auto-update** command, you can 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.

NOTE

Starting with Podman v4.6, you can use the Quadlets that describe how to run a container in a format similar to regular systemd unit files and hides the complexity of running containers under systemd.

Prerequisites

- The **container-tools** meta-package is installed.

Procedure

1. Start a **myubi** container based on the **registry.access.redhat.com/ubi10/ubi-init** image:

    ```
    # podman run --label "io.containers.autoupdate=image" \
    --name myubi -dt registry.access.redhat.com/ubi10/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/10/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
    ```

    For more information, see the **podman-generate-systemd(1)** and **systemctl(1)** man pages on your system.

Additional resources

- Managing system service with systemctl

12.9. AUTOMATICALLY UPDATING CONTAINERS BY USING SYSTEMD

Schedule automatic container updates by using the **podman-auto-update** timer and service. This helps ensure your system regularly checks for and applies image updates without manual intervention.

As mentioned in section Automatically updating containers by using Podman , you can update the container by 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.

NOTE

Starting with Podman v4.6, you can use the Quadlets that describe how to run a container in a format similar to regular **systemd** unit files and hides the complexity of running containers under systemd.

Prerequisites

- The **container-tools** meta-package is installed.

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

- Managing system services with systemd

CHAPTER 13. PORTING CONTAINERS TO OPENSHIFT BY USING PODMAN

Generate portable YAML descriptions of your local containers for deployment to OpenShift or Kubernetes. With this feature, you can develop locally with Podman and deploy globally.

You can use YAML files to:

- Re-run a local orchestrated set of containers and pods with minimal input required which can be useful for iterative development.
- Run same containers and pods on another machine.

The **podman kube play** command supports a subset of Kubernetes YAML capabilities. For more information, see the **podman-kube-play(1)** man page on your system.

13.1. GENERATING A KUBERNETES YAML FILE USING PODMAN

You can create a pod with one container and generate the Kubernetes YAML file by using the **podman generate kube** command.

Prerequisites

- The **container-tools** meta-package is installed.
- 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/ubi10/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/ubi10/ubi:latest
        name: myubi
        resources: {}
        securityContext:
              allowPrivilegeEscalation: true
              capabilities: {}
              privileged: false
              readOnlyRootFilesystem: false
        tty: true
        workingDir: /
status: {}
```

13.2. 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.

NOTE

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

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.

13.3. STARTING CONTAINERS AND PODS WITH PODMAN

With the generated YAML files, you can automatically start containers and pods in any environment. The YAML files can be generated by using tools other than Podman, such as Kubernetes or OpenShift.

With the **podman play kube** command, you can re-create pods and containers based on the YAML input file.

Prerequisites

- The **container-tools** meta-package is installed.

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/ubi10/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.

For more information, see the **podman-play-kube(1)** man page on your system.

13.4. 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.

13.5. MANUALLY RUNNING CONTAINERS AND PODS BY USING PODMAN

Manually deploy a multi-container application, such as WordPress with MariaDB, by using Podman. This involves creating a pod and adding connected containers to it.

Suppose the following directory layout:

```
├─── mariadb-conf
|    ├─── Containerfile
|    ├─── my.cnf
```

Prerequisites

- The **container-tools** meta-package is installed.

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 by 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 by using the **curl** command:

```
$ curl 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>
...
```

For more information, see the **podman-play-kube(1)** man page on your system.

13.6. GENERATING A YAML FILE BY USING PODMAN

You can generate a Kubernetes YAML file by using the **podman generate kube** command.

Prerequisites

- The **container-tools** meta-package is installed.
- 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
```

For more information, see the **podman-play-kube(1)** man page on your system.

13.7. 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 by 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 **container-tools** meta-package is installed.
- 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.

Procedure

1. 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
```

2. Create the wordpress pod by using the **wordpress.yaml** file:

```
$ podman play kube wordpress.yaml
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**.

3. 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 by using the **curl** command:

```
$ curl 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>
...
```

For more information, see the **podman-play-kube(1)** man page on your system.

13.8. AUTOMATICALLY STOPPING AND REMOVING PODS BY 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 **container-tools** meta-package is installed.
- The pod named **wordpresspod** has been created. For details, see section Manually running containers and pods by using Podman.
- The YAML file has been generated. For details, see section Generating a YAML file by using Podman.
- The pod is running. For details, see section Automatically running containers and pods by 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
```

Refer to **podman-play-kube(1)** man page on your system for more information.

CHAPTER 14. MANAGING CONTAINERS BY USING RHEL SYSTEM ROLES

The **podman** RHEL system roles manage containers on Red Hat Enterprise Linux systems. This role uses Ansible to configure **Podman**, manage container lifecycles, and even deploy containerized applications as systemd services.

14.1. CONFIGURING IMAGE REGISTRY MANAGEMENT FOR PODMAN AND OTHER CONTAINER TOOLS

With the **podman** RHEL system role, you can automate the Podman management, including registry configuration, across multiple RHEL systems. Instead of manually editing files, you define your desired registry configuration in an Ansible playbook.

The **podman** RHEL system role uses the **podman_registries_conf** variable, which accepts a dictionary containing the registry settings. The role then creates a drop-in file, for example, in the **/etc/containers/registries.conf.d/** to apply your configuration, following best practices for managing system configurations.

Prerequisites

- You have prepared the control node and the managed nodes .
- You are logged in to the control node as a user who can run playbooks on the managed nodes.
- The account you use to connect to the managed nodes has **sudo** permissions for these nodes.

Procedure

1. Create a playbook file, for example, **~/playbook.yml**, with the following content:

```
---
- name: Configure Podman registries with RHEL system roles
  hosts: managed-node-01.example.com
  vars:
    podman_registries_conf:
      unqualified-search-registries:
        - "registry.access.redhat.com"
        - "docker.io"
        - "my-company-registry.com"
      registry:
        - location: "my-company-registry.com"
        - location: "my-local-registry:5000"
          insecure: true
  tasks:
    - name: Include the podman system role
      ansible.builtin.include_role:
        name: redhat.rhel_system_roles.podman
```

The settings specified in the example playbook include the following:

- **unqualified-search-registries**: Extends the list of registries Podman searches when you use a short image name (for example, **podman pull <my-image>**). Podman searches for images in my-company-registry.com after the default registries.
- **[registry]**: Defines specific properties for a given registry. For example, you can enable an insecure connection by setting **insecure=true** to a local registry running at my-local-registry:5000.

The **podman_use_new_toml_formatter** variable generates TOML-compliant configuration files that are compatible with Podman. This variable enhances the Podman role by supporting all TOML features, including tables and inline tables, through a true TOML formatter instead of the Jinja template used previously.

The new formatter is disabled by default to maintain compatibility with the previous formatter's behavior. To enable the new formatter, set **podman_use_new_toml_formatter: true** in your configuration:

```
podman_use_new_toml_formatter: true
podman_containers_conf:
  containers:
    annotations:
      - environment=production
      - status=tier2
```

2. Validate the playbook syntax:

```
$ ansible-playbook --syntax-check ~/playbook.yml
```

Note that this command only validates the syntax and does not protect against a wrong but valid configuration.

3. Run the playbook:

```
$ ansible-playbook ~/playbook.yml
```

Verification

1. Run the **podman info** command on the host:

```
$ ansible managed-node-01.example.com -m command -a 'podman info'
```

2. Verify the registeries section:

```
registries:
  my-company-registry.com:
    Blocked: false
    Insecure: false
    Location: my-company-registry.com
    MirrorByDigestOnly: false
    Mirrors: null
    Prefix: my-company-registry.com
    PullFromMirror: ""
  my-local-registry:5000:
    Blocked: false
    Insecure: true
    Location: my-local-registry:5000
    MirrorByDigestOnly: false
    Mirrors: null
    Prefix: my-local-registry:5000
    PullFromMirror: ""
  search:
  - registry.access.redhat.com
  - docker.io
  - my-company-registry.com
```

14.2. CREATING A ROOTLESS CONTAINER WITH BIND MOUNT BY USING THE PODMAN RHEL SYSTEM ROLE

You can use the **podman** RHEL system role to create rootless containers with bind mount by running an Ansible playbook and with that, manage your application configuration.

The example Ansible playbook starts two Kubernetes pods: one for a database and another for a web application. The database pod configuration is specified in the playbook, while the web application pod is defined in an external YAML file.

Prerequisites

- You have prepared the control node and the managed nodes .
- You are logged in to the control node as a user who can run playbooks on the managed nodes.
- The account you use to connect to the managed nodes has **sudo** permissions for these nodes.
- The user and group **webapp** exist, and must be listed in the **/etc/subuid** and **/etc/subgid** files on the host.
- The user named **dbuser** and a group named **dbgroup** must be already created.

Procedure

1. Create a playbook file, for example, **~/playbook.yml**, with the following content:

```
- name: Configure Podman
  hosts: managed-node-01.example.com
  tasks:
    - name: Create a web application and a database
      ansible.builtin.include_role:
        name: redhat.rhel_system_roles.podman
      vars:
        podman_create_host_directories: true
        podman_firewall:
          - port: 8080-8081/tcp
            state: enabled
          - port: 12340/tcp
            state: enabled
        podman_selinux_ports:
          - ports: 8080-8081
            setype: http_port_t
    podman_kube_specs:
      - state: started
        run_as_user: dbuser
        run_as_group: dbgroup
        kube_file_content:
          apiVersion: v1
          kind: Pod
          metadata:
            name: db
          spec:
            containers:
              - name: db
                image:  quay.io/rhel-system-roles/mysql:5.6
                ports:
                  - containerPort: 1234
                    hostPort: 12340
                volumeMounts:
                  - mountPath: /var/lib/db:Z
                    name: db
            volumes:
              - name: db
                hostPath:
                  path: /var/lib/db
      - state: started
        run_as_user: webapp
        run_as_group: webapp
        kube_file_src: /path/to/webapp.yml
```

The settings specified in the example playbook include the following:

run_as_user and **run_as_group**

Specify that containers are rootless.

kube_file_content

Contains a Kubernetes YAML file defining the first container named **db**. You can generate the Kubernetes YAML file by using the **podman kube generate** command.

- The **db** container is based on the **quay.io/db/db:stable** container image.
- The **db** bind mount maps the **/var/lib/db** directory on the host to the **/var/lib/db** directory in the container. The **Z** flag labels the content with a private unshared label, therefore, only the **db** container can access the content.

kube_file_src: *<path>*

Defines the second container. The content of the **/path/to/webapp.yml** file on the controller node will be copied to the **kube_file** field on the managed node.

volumes: *<list>*

A YAML list to define the source of the data to provide in one or more containers. For example, a local disk on the host (**hostPath**) or other disk device.

volumeMounts: *<list>*

A YAML list to define the destination where the individual container will mount a given volume.

podman_create_host_directories: true

Creates the directory on the host. This instructs the role to check the kube specification for **hostPath** volumes and create those directories on the host. If you need more control over the ownership and permissions, use **podman_host_directories**.

For details about all variables used in the playbook, see the **/usr/share/ansible/roles/rhel-system-roles.podman/README.md** file on the control node.

2. Validate the playbook syntax:

```
$ ansible-playbook --syntax-check --ask-vault-pass ~/playbook.yml
```

Note that this command only validates the syntax and does not protect against a wrong but valid configuration.

3. Run the playbook:

```
$ ansible-playbook --ask-vault-pass ~/playbook.yml
```

14.3. CREATING A ROOTFUL CONTAINER WITH PODMAN VOLUME BY USING THE PODMAN RHEL SYSTEM ROLE

You can use the **podman** RHEL system role to create a rootful container with a Podman volume by running an Ansible playbook and with that, manage your application configuration.

The example Ansible playbook deploys a Kubernetes pod named **ubi10-httpd** running an HTTP server container from the **registry.access.redhat.com/ubi10/httpd-24** image. The container's web content is mounted from a persistent volume named **ubi10-html-volume**. By default, the **podman** role creates rootful containers.

Prerequisites

- You have prepared the control node and the managed nodes .
- You are logged in to the control node as a user who can run playbooks on the managed nodes.
- The account you use to connect to the managed nodes has **sudo** permissions for these nodes.

Procedure

1. Create a playbook file, for example, **~/playbook.yml**, with the following content:

```
- name: Configure Podman
  hosts: managed-node-01.example.com
  tasks:
    - name: Start Apache server on port 8080
      ansible.builtin.include_role:
        name: redhat.rhel_system_roles.podman
  vars:
    podman_firewall:
      - port: 8080/tcp
        state: enabled
    podman_kube_specs:
      - state: started
        kube_file_content:
```

```
        apiVersion: v1
        kind: Pod
        metadata:
          name: ubi10-httpd
        spec:
          containers:
            - name: ubi10-httpd
              image: registry.access.redhat.com/ubi10/httpd-24
              ports:
                - containerPort: 8080
                  hostPort: 8080
              volumeMounts:
                - mountPath: /var/www/html:Z
                  name: ubi10-html
          volumes:
            - name: ubi10-html
              persistentVolumeClaim:
                claimName: ubi10-html-volume
```

The settings specified in the example playbook include the following:

kube_file_content

Contains a Kubernetes YAML file defining the first container named **db**. You can generate the Kubernetes YAML file by using the **podman kube generate** command.

- The **ubi10-httpd** container is based on the **registry.access.redhat.com/ubi10/httpd-24** container image.
- The **ubi10-html-volume** maps the **/var/www/html** directory on the host to the container. The **Z** flag labels the content with a private unshared label, therefore, only the **ubi10-httpd** container can access the content.
- The pod mounts the existing persistent volume named **ubi10-html-volume** with the mount path **/var/www/html**.

For details about all variables used in the playbook, see the **/usr/share/ansible/roles/rhel-system-roles.podman/README.md** file on the control node.

2. Validate the playbook syntax:

```
$ ansible-playbook --syntax-check ~/playbook.yml
```

Note that this command only validates the syntax and does not protect against a wrong but valid configuration.

3. Run the playbook:

```
$ ansible-playbook ~/playbook.yml
```

14.4. CREATING A QUADLET APPLICATION WITH SECRETS BY USING THE PODMAN RHEL SYSTEM ROLE

You can use the **podman** RHEL system role to create a Quadlet application with secrets by running an Ansible playbook.

Prerequisites

- You have prepared the control node and the managed nodes .
- You are logged in to the control node as a user who can run playbooks on the managed nodes.
- The account you use to connect to the managed nodes has **sudo** permissions for these nodes.
- The certificate and the corresponding private key that the web server in the container should use are stored in the **~/certificate.pem** and **~/key.pem** files.

Procedure

1. Display the contents of the certificate and private key files:

   ```
   $ cat ~/certificate.pem
   -----BEGIN CERTIFICATE-----
   ...
   -----END CERTIFICATE-----

   $ cat ~/key.pem
   -----BEGIN PRIVATE KEY-----
   ...
   -----END PRIVATE KEY-----
   ```

   You require this information in a later step.

2. Store your sensitive variables in an encrypted file:

   a. Create the vault:

      ```
      $ ansible-vault create ~/vault.yml
      New Vault password: <vault_password>
      Confirm New Vault password: <vault_password>
      ```

   b. After the **ansible-vault create** command opens an editor, enter the sensitive data in the ***<key>*: *<value>*** format:

      ```
      root_password: <root_password>
      certificate: |-
        -----BEGIN CERTIFICATE-----
        ...
        -----END CERTIFICATE-----
      key: |-
        -----BEGIN PRIVATE KEY-----
        ...
        -----END PRIVATE KEY-----
      ```

      Ensure that all lines in the **certificate** and **key** variables start with two spaces.

   c. Save the changes, and close the editor. Ansible encrypts the data in the vault.

3. Create a playbook file, for example, **~/playbook.yml**, with the following content:

```
- name: Deploy a wordpress CMS with MySQL database
  hosts: managed-node-01.example.com
  vars_files:
    - ~/vault.yml
  tasks:
  - name: Create and run the container
    ansible.builtin.include_role:
      name: redhat.rhel_system_roles.podman
    vars:
      podman_create_host_directories: true
      podman_activate_systemd_unit: false
      podman_quadlet_specs:
        - name: quadlet-demo
          type: network
          file_content: |
            [Network]
            Subnet=192.168.30.0/24
            Gateway=192.168.30.1
            Label=app=wordpress
        - file_src: quadlet-demo-mysql.volume
        - template_src: quadlet-demo-mysql.container.j2
        - file_src: envoy-proxy-configmap.yml
        - file_src: quadlet-demo.yml
        - file_src: quadlet-demo.kube
          activate_systemd_unit: true
      podman_firewall:
        - port: 8000/tcp
          state: enabled
        - port: 9000/tcp
          state: enabled
      podman_secrets:
        - name: mysql-root-password-container
          state: present
          skip_existing: true
          data: "{{ root_password }}"
        - name: mysql-root-password-kube
          state: present
          skip_existing: true
          data: |
            apiVersion: v1
            data:
              password: "{{ root_password | b64encode }}"
            kind: Secret
            metadata:
              name: mysql-root-password-kube
        - name: envoy-certificates
          state: present
          skip_existing: true
          data: |
            apiVersion: v1
            data:
              certificate.key: {{ key | b64encode }}
              certificate.pem: {{ certificate | b64encode }}
        kind: Secret
        metadata:
          name: envoy-certificates
```

The procedure creates a WordPress content management system paired with a MySQL database. The **podman_quadlet_specs role** variable defines a set of configurations for the Quadlet, which refers to a group of containers or services that work together in a certain way. It includes the following specifications:

- The Wordpress network is defined by the **quadlet-demo** network unit.
- The volume configuration for MySQL container is defined by the **file_src: quadlet-demo-mysql.volume** field.
- The **template_src: quadlet-demo-mysql.container.j2** field is used to generate a configuration for the MySQL container.
- Two YAML files follow: **file_src: envoy-proxy-configmap.yml** and **file_src: quadlet-demo.yml**. Note that .yml is not a valid Quadlet unit type, therefore these files will just be copied and not processed as a Quadlet specification.
- The Wordpress and envoy proxy containers and configuration are defined by the **file_src: quadlet-demo.kube** field. The kube unit refers to the previous YAML files in the **[Kube]** section as **Yaml=quadlet-demo.yml** and **ConfigMap=envoy-proxy-configmap.yml**.
  For details about all variables used in the playbook, see the **/usr/share/ansible/roles/rhel-system-roles.podman/README.md** file on the control node.

4. Validate the playbook syntax:

   ```
   $ ansible-playbook --syntax-check --ask-vault-pass ~/playbook.yml
   ```

   Note that this command only validates the syntax and does not protect against a wrong but valid configuration.

5. Run the playbook:

   ```
   $ ansible-playbook --ask-vault-pass ~/playbook.yml
   ```

Additional resources

- Ansible vault

CHAPTER 15. MANAGING CONTAINER IMAGES BY USING THE RHEL WEB CONSOLE

You can use the RHEL web console to manage container images. The web console provides a graphical interface and allows you to pull, prune, or delete your container images through the **Podman containers** section.

15.1. PULLING CONTAINER IMAGES IN THE WEB CONSOLE

You can download container images to your local system and use them to create your containers.

Prerequisites

- You have installed the RHEL 10 web console.
  For instructions, see Installing and enabling the web console .
- The **cockpit-podman** add-on is installed.

Procedure

1. Log in to the RHEL 10 web console.
2. Click **Podman containers** in the main menu.
3. In the **Images** table, click the overflow menu in the upper-right corner and select **Download new image**.
4. The **Search for an image**dialog box appears.
5. In the **Search for** field, enter the name of the image or specify its description.
6. In the **in** drop-down list, select the registry from which you want to pull the image.
7. Optional: In the **Tag** field, enter the tag of the image.
8. Click **Download**.

Verification

- Click **Podman containers** in the main menu. You can see the newly downloaded image in the **Images** table.

NOTE

You can create a container from the downloaded image by clicking the **Create container** button in the **Images** table. To create the container, follow steps 3-8 in Creating containers in the web console.

15.2. PRUNING CONTAINER IMAGES IN THE WEB CONSOLE

You can remove all unused images that do not have any containers based on it.

Prerequisites

- At least one container image is pulled.
- You have installed the RHEL 10 web console. For instructions, see Installing and enabling the web console.
- The **cockpit-podman** add-on is installed:

  ```
  # dnf install cockpit-podman
  ```

Procedure

1. Log in to the RHEL 10 web console.
2. Click **Podman containers** in the main menu.
3. In the **Images** table, click the overflow menu in the upper-right corner and select **Prune unused images**.
4. The window with the list of images appears. Click **Prune** to confirm your choice.

Verification

- Click **Podman containers** in the main menu. The deleted images should not be listed in the **Images** table.

15.3. DELETING CONTAINER IMAGES IN THE WEB CONSOLE

You can delete a previously pulled container image by using the web console.

Prerequisites

- At least one container image is pulled.
- You have installed the RHEL 10 web console.
- The **cockpit-podman** add-on is installed.

Procedure

1. Log in to the RHEL 10 web console.
2. Click the **Podman containers** button in the main menu.
3. In the **Images** table, select the image you want to delete and click the overflow menu and select **Delete**.
4. In the following dialog, click **Delete tagged images** to confirm your choice.

Verification

- Click the **Podman containers** in the main menu. The deleted container should not be listed in the **Images** table.

CHAPTER 16. MANAGING CONTAINERS BY USING THE RHEL WEB CONSOLE

You can use the RHEL web console to manage your containers and pods. With the web console, you can create containers as a non-root or root user.

- As a *root* user, you can create system containers with extra privileges and options.
- As a *non-root* user, you have two options:
  - To only create user containers, you can use the web console in its default mode - **Limited access**.
  - To create both user and system containers, click **Administrative access** in the top panel of the web console page.

For details about differences between root and rootless containers, see Special considerations for rootless containers.

16.1. CREATING A CONTAINER CHECKPOINT IN THE WEB CONSOLE

With the web console, you can set a checkpoint on a running container or an individual application and store its state to disk.

NOTE

Creating a checkpoint is available only for system containers.

Prerequisites

- The container is running.
- You have installed the RHEL 10 web console.
  For instructions, see Installing and enabling the web console .
- The **cockpit-podman** add-on is installed.

Procedure

1. Log in to the RHEL 10 web console.
2. Click **Podman containers** in the main menu.
3. In the **Containers** table, select the container you want to modify and click the overflow icon menu and select **Checkpoint**.
4. Optional: In the **Checkpoint container** form, check the options you need:
   - Keep all temporary checkpoint files: keep all temporary log and statistics files created by CRIU during checkpointing. These files are not deleted if checkpointing fails for further debugging.
   - Leave running after writing checkpoint to disk: leave the container running after checkpointing instead of stopping it.

- Support preserving established TCP connections

5. Click **Checkpoint**.

Verification

- Click the **Podman containers** in the main menu. Select the container you checkpointed, click the overflow menu icon and verify that there is a **Restore** option.

16.2. RESTORING A CONTAINER CHECKPOINT IN THE WEB CONSOLE

You can use data saved to restore the container after a reboot at the same point in time it was checkpointed.

NOTE

Creating a checkpoint is available only for system containers.

Prerequisites

- The container was checkpointed.
- You have installed the RHEL 10 web console. For instructions, see Installing and enabling the web console.
- The **cockpit-podman** add-on is installed.

Procedure

1. Log in to the RHEL 10 web console.
2. Click **Podman containers** in the main menu.
3. In the **Containers** table, select the container you want to modify and click the overflow menu and select **Restore**.
4. Optional: In the **Restore container** form, check the options you need:
    - **Keep all temporary checkpoint files**: Keep all temporary log and statistics files created by CRIU during checkpointing. These files are not deleted if checkpointing fails for further debugging.
    - **Restore with established TCP connections**
    - **Ignore IP address if set statically**: If the container was started with IP address the restored container also tries to use that IP address and restore fails if that IP address is already in use. This option is applicable if you added port mapping in the Integration tab when you create the container.
    - **Ignore MAC address if set statically**: If the container was started with MAC address the restored container also tries to use that MAC address and restore fails if that MAC address is already in use.
5. Click **Restore**.

Verification

- Click the **Podman containers** in the main menu. You can see that the restored container in the **Containers** table is running.

16.3. CREATING PODS IN THE WEB CONSOLE

You can create pods in the RHEL web console interface.

Prerequisites

- You have installed the RHEL 10 web console.
  For instructions, see Installing and enabling the web console .
- The **cockpit-podman** add-on is installed.

Procedure

1. Log in to the RHEL 10 web console.
2. Click **Podman containers** in the main menu.
3. Click **Create pod**.
4. Provide desired information in the **Create pod** form:
   - *Available only with the administrative access*: Select the Owner of the container: System or User.
   - In the **Name** field, enter the name of your container.
   - Click **Add port mapping** to add port mapping between container and host system.
     - Enter the IP address, Host port, Container port and Protocol.
   - Click **Add volume** to add volume.
     - Enter the host path, Container path. You can check the Writable checkbox to create a writable volume. In the SELinux drop down list, select one of the following options: No Label, Shared or Private.
5. Click **Create**.

Verification

- Click **Podman containers** in the main menu. You can see the newly created pod in the **Containers** table.

16.4. CREATING CONTAINERS IN THE POD IN THE WEB CONSOLE

You can create a container in a pod by using the RHEL web console GUI.

Prerequisites

- You have installed the RHEL 10 web console.

For instructions, see Installing and enabling the web console .

- The **cockpit-podman** add-on is installed.

Procedure

1. Log in to the RHEL 10 web console.
2. Click **Podman containers** in the main menu.
3. Click **Create container in pod**.
4. In the **Name** field, enter the name of your container.
5. Provide the required information in the **Details** tab.
    - *Available only with the administrative access*: Select the Owner of the container: System or User.
    - In the **Image** drop down list select or search the container image in selected registries.
        - Optional: Check the **Pull latest image** checkbox to pull the latest container image.
    - The **Command** field specifies the command. You can change the default command if you need.
        - Optional: Check the **With terminal** checkbox to run your container with a terminal.
    - The **Memory limit** field specifies the memory limit for the container. To change the default memory limit, check the checkbox and specify the limit.
    - *Available only for system containers*: In the **CPU shares field**, specify the relative amount of CPU time. Default value is 1024. Check the checkbox to modify the default value.
    - *Available only for system containers*: In the **Restart policy** drop down menu, select one of the following options:
        - **No** (default value): No action.
        - **On Failure**: Restarts a container on failure.
        - **Always**: Restarts container when exits or after system boot.
6. Provide the required information in the **Integration** tab.
    - Click **Add port mapping** to add port mapping between the container and host system.
        - Enter the *IP address*, *Host port*, *Container port* and *Protocol*.
    - Click **Add volume** to add volume.
        - Enter the *host path*, *Container path*. You can check the **Writable** option checkbox to create a writable volume. In the SELinux drop down list, select one of the following options: **No Label**, **Shared**, or **Private**.
    - Click **Add variable** to add environment variable.
        - Enter the *Key* and *Value*.

7. Provide the required information in the **Health check** tab.
   - In the **Command** fields, enter the healthcheck command.
   - Specify the healthcheck options:
     - **Interval** (default is 30 seconds)
     - **Timeout** (default is 30 seconds)
     - **Start period**
     - **Retries** (default is 3)
     - When unhealthy: Select one of the following options:
       - **No action** (default): Take no action.
       - **Restart**: Restart the container.
       - **Stop**: Stop the container.
       - **Force stop**: Force stops the container, it does not wait for the container to exit.

NOTE

The owner of the container and pod are same. In the pod, you can inspect containers, change the status of containers, commit containers, or delete containers.

Verification

- Click **Podman containers** in the main menu. You can see the newly created container in the pod under the **Containers** table.

CHAPTER 17. RUNNING SKOPEO, BUILDAH, AND PODMAN IN A CONTAINER

Run container tools such as Skopeo, Buildah, and Podman inside containers. This approach supports CI/CD workflows and rootless operations by isolating the tools from the host system.

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.

17.1. OVERVIEW OF SKOPEO, BUILDAH, AND PODMAN IN A CONTAINER

You can run Skopeo, Buildah, and Podman within a containerized environment on Red Hat Enterprise Linux. This offers several advantages, including rootless operation and a daemonless architecture. This setup allows you to manage, build, and inspect container images within a self-contained environment.

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 Hat recommends using Buildah in a container.
- **Different versions**:
  - **All**: You are running an older operating system 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.

17.2. RUNNING SKOPEO IN A CONTAINER

You can inspect a remote container image by 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 must mount files and directories.

Prerequisites

- The **container-tools** meta-package is installed.

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/rhel10/skopeo** container image:

   ```
   $ podman pull registry.redhat.io/rhel10/skopeo
   ```

3. Inspect a remote container image **registry.access.redhat.com/ubi10/ubi** using Skopeo:

   ```
   $ podman run --rm registry.redhat.io/rhel10/skopeo \
     skopeo inspect docker://registry.access.redhat.com/ubi10/ubi
   {
       "Name": "registry.access.redhat.com/ubi10/ubi",
       ...
       "Labels": {
           "architecture": "x86_64",
           ...
           "name": "ubi10",
           ...
           "summary": "Provides the latest release of Red Hat Universal Base Image 10.",
           "url":
   "https://access.redhat.com/containers/#/registry.access.redhat.com/ubi10/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/rhel10/skopeo** image after the container exits.

Additional resources

- How to run skopeo in a container

17.3. 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. For example, you can specify credentials on the command line by using the **--cred USERNAME[:PASSWORD]** option.

Prerequisites

- The **container-tools** meta-package is installed.

Procedure

- Inspect a remote container image using Skopeo against a locked registry:

```
$ podman run --rm registry.redhat.io/rhel10/skopeo inspect --creds
$USER:$PASSWORD docker://$IMAGE
```

Additional resources

- How to run Skopeo in a container

17.4. 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** meta-package is installed.

Procedure

- Inspect a remote container image using Skopeo against a locked registry:

  ```
  $ podman run --rm -v $AUTHFILE:/auth.json registry.redhat.io/rhel10/skopeo inspect
  docker://$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.

  The 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

17.5. 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 must 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** meta-package is installed.

Procedure

1. Copy the **registry.access.redhat.com/ubi10/ubi** image into your local container storage:

```
$ podman run --privileged --rm -v
$HOME/.local/share/containers/storage:/var/lib/containers/storage \
registry.redhat.io/rhel10/skopeo skopeo copy \
docker://registry.access.redhat.com/ubi10/ubi containers-
storage:registry.access.redhat.com/ubi10/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/rhel10/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/ubi10/ubi    latest  ecbc6f53bba0  8 weeks ago   211 MB
```

Additional resources

- How to run Skopeo in a container

17.6. RUNNING BUILDAH IN A CONTAINER

Run Buildah within a container to build images in an isolated environment. This enables image creation without modifying the host system.

Prerequisites

- The **container-tools** meta-package is installed.

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/rhel10/buildah** image:

   ```
   # podman run --rm --device /dev/fuse -it \
     registry.redhat.io/rhel10/buildah /bin/bash
   ```

   - The **--rm** option removes the **registry.redhat.io/rhel10/buildah** image after the container exits.
   - The **--device** option adds a host device to the container.
   - The **sys_chroot** - capability to change to a different root directory. It is not included in the default capabilities of a container.

3. Create a new container using a **registry.access.redhat.com/ubi10/** image:

   ```
   # buildah from registry.access.redhat.com/ubi10/
   ...
   ubi10/-working-container
   ```

4. Run the **ls /** command inside the **ubi10/-working-container** container:

   ```
   # buildah run --isolation=chroot ubi10/-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/ubi10/   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... ubi10/-working-
   ```

```
container
```

7. Optional: Push the **registry.access.redhat.com/ubi10/** image to the a local registry located on **registry.example.com**:

```
# buildah push ecbc6f53bba0 registry.example.com:5000/ubi10/ubi
```

Additional resources

- Best practices for running Buildah in a container

17.7. PRIVILEGED AND UNPRIVILEGED PODMAN CONTAINERS

By default, Podman containers are unprivileged and cannot, for example, modify parts of the host operating system. This is because a container has only limited access to devices by default.

The following list emphasizes important properties of privileged containers. You can run the privileged container by 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 or 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

17.8. RUNNING PODMAN WITH EXTENDED PRIVILEGES

You can run Podman within a container using extended privileges when rootless operation is not feasible. Use this method with caution as it reduces security isolation.

Prerequisites

- The **container-tools** meta-package is installed.

Procedure

- Run the Podman container in the Podman container:

```
$ podman run --privileged --name=privileged_podman \
  registry.access.redhat.com//podman podman run ubi10 echo hello
Resolved "ubi10" as an alias (/etc/containers/registries.conf.d/001-rhel-shortnames.conf)
Trying to pull registry.access.redhat.com/ubi10:latest...
...
Storing signatures
hello
```

- Run the outer container named **privileged_podman** based on the **registry.access.redhat.com/ubi10/podman** image.
- The **--privileged** option disables the security features that isolate the container from the host.
- Run **podman run ubi10 echo hello** command to create the inner container based on the **ubi10** image.
- Notice that the **ubi10** short image name was resolved as an alias. As a result, the **registry.access.redhat.com/ubi10:latest** image is pulled.

Verification

- List all containers:

```
$ podman ps -a
CONTAINER ID  IMAGE                            COMMAND           CREATED          STATUS
PORTS   NAMES
52537876caf4  registry.access.redhat.com/ubi10/podman            podman run ubi10 e...  30
seconds ago     Exited (0) 13 seconds ago             privileged_podman
```

For more information, see the **podman-run(1)** man page on your system.

Additional resources

- How to use Podman inside of a container

17.9. 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** meta-package is installed.

Procedure

- Run two nested containers:

```
$ podman run --name=unprivileged_podman --security-opt label=disable \
  --user podman --device /dev/fuse \
  registry.access.redhat.com/ubi10/podman \
  podman run ubi10 echo hello
```

- Run the outer container named **unprivileged_podman** based on the **registry.access.redhat.com/ubi10/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 ubi10 echo hello** command to create the inner container based on the **ubi10** image.
- Notice that the ubi10 short image name was resolved as an alias. As a result, the **registry.access.redhat.com/ubi10:latest** image is pulled.

Verification

- List all containers:

```
$ podman ps -a
CONTAINER ID  IMAGE                            COMMAND           CREATED          STATUS
PORTS   NAMES
a47b26290f43            podman run ubi10 e...  30 seconds ago    Exited (0) 13 seconds ago
unprivileged_podman
```

17.10. BUILDING A CONTAINER INSIDE A PODMAN CONTAINER

You can build container images from within an existing Podman container to create isolated development environments or automate image construction in CI/CD pipelines. This approach on RHEL allows you to test and develop images without modifying your host system configuration.

Prerequisites

- The **container-tools** meta-package is installed.
- 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/rhel10/podman** image:

```
# podman run --privileged --name podman_container -it \
  registry.redhat.io/rhel10/podman /bin/bash
```

   - Run the outer container named **podman_container** based on the **registry.redhat.io/rhel10/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/ubi10/ubi
RUN dnf install -y https://dl.fedoraproject.org/pub/epel/epel-release-latest-8.noarch.rpm
RUN dnf -y install moon-buggy && dnf 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/ubi10/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/ubi10/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
```

This example shows how to use Podman to build and run another container from within this container. The container runs the "Moon-buggy", a simple text-based game.

CHAPTER 18. MONITORING CONTAINERS

You can monitor containers on RHEL by using tools such as Podman and the RHEL web console. Key methods include inspecting container logs, monitoring resource usage, and checking the health of container processes.

18.1. 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 by 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 by 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 outlay much 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** 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.

For more information, see the **podman-healthcheck(1)** and **podman-run(1)** man pages on your system.

Additional resources

- Monitoring container vitality and availability with Podman

18.2. PERFORMING A HEALTH CHECK USING THE COMMAND LINE

You can set a health check when creating the container on the command line. With this, you can specify commands and intervals to monitor the container's application status.

Prerequisites

- The **container-tools** meta-package is installed.

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/ubi10/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
     ```

     For more information, see the **podman-healthcheck(1)** and **podman-run(1)** man pages on your system

Additional resources

- Monitoring container vitality and availability with Podman

18.3. PERFORMING A HEALTH CHECK USING A CONTAINERFILE

You can set a health check by using the **HEALTHCHECK** instruction in the **Containerfile**.

Prerequisites

- The **container-tools** meta-package is installed.

Procedure

1. Create a **Containerfile**:

   ```
   $ cat Containerfile
   FROM registry.access.redhat.com/ubi10/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/ubi10/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
```

For more information, see **podman-healthcheck(1)** and **podman-run(1)** man pages on your system.

Additional resources

- Monitoring container vitality and availability with Podman

18.4. DISPLAYING PODMAN SYSTEM INFORMATION

View comprehensive system information and disk usage for Podman by using the **podman system** command. This helps you monitor storage consumption and verify environment details.

Prerequisites

- The **container-tools** meta-package is installed.

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 about 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/ubi10          latest      b1e63aaae5cf  13 days     233.4MB
233.4MB      0B           0
registry.access.redhat.com/ubi10/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 ubi10 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: v2
  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.
    For more information, see the **podman-system-df**, **podman-system-info**, and **podman-system-prune** man pages on your system.

18.5. PODMAN EVENT TYPES

Understand the various event types reported by Podman, such as container creation, image pulling, and pod lifecycle changes. Monitoring these events helps track system activity.

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

For more information, see the **podman-events(1)** man page on your system.

18.6. MONITORING PODMAN EVENTS

Track real-time system activity by using the **podman events** command. This displays a stream of events with timestamps and details, useful for auditing and troubleshooting.

Prerequisites

- The **container-tools** meta-package is installed.

Procedure

1. Run the **myubi** container:

   ```
   $ podman run -q --rm --name=myubi registry.access.redhat.com/ubi10/ubi:latest
   ```

2. Display the Podman events:

   - To display all Podman events, enter:

     ```
     $ now=$(date --iso-8601=seconds)
     $ podman events --since=now --stream=false
     2023-03-08 14:27:20.696167362 +0100 CET container create
     d4748226a2bcd271b1bc4b9f88b54e8271c13ffea9b30529968291c62d72fe09
     (image=registry.access.redhat.com/ubi10/ubi:latest, name=myubi,...)
     2023-03-08 14:27:20.652325082 +0100 CET image pull
     registry.access.redhat.com/ubi10/ubi:latest
     2023-03-08 14:27:20.795695396 +0100 CET container init
     d4748226a2bcd271b1bc4b9f88b54e8271c13ffea9b30529968291c62d72fe09
     (image=registry.access.redhat.com/ubi10/ubi:latest, name=myubi...)
     2023-03-08 14:27:20.809205161 +0100 CET container start
     d4748226a2bcd271b1bc4b9f88b54e8271c13ffea9b30529968291c62d72fe09
     (image=registry.access.redhat.com/ubi10/ubi:latest, name=myubi...)
     2023-03-08 14:27:20.809903022 +0100 CET container attach
     d4748226a2bcd271b1bc4b9f88b54e8271c13ffea9b30529968291c62d72fe09
     (image=registry.access.redhat.com/ubi10/ubi:latest, name=myubi...)
     2023-03-08 14:27:20.831710446 +0100 CET container died
     d4748226a2bcd271b1bc4b9f88b54e8271c13ffea9b30529968291c62d72fe09
     (image=registry.access.redhat.com/ubi10/ubi:latest, name=myubi...)
     2023-03-08 14:27:20.913786892 +0100 CET container remove
     d4748226a2bcd271b1bc4b9f88b54e8271c13ffea9b30529968291c62d72fe09
     (image=registry.access.redhat.com/ubi10/ubi:latest, name=myubi...)
     ```

     The **--stream=false** option ensures that the **podman events** command exits when reading the last known event.

     You can see several events that happened when you enter the **podman run** command:

     - **container create** when creating a new container.

- **image pull** when pulling an image if the container image is not present in the local storage.
- **container init** when initializing the container in the runtime and setting a network.
- **container start** when starting the container.
- **container attach** when attaching to the terminal of a container. That is because the container runs in the foreground.
- **container died** is emitted when the container exits.
- **container remove** because the **--rm** flag was used to remove the container after it exits.

- You can also use the **journalctl** command to display Podman events:

```
$ journalctl --user -r SYSLOG_IDENTIFIER=podman
Mar 08 14:27:20 fedora podman[129324]: 2023-03-08 14:27:20.913786892 +0100 CET
m=+0.066920979 container remove
...
Mar 08 14:27:20 fedora podman[129289]: 2023-03-08 14:27:20.696167362 +0100 CET
m=+0.079089208 container create
d4748226a2bcd271b1bc4b9f88b54e8271c13ffea9b30529968291c62d72f>
```

- To show only Podman create events, enter:

```
$ podman events --filter event=create
2023-03-08 14:27:20.696167362 +0100 CET container create
d4748226a2bcd271b1bc4b9f88b54e8271c13ffea9b30529968291c62d72fe09
(image=registry.access.redhat.com/ubi10/ubi:latest, name=myubi,...)
```

- You can also use the **journalctl** command to display Podman create events:

```
$ journalctl --user -r PODMAN_EVENT=create
Mar 08 14:27:20 fedora podman[129289]: 2023-03-08 14:27:20.696167362 +0100 CET
m=+0.079089208 container create
d4748226a2bcd271b1bc4b9f88b54e8271c13ffea9b30529968291c62d72f>
```

For more information, see the **podman-events(1)** man page on your system.

Additional resources

- Container events and auditing

18.7. USING PODMAN EVENTS FOR AUDITING

Enhance auditing by configuring Podman to include inspect data in event logs. This provides detailed context for events, such as security settings and configuration, directly in the log entry.

Previously, the events had to be connected to an event to interpret them correctly. For example, the **container-create** event had to be linked with an **image-pull** event to know which image had been used. The **container-create** event also did not include all data, for example, the security settings, volumes, mounts, and so on.

Beginning with Podman v4.4, you can gather all relevant information about a container directly from a single event and **journald** entry. The data is in JSON format, the same as from the **podman container inspect** command and includes all configuration and security settings of a container. You can configure Podman to attach the container-inspect data for auditing purposes.

Prerequisites

- The **container-tools** meta-package is installed.

Procedure

1. Modify the **~/.config/containers/containers.conf** file and add the **events_container_create_inspect_data=true** option to the **[engine]** section:

   ```
   $ cat ~/.config/containers/containers.conf
   [engine]
   events_container_create_inspect_data=true
   ```

   For the system-wide configuration, modify the **/etc/containers/containers.conf** or **/usr/share/container/containers.conf** file.

2. Create the container:

   ```
   $ podman create registry.access.redhat.com/ubi10/ubi:latest
   19524fe3c145df32d4f0c9af83e7964e4fb79fc4c397c514192d9d7620a36cd3
   ```

3. Display the Podman events:

   - Using the **podman events** command:

     ```
     $ now=$(date --iso-8601=seconds)
     $ podman events --since $now --stream=false --format "{{.ContainerInspectData}}"
     | jq ".Config.CreateCommand"
     [
       "/usr/bin/podman",
       "create",
       "registry.access.redhat.com/ubi10"
     ]
     ```

     - The **--format "{{.ContainerInspectData}}"** option displays the inspect data.
     - The **jq ".Config.CreateCommand"** transforms the JSON data into a more readable format and displays the parameters for the **podman create** command.

   - Using the **journalctl** command:

     ```
     $ journalctl --user -r PODMAN_EVENT=create --all -o json | jq
     ".PODMAN_CONTAINER_INSPECT_DATA | fromjson" | jq
     ".Config.CreateCommand"
     [
       "/usr/bin/podman",
       "create",
       "registry.access.redhat.com/ubi10"
     ]
     ```

The output data for the **podman events** and **journalctl** commands are the same.

For more information, see the **podman-events(1)** and **containers.conf(5)** man pages on your system.

Additional resources

- Container Events and Auditing

CHAPTER 19. USING TOOLBX FOR DEVELOPMENT AND TROUBLESHOOTING

Use Toolbx containers to install development tools without modifying the host system. Toolbx provides a seamless environment where you can access host resources while keeping your system clean.

Installing software on a system presents certain risks: it can change a system's behavior, and can leave unwanted files and directories behind after they are no longer needed. You can prevent these risks by installing your favorite development and debugging tools, editors, and software development kits (SDKs) into the Toolbx fully mutable container without affecting the base operating system. You can perform changes on the host system with commands such as **less**, **lsof**, **rsync**, **ssh**, **sudo**, and **unzip**.

You can install development tools and SDKs in a Toolbx container to avoid system changes and unwanted files on your base operating system. You can also perform changes on the host system with commands such as **less**, **lsof**, **rsync**, **ssh**, **sudo**, and **unzip**. The Toolbx utility performs the following actions:

1. Pulling the **registry.access.redhat.com/ubi10/toolbox:latest** image to your local system
2. Starting up a container from the image
3. Running a shell inside the container from which you can access the host system

NOTE

Whether a Toolbx container runs as a root or rootless container depends on the rights of the user who creates it. You should also run utilities that require root rights on the host system in root containers.

19.1. STARTING A TOOLBX CONTAINER

You can start a Toolbx container to enter an isolated environment for development or troubleshooting. Entering the container ensures you can install and run command-line tools without modifying the underlying host operating system.

Procedure

1. Create a Toolbx container:
   - As a rootless user:

     ```
     $ toolbox create <mytoolbox>
     Created container: <mytoolbox>
     Enter with: toolbox enter <mytoolbox>
     ```

   - As a root user:

     ```
     $ sudo toolbox create <mytoolbox>
     Created container: <mytoolbox>
     Enter with: toolbox enter <mytoolbox>
     ```

   - Verify that you pulled the correct image:

     ```
     [user@toolbox ~]$ toolbox list
     ```

```
IMAGE ID      IMAGE NAME    CREATED
fe0ae375f149   registry.access.redhat.com/ubi10/toolbox:latest 5 weeks ago

CONTAINER ID  CONTAINER NAME  CREATED         STATUS   IMAGE NAME
5245b924c2cb  <mytoolbox>       7 minutes ago   created
registry.access.redhat.com/ubi10/toolbox:latest
```

2. Enter the Toolbx container:

```
[user@toolbox ~]$ toolbox enter <mytoolbox>
```

Verification

- Enter a command inside the ***<mytoolbox>*** container and display the name of the container and the image:

```
⬢ [user@toolbox ~]$ cat /run/.containerenv
engine="podman-4.8.2"
name="<mytoolbox>"
id="5245b924c2cb..."
image="registry.access.redhat.com/ubi10/toolbox:latest"
imageid="fe0ae375f14919cbc0596142e3aff22a70973a36e5a165c75a86ea7ec5d8d65c"
```

19.2. USING TOOLBX FOR DEVELOPMENT

You can use a Toolbx container as a rootless user for installation of development tools, such as editors, compilers, and software development kits (SDKs). After installation, you can continue using those tools as a rootless user.

Prerequisites

- The Toolbx container is created and is running. You entered the Toolbx container. You do not need to create the Toolbx container with root privileges. See Starting a Toolbox container.

Procedure

- Install the tools of your choice, for example, the Emacs text editor, GCC compiler and GNU Debugger (GDB):

```
⬢[user@toolbox ~]$ sudo dnf install emacs gcc gdb
```

Verification

- Verify that the tools are installed:

```
⬢[user@toolbox ~]$  dnf repoquery --info --installed <package_name>
```

19.3. USING TOOLBX FOR TROUBLESHOOTING A HOST SYSTEM

You can use a Toolbx container with root privileges to find the root cause of various problems with the host system by using tools such as **systemctl**, **journalctl** , and **nmap**, without installing them on the host system.

Prerequisites

- The Toolbx container is created and is running. You entered the Toolbx container. You need to create the Toolbx container with root privileges. See Starting a Toolbox container.

Procedure

1. Install the **systemd** package to be able to run the **journalctl** command:

```
⬢[root@toolbox ~]# dnf install systemd
```

2. Display log messages for all processes running on the host:

```
⬢[root@toolbox ~]# j journalctl --boot -0
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: microcode: updated ear>
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: Linux version 6.6.8-10>
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: Command line: BOOT_IMA>
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: x86/split lock detecti>
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: BIOS-provided physical>
```

3. Display log messages for the kernel:

```
⬢[root@toolbox ~]# journalctl --boot -0 --dmesg
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: microcode: updated ear>
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: Linux version 6.6.8-10>
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: Command line: BOOT_IMA>
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: x86/split lock detecti>
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: BIOS-provided physical>
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: BIOS-e820: [mem 0x0000>
```

4. Install the **nmap** network scanning tool:

```
⬢[root@toolbox ~]# dnf install nmap
```

5. Scan IP addresses and ports in a network:

```
⬢[root@toolbox ~]# nmap -sS scanme.nmap.org
Starting Nmap 7.93 ( https://nmap.org ) at 2024-01-02 10:39 CET
Stats: 0:01:01 elapsed; 0 hosts completed (0 up), 256 undergoing Ping Scan
Ping Scan Timing: About 29.79% done; ETC: 10:43 (0:02:24 remaining)
Nmap done: 256 IP addresses (0 hosts up) scanned in 206.45 seconds
```

   - The **-sS** option performs a TCP SYN scan. Most of Nmap’s scan types are only available to privileged users, because they send and receive raw packets, which requires root access on UNIX systems.

19.4. STOPPING THE TOOLBX CONTAINER

You can stop an active Toolbx container to release system resources and conclude your development session. Ending the container process ensures that background tasks are terminated, enabling you to maintain a clean and efficient environment for subsequent workloads.

Use the **exit** command to leave the Toolbox container and the **podman stop** command to stop the container.

Procedure

1. Leave the container and return to the host:

```
⬢ [user@toolbox ~]$ exit
```

2. Stop the toolbox container:

```
⬢ [user@toolbox ~]$ podman stop <mytoolbox>
```

3. Optional: Remove the toolbox container:

```
⬢ [user@toolbox ~]$ toolbox rm <mytoolbox>
```

Alternatively, you can also use the **podman rm** command to remove the container.

CHAPTER 20. RUNNING SPECIAL CONTAINER IMAGES

Run containers with predefined commands that use built-in labels called *runlabels*. With these labels, you can trigger specific actions such as install, run, or uninstall directly from the image metadata.

20.1. OPENING PRIVILEGES TO THE HOST

Understand the capabilities of privileged containers, which can access host resources. Privileged containers, such as Toolbx, bypass standard isolation to interact with the host system.

- **Privileges**: A privileged container disables the security features that isolate the container from the host. You can run a privileged container by 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.

20.2. CONTAINER IMAGES WITH RUNLABELS

You can use runlabels to enter predefined commands stored within container images. Common labels such as install and run simplify the deployment and management of complex containers.

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 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.

20.3. RUNNING SUPPORT-TOOLS WITH RUNLABELS

Deploy the **support-tools** container by using its built-in runlabels. This method simplifies installation, execution, and removal by using the commands defined directly in the image.

The **rhel10/support-tools** container image is made to run a containerized version of the **support-toolsd** daemon. The **support-tools** image contains the following runlabels: **install**, **run** and **uninstall**.

Prerequisites

- The **container-tools** meta-package is installed.

Procedure

1. Pull the **support-tools** image:

```
# podman pull registry.redhat.io/rhel10/support-tools
```

2. Display the **install** runlabel for **support-tools**:

```
# podman container runlabel install --display rhel10/support-tools
command: podman run --rm --privileged -v /:/host -e HOST=/host -e
IMAGE=registry.redhat.io/rhel10/support-tools:latest -e NAME=support-tools
registry.redhat.io/rhel10/support-tools: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 **support-tools**:

```
# podman container runlabel install rhel10/support-tools
command: podman run --rm --privileged -v /:/host -e HOST=/host -e
IMAGE=registry.redhat.io/rhel10/support-tools:latest -e NAME=support-tools
registry.redhat.io/rhel10/support-tools:latest /bin/install.sh
Creating directory at /host//etc/pki/support-tools
Creating directory at /host//etc/support-tools.d
Installing file at /host//etc/support-tools.conf
Installing file at /host//etc/sysconfig/support-tools
Installing file at /host//etc/logrotate.d/syslog
```

This creates files on the host system that the **support-tools** image will use later.

4. Display the **run** runlabel for **support-tools**:

```
# podman container runlabel run --display rhel10/support-tools
command: podman run -d --privileged --name support-tools --net=host --pid=host -v
/etc/pki/support-tools:/etc/pki/support-tools -v /etc/support-tools.conf:/etc/support-tools.conf -v
/etc/sysconfig/support-tools:/etc/sysconfig/support-tools -v /etc/support-tools.d:/etc/support-
tools.d -v /var/log:/var/log -v /var/lib/support-tools:/var/lib/support-tools -v /run:/run -v
/etc/machine-id:/etc/machine-id -v /etc/localtime:/etc/localtime -e
IMAGE=registry.redhat.io/rhel10/support-tools:latest -e NAME=support-tools --restart=always
registry.redhat.io/rhel10/support-tools:latest /bin/support-tools.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 **support-tools** container to run the **support-toolsd** daemon.

5. Execute the **run** runlabel for **support-tools**:

```
# podman container runlabel run rhel10/support-tools
command: podman run -d --privileged --name support-tools --net=host --pid=host -v
/etc/pki/support-tools:/etc/pki/support-tools -v /etc/support-tools.conf:/etc/support-tools.conf -v
```

```
/etc/sysconfig/support-tools:/etc/sysconfig/support-tools -v /etc/support-tools.d:/etc/support-
tools.d -v /var/log:/var/log -v /var/lib/support-tools:/var/lib/support-tools -v /run:/run -v
/etc/machine-id:/etc/machine-id -v /etc/localtime:/etc/localtime -e
IMAGE=registry.redhat.io/rhel10/support-tools:latest -e NAME=support-tools --restart=always
registry.redhat.io/rhel10/support-tools:latest /bin/support-tools.sh
28a0d719ff179adcea81eb63cc90fcd09f1755d5edb121399068a4ea59bd0f53
```

The **support-tools** container opens privileges, mounts what it needs from the host, and runs the **support-toolsd** daemon in the background (**-d**). The **support-toolsd** daemon begins gathering log messages and directing messages to files in the **/var/log** directory.

6. Display the **uninstall** runlabel for **support-tools**:

```
# podman container runlabel uninstall --display rhel10/support-tools
command: podman run --rm --privileged -v /:/host -e HOST=/host -e
IMAGE=registry.redhat.io/rhel10/support-tools:latest -e NAME=support-tools
registry.redhat.io/rhel10/support-tools:latest /bin/uninstall.sh
```

7. Run the **uninstall** runlabel for **support-tools**:

```
# podman container runlabel uninstall rhel10/support-tools
command: podman run --rm --privileged -v /:/host -e HOST=/host -e
IMAGE=registry.redhat.io/rhel10/support-tools:latest -e NAME=support-tools
registry.redhat.io/rhel10/support-tools: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 21. USING THE CONTAINER-TOOLS API

Interact with Podman programmatically by using its REST-based API. This API supports both Libpod and Docker-compatible endpoints, allowing integration with external tools and clients.

Podman 2.0's new REST-based API, replacing the old varlink remote API, provides both rootful and rootless support. It includes the Libpod API for Podman and a Docker-compatible API to call Podman from platforms such as cURL, Postman, Google's Advanced REST client, and others.

NOTE

Because the Podman service relies on socket activation, it cannot run unless connections are active. You must manually start the **podman.socket** service to enable this feature. When a connection activates, the Podman service starts, executes the requested API action, then ends, returning to an inactive state.

21.1. ENABLING THE PODMAN API USING SYSTEMD IN ROOT MODE

You can enable the Podman API on RHEL by starting and enabling the systemd socket unit in root mode. This configuration facilitates system-wide container management and integrates Podman seamlessly with existing infrastructure while providing a stable endpoint for external management tools.

Prerequisites

- The **podman-remote** package is installed.

Procedure

1. Start the service immediately:

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

2. To enable the link to **var/lib/docker.sock** by using the **docker-podman** package:

```
# dnf install podman-docker
```

Verification

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

21.2. ENABLING THE PODMAN API USING SYSTEMD IN ROOTLESS MODE

You can use systemd to activate the Podman API socket and API service.

Prerequisites

- The **podman-remote** package is installed.

Procedure

1. Enable and start the service immediately:

```
$ systemctl --user enable --now podman.socket
```

2. Optional: To enable programs by using Docker to interact with the rootless Podman socket:

```
$ export DOCKER_HOST=unix:///run/user/<uid>/podman//podman.sock
```

Verification

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

21.3. RUNNING THE PODMAN API MANUALLY

You can run the Podman API. This is useful for debugging API calls, especially when using the Docker compatibility layer.

Prerequisites

- The **podman-remote** package is installed.

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": "v2",
       "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/ubi10/ubi** container image:

   ```
   # curl -XPOST --unix-socket /run/podman/podman.sock -v
   'http://d/v1.0.0/images/create?
   fromImage=registry.access.redhat.com%2Fubi10%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%2Fubi10%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/ubi10/ubi:latest,
registry.redhat.io/ubi10/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/ubi10/ubi:latest",
"registry.redhat.io/ubi10/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": "ubi10", ... "summary": "Provides the latest release of Red Hat Universal
Base Image 8.", "url":
"https://access.redhat.com/containers//registry.access.redhat.com/ubi10/images/8.2-347",
      ...
    },
    "Names": [
      "registry.access.redhat.com/ubi10/ubi:latest",
      "registry.redhat.io/ubi10/ubi:latest"
    ],
    ...
    ]
  }
]
```