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

This guide describes how to work with Linux containers on RHEL 8 systems using command-line tools such as podman, buildah, skopeo and runc.
11.1. PODMAN
11.1.1. Using podman commands
11.1.2. Creating SELinux policies for containers
11.1.3. Using podman with MPI
11.1.4. Creating and restoring container checkpoints
   11.1.4.1. Creating and restoring a container checkpoint locally
   11.1.4.2. Reducing startup time using container restore
   11.1.4.3. Migrating containers among systems
11.2. RUNC
   11.2.1. Running containers with runc
11.3. SKOPEO
   11.3.1. Inspecting container images with skopeo
   11.3.2. Copying container images with skopeo
   11.3.3. Getting image layers with skopeo

CHAPTER 12. USING THE CONTAINER-TOOLS API ................................................. 103
12.1. ENABLING THE PODMAN API USING SYSTEMD IN ROOT MODE 103
12.2. ENABLING THE PODMAN API USING SYSTEMD IN ROOTLESS MODE 104
12.3. RUNNING THE PODMAN API MANUALLY 106

CHAPTER 13. ADDITIONAL RESOURCES ......................................................... 109
PREFACE

Red Hat classifies container use cases into two distinct groups: single node and multi-node, with multi-node sometimes called distributed systems. OpenShift was built to provide public, scalable deployments of containerized applications. Beyond OpenShift, however, it is useful to have a small, nimble set of tools for working with containers.

The set of container tools we are referring to can be used in a single-node use case. However, you can also wire these tools into existing build systems, CI/CD environments, and even use them to tackle workload-specific use cases, such as big data. To target the single-node use case, Red Hat Enterprise Linux (RHEL) 8 offers a set of tools to find, run, build, and share individual containers.

This guide describes how to work with Linux containers on RHEL 8 systems using command-line tools such as podman, buildah, skopeo and runc. In addition to these tools, Red Hat provides base images, to act as the foundation for your own images. Some of these base images target use cases ranging from business applications (such as Node.js, PHP, Java, and Python) to infrastructure (such as logging, data collection, and authentication).
MAKING OPEN SOURCE MORE INCLUSIVE

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

We appreciate your input on our documentation. Please let us know how we could make it better. To do so:

- For simple comments on specific passages:
  1. Make sure you are viewing the documentation in the *Multi-page HTML* format. In addition, ensure you see the Feedback button in the upper right corner of the document.
  2. Use your mouse cursor to highlight the part of text that you want to comment on.
  3. Click the Add Feedback pop-up that appears below the highlighted text.
  4. Follow the displayed instructions.

- For submitting more complex feedback, create a Bugzilla ticket:
  1. Go to the Bugzilla website.
  2. As the Component, use Documentation.
  3. Fill in the Description field with your suggestion for improvement. Include a link to the relevant part(s) of documentation.
  4. Click Submit Bug.
CHAPTER 1. STARTING WITH CONTAINERS

Linux containers have emerged as a key open source application packaging and delivery technology, combining lightweight application isolation with the flexibility of image-based deployment methods.

Red Hat Enterprise Linux implements Linux containers using core technologies such as:

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

to reduce the potential for security exploits. All this is meant to 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. However, there are times when you might want to build and manage individual containers and container images outside of OpenShift. Tools provided to perform those tasks that run directly on RHEL systems are described in this guide.

Unlike other container tools implementations, tools described here do not center around the monolithic Docker container engine and docker command. Instead, we provide 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

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. Instead of relying on the single-node, daemonless tools described in this document, OpenShift requires a daemon-based container engine. Please see Using the CRI-O Container Engine for details.

1.1. RUNNING CONTAINERS WITHOUT DOCKER

Red Hat did not just remove the Docker container engine from OpenShift. It also removed the Docker container engine, along with the docker command, from Red Hat Enterprise Linux 8 entirely. For RHEL 8, Docker is not included and not supported by Red Hat (although it is still available from other sources).

The removal of Docker reflects a change in Red Hat’s way of thinking about how containers are handled:

- In the enterprise, the focus is not on running individual containers from the command line. The primary venue for running containers is a Kubernetes-based platform, such as OpenShift.
By repositioning OpenShift as the platform for running containers, container engines like Docker become just another component abstracted by OpenShift.

Because the container engine in OpenShift is not meant to be used directly, it can be implemented with a limited feature set that focuses on doing everything that OpenShift needs, without having to implement many standalone features.

Although Docker is gone from RHEL 8, and OpenShift’s container engine is disconnected from single-node uses, people still want to use commands to work with containers and images manually. So Red Hat set about to create a set of tools to implement most of what the `docker` command does.

Tools like `podman`, `skopeo`, and `buildah` were developed to take over those `docker` command features. Each tool in this scenario can be more lightweight and focused on a subset of features. And with no need for a daemon process running to implement a container engine, these tools can run without the overhead of having to work with a daemon process.

If you still want to use Docker in RHEL 8, you can get Docker from different upstream projects, but it is unsupported in RHEL 8. Because so many `docker` command-line features have been implemented exactly in `podman`, you can set up an alias so that typing `docker` causes podman to run.

Installing the podman-docker package sets up such an alias. So every time you run a `docker` command line, it actually runs podman for you.

### 1.2. CHOOSING A RHEL ARCHITECTURE FOR CONTAINERS

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

- AMD64 and Intel 64 (base and layered images; no support for 32-bit architectures)
- PowerPC 8 and 9 64-bit (base image and most layered images)
- 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. See Universal Base Images (UBI): Images, repositories, and packages for a list of supported images.

### 1.3. GETTING CONTAINER TOOLS

To get an environment where you can manipulate individual containers, you can install a Red Hat Enterprise Linux 8 system, then add a set of container tools to find, run, build and share containers. Here are examples of container-related tools you can install with RHEL 8:

- **podman** - Client tool for managing containers. Can replace most features of the `docker` command for working with individual containers and images.
- **buildah** - Client tool for building OCI-compliant container images.
- **skopeo** - Client tool for copying container images to and from container registries. Includes features for signing and authenticating images as well.
- **runc** - Container runtime client for running and working with Open Container Initiative (OCI) format containers.
If you want to create container images using the RHEL subscription model, you must properly register and entitle the host computer on which you build them. When you install packages, as part of the process of building a container, the build process automatically has access to entitlements available from the RHEL host. So it can get RPM packages from any repository enabled on that host.

1. **Install RHEL**: If you are ready to begin, you can start by installing a Red Hat Enterprise Linux system.

2. **Register RHEL**: Once RHEL is installed, register the system. You will be prompted to enter your user name and password. Note that the user name and password are the same as your login credentials for Red Hat Customer Portal.
   ```bash
   # subscription-manager register
   Registering to: subscription.rhsm.redhat.com:443/subscription
   Username: ********
   Password: *********
   ```

3. **Subscribe RHEL**: Either auto subscribe or determine the pool ID of a subscription that includes Red Hat Enterprise Linux. Here is an example of auto-attaching a subscription:
   ```bash
   # subscription-manager attach --auto
   ```

4. **Install packages**: To start building and working with individual containers, install the container-tools module, which pulls in the full set of container software packages:
   ```bash
   # yum module install -y container-tools
   ```

5. **Install podman-docker (optional)**: If you are comfortable with the `docker` command or use scripts that call `docker` directly, you can install the podman-docker package. That package installs a link that replaces the `docker` command-line interface with the matching `podman` commands instead. It also links the man pages together, so `man docker info` will show the `podman info` man page.
   ```bash
   # yum install -y podman-docker
   ```

### 1.4. RUNNING CONTAINERS AS ROOT OR ROOTLESS

Running the container tools such as `podman`, `skopeo`, or `buildah` as a user with superuser privilege (root user) is the best way to ensure that your containers have full access to any feature available on your system. However, with the feature called "Rootless Containers," generally available as of RHEL 8.1, you can work with containers as a regular user.

Although container engines, such as Docker, let you run `docker` commands as a regular (non-root) user, the docker daemon that carries out those requests runs as root. So, effectively, regular users can make requests through their containers that harm the system, without there being clarity about who made those requests. By setting up rootless container users, system administrators limit potentially damaging container activities from regular users, while still allowing those users to safely run many container features under their own accounts.

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

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

1. **Install RHEL**: Install RHEL 8.1 or upgrade to RHEL 8.1 from RHEL 8.0. Earlier RHEL 7 versions are missing features needed for this procedure. If you are upgrading from RHEL 7.6 or earlier, continue to “Upgrade to rootless containers” after this procedure is done.

2. **Install podman and slirp4netns**: If not already installed, install the podman and slirp4netns packages:
   
   ```
   # yum install slirp4netns podman -y
   ```

3. **Increase user namespaces**: To increase the number of user namespaces in the kernel, type the following:
   
   ```
   # echo "user.max_user_namespaces=28633" > /etc/sysctl.d/userns.conf
   # sysctl -p /etc/sysctl.d/userns.conf
   ```

4. **Create a new user account**: To create a new user account and add a password for that account (for example, joe), type the following:
   
   ```
   # useradd -c "Joe Jones" joe
   # passwd joe
   ```
   
   The user is automatically configured to be able to use rootless podman. In case you would like to enable an existing user to use rootless podman see section **Upgrade to rootless containers**.

5. If you want to run containers with systemd, see section **Using the UBI init images**.

6. **Try a podman command**: Log in directly as the user you just configured (don’t use su or su - to become that user because that doesn’t set the correct environment variables) and try to pull and run an image:
   
   ```
   $ podman pull registry.access.redhat.com/ubi8/ubi
   $ podman run registry.access.redhat.com/ubi8/ubi cat /etc/os-release
   NAME="Red Hat Enterprise Linux"
   VERSION="8.1 (Ootpa)"
   ...
   ```

7. **Check rootless configuration**: To check that your rootless configuration is set up properly, you can run commands inside the modified user namespace with the podman unshare command. As the rootless user, the following command lets you see how the uids are assigned to the user namespace:
   
   ```
   $ podman unshare cat /proc/self/uid_map
   0 1001 1
   1 65537 65536
   ```

1.4.2. Upgrade to rootless containers

If you have upgraded from RHEL 7, you must configure subuid and subgid values manually for any existing user you want to be able to use rootless podman.
Using an existing user name and group name (for example, jill), set the range of accessible user and group IDs that can be used for their containers. Here are a couple of warnings:

- Don’t include the rootless user’s UID and GID in these ranges
- If you set multiple rootless container users, use unique ranges for each user
- We recommend 65536 UIDs and GIDs for maximum compatibility with existing container images, but the number can be reduced
- Never use UIDs or GIDs under 1000 or reuse UIDs or GIDs from existing user accounts (which, by default, start at 1000)

Here is an example:

```
# echo "jill:165537:65536" >> /etc/subuid
# echo "jill:165537:65536" >> /etc/subgid
```

The user/group jill is now allocated 65535 user and group IDs, ranging from 165537-231072. That user should be able to begin running commands to work with containers now.

1.4.3. Special considerations for rootless

Here are some things to consider when running containers as a non-root user:

- As a non-root container user, container images are stored under your home directory ($HOME/.local/share/containers/storage/), instead of /var/lib/containers.
- Users running rootless containers are given special permission to run as a range of user and group IDs on the host system. However, they otherwise have no root privileges to the operating system on the host.
- If you need to configure your rootless container environment, edit configuration files in your home directory ($HOME/.config/containers). Configuration files include storage.conf (for configuring storage) and libpod.conf (for a variety of container settings). You could also create a registries.conf file to identify container registries available when you use podman to pull, search or run images.
- A container running as root in a rootless account can turn on privileged features within its own namespace. But that doesn’t provide any special privileges to access protected features on the host (beyond having extra UIDs and GIDs). Here are examples of container actions you might expect to work from a rootless account that will not work:
  - Anything you want to access from a mounted directory from the host must be accessible by the UID running your container or your request to access that component will fail.
  - There are some system features you won’t be able to change without privilege. For example, you cannot change the system clock by simply setting a SYS_TIME capability inside a container and running the network time service (ntpd). You would have to run that container as root, bypassing your rootless container environment and using the root user’s environment, for that capability to work, such as:

```
$ sudo podman run -d --cap-add SYS_TIME ntpd
```

Note that this example allows ntpd to adjust time for the entire system, and not just within the container.
• A rootless container has no ability to access a port less than 1024. Inside the rootless container’s namespace it can, for example, start a service that exposes port 80 from an httpd service from the container, but it will not be accessible outside of the namespace:

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

However, a container would need root privilege, again using the root user’s container environment, to expose that port to the host system:

```bash
$ sudo podman run -d -p 80:80 httpd
```

• The administrator of a workstation can configure it to allow users to expose services below 1024, but they should understand the security implications. A regular user could, for example, run a web server on the official port 80 and trick external users into believing that it was configured by the administrator. This is generally OK on a workstation, but might not be 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:

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

• Rootless containers currently relies on setting static subuid and subgid ranges. If you are using LDAP or Active Directory to provide user authentication, there is no automated way to provide those UID and GID ranges to users. A current workaround could be to set static ranges in `/etc/subuid` and `/etc/subgid` files to match the known UIDs and GIDs in use.

• Container storage must be on a local file system, because remote file systems do not work well with unprivileged user namespaces.

• An on-going list of shortcomings of running `podman` and related tools without root privilege is contained in `Shortcomings of Rootless Podman`.  

---

CHAPTER 1. STARTING WITH CONTAINERS
CHAPTER 2. WORKING WITH CONTAINER IMAGES

Red Hat Enterprise Linux (RHEL) base images can be used as the foundation for the container images. For RHEL 8, all Red Hat base images are available as new Universal Base Images (UBI), which means that you can freely obtain and redistribute them. These include versions of RHEL standard, minimal, init, and Red Hat Software Collections that are all now freely available and redistributable. The RHEL base 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, so you know you are getting the latest software (see Red Hat Container Image Updates).

- **tracked**: Tracked by errata to help understand the changes that go into each update.

- **Reusable**: The base images need to be downloaded and cached in your production environment once. Each base image can be reused by all containers that include it as their foundation.

UBIs for RHEL 8 provide the same quality RHEL software for building container images as their non-UBI predecessors (**rhel6**, **rhel7**, **rhel-init**, and **rhel-minimal** base images), but offer more freedom in how they are used and distributed.

For RHEL 8, standard, minimal and init base images are available. Red Hat also provides a set of language runtime images, based on Application Streams, that you can build on when you are creating containers for applications that require specific runtimes. Runtime images include python, php, ruby, nodejs, and others.

There is a set of RHEL 7 images as well that you can run on RHEL 8 systems. For RHEL 7, there are both UBI (redistributable) and non-UBI (require subscription access and are non-redistributable) base images. Those images include three regular base images (**rhel7**, **rhel-init**, and **rhel-minimal**) and three UBI images (**ubi7**, **ubi7-init**, and **ubi7-minimal**).

Although Red Hat does not offer tools for running containers on RHEL 6 systems, it does offer RHEL 6 container images you can use. There are standard (**rhel6**) and init (**rhel6-init**) base image available for RHEL 6, but no minimal RHEL 6 image. Likewise, there are no RHEL 6 UBI images.

Although the legacy RHEL 7 base images will continue to be supported, UBI images are recommended going forward. For that reason, examples in the rest of this chapter are done with RHEL 8 UBI images. For a list of available Red Hat UBI images, and associated information about UBI repositories and source code, see article Universal Base Images (UBI): Images, repositories, and packages.

### 2.1. DIFFERENCES BETWEEN RHEL IMAGES AND UBI IMAGES

UBI images were created so you can build your container images on a foundation of official Red Hat software that can be freely shared and deployed. From a technical perspective, they are nearly identical to legacy Red Hat Enterprise Linux images, which means they have great security, performance, and life cycles, but they are released under a different End User License Agreement. Here are some attributes of Red Hat 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. All of the content used to build selected UBI images is released in a publicly available set of yum repositories. This lets you install extra packages, as
well as update any package in UBI base images.

- **Redistributable**: The intent of UBI images is to allow Red Hat customers, partners, ISVs, and others to standardize UBI images so you can build your container images on a foundation of official Red Hat software that can be freely shared and deployed. From a technical perspective, they are nearly identical to legacy Red Hat Enterprise Linux images, which means they have great security, performance, and life cycles, but they are released under a different End User License Agreement.

- **Base and runtime images** Besides the three types of base images, UBI versions of various runtime images are available as well. These runtime images provide a foundation for applications that can benefit from standard, supported runtimes such as python, php, nodejs, and ruby.

- **Enabled yum repositories**: The following yum repositories are enabled within each RHEL 8 UBI image:
  - The `ubi-8-baseos` repository holds the redistributable subset of RHEL packages you can include in your container.
  - The `ubi-8-appstream` repository holds Application streams packages that you can add to a UBI image to help you standardize the environments you use with applications that require particular runtimes.

- **Licensing**: You are free to use and redistribute UBI images, provided you adhere to the Red Hat Universal Base Image End User Licensing Agreement.

- **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 Learn more on details.

### 2.2. UNDERSTANDING STANDARD RED HAT BASE IMAGES

Standard RHEL 8 base images (`ubi8`) have a robust set of software features that include the following:

- **init system**: All the features of the systemd initialization system you need to manage systemd services are available in the standard base images. These init systems let you install RPM packages that are pre-configured to start up services automatically, such as a Web server (`httpd`) or FTP server (`vsftpd`).

- **yum**: Software needed to install software packages is included via the standard set of `yum` commands (`yum, yum-config-manager, yumdownloader, and so on`). For the UBI base images, you have access to free yum repositories for adding and updating software.

- **utilities**: The standard base image includes some useful utilities for working inside the container. Utilities that are in this base image that are not in the minimal images include `tar, dmidecode, gzip, getfacl` (and other acl commands), `dmsetup` (and other device mapper commands), and others.

### 2.3. UNDERSTANDING MINIMAL RED HAT BASE IMAGES

The `ubi8-minimal` images are stripped-down RHEL images to use when a bare-bones base image is desired. If you are looking for the smallest possible base image to use as part of the larger Red Hat ecosystem, you can start with these minimal images.
RHEL minimal images provide a base for your own container images that is less than half the size of the standard image, while still being able to draw on RHEL software repositories and maintain any compliance requirements your software has.

Here are some features of the minimal base images:

- **Small size**: Minimal images are about 92M on disk and 32M compressed. This makes it less than half the size of the standard images.

- **Software installation (microdnf)**: Instead of including the full-blown `yum` facility for working with software repositories and RPM software packages, the minimal images include the `microdnf` utility. The `microdnf` is a scaled-down version of `dnf`. It includes only what is needed to enable and disable repositories, as well as install, remove, and update packages. It also has a clean option, to clean out cache after packages have been installed.

- **Based on RHEL packaging**: Because minimal images incorporate regular RHEL software RPM packages, with a few features removed such as extra language files or documentation, you can continue to rely on RHEL repositories for building your images. This allows you to still maintain compliance requirements you have that are based on RHEL software. Features of minimal images make them perfect for trying out applications you want to run with RHEL, while carrying the smallest possible amount of overhead. What you do not get with minimal images is an initialization and service management system (systemd or System V init), a Python run-time environment, and a bunch of common shell utilities.

- **Modules for microdnf are not supported**: Modules used with the `dnf` command let you install multiple versions of the same software, when available. The `microdnf` utility included with minimal images does not support modules. So if modules are required, you should use a non-minimal base images, which include `yum`.

If your goal, however, is just to try to run some simple binaries or pre-packaged software that does not have a lot of requirements from the operating system, the minimal images might suit your needs. If your application does have dependencies on other software from RHEL, you can use `microdnf` to install the needed packages at build time.

Red Hat intends for you to always use the latest version of the minimal images, which is implied by requesting `ubi8/ubi-minimal` or `ubi8-minimal`. Red Hat does not expect to support older versions of minimal images going forward.

### 2.4. UNDERSTANDING INIT RED HAT BASE IMAGES

The UBI `ubi8-init` images contain the systemd initialization system, making them useful for building images in which you want to run systemd services, such as a web server or file server. The init image contents are less than what you get with the standard images, but more than what is in the minimal images.

**NOTE**

Because the `ubi8-init` image builds on top of the `ubi8` image, their contents are mostly the same. There are a few critical differences, however. In `ubi8-init`, the `Cmd` is set to `/sbin/init`, instead of `bash`, to start the systemd Init service by default. It includes `ps` and process related commands (procpss-ng package), which `ubi8` does not. Also, `ubi8-init` sets `SIGRTMIN+3` as the `StopSignal`, as systemd in `ubi8-init` ignores normal signals to exit (`SIGTERM` and `SIGKILL`), but will terminate if it receives `SIGRTMIN+3`.

Historically, Red Hat Enterprise Linux base container images were designed for Red Hat customers to
run enterprise applications, but were not free to redistribute. This can create challenges for some organizations that need to redistribute their applications. That is where the Red Hat Universal Base Images come in.

### 2.5. Using the UBI Init Images

This procedure shows how to build a container using a Dockerfile 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.

**Procedure**

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

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

   The Dockerfile 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. Optionally, 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.

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
   ```
2.6. REDISTRIBUTING UBI IMAGES

This procedure describes how to redistribute UBI images. After you pull a UBI image, you are free to push a UBI image to your own or another third-party registry and share it with others. You can upgrade or add to that image from UBI yum repositories as you like.

Procedure

1. To pull the `ubi` image from the registry.redhat.io registry, enter:
   ```bash
   # podman pull registry.redhat.io/ubi8/ubi
   ```

2. To add an additional name to the `ubi` image, enter:
   ```bash
   # podman tag registry.redhat.io/ubi8/ubi registry.example.com:5000/ubi8/ubi
   ```

3. To push the `ubi` image from your local storage to a registry, enter:
   ```bash
   # podman push registry.example.com:5000/ubi8/ubi
   ```

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.

2.7. SEARCHING FOR CONTAINER IMAGES

The `podman search` command lets you search selected container registries for images.

**NOTE**

You can also search for images in the Red Hat Container Registry. The Red Hat Container Registry includes the image description, contents, health index, and other information.

You can find the list of registries in the configuration file `registries.conf`:

```plaintext
[registries.search]
registries = ['registry.access.redhat.com', 'registry.redhat.io', 'docker.io']

[registries.insecure]
registries = []

[registries.block]
registries = []
```

- By default, the `podman search` command searches for container images from registries listed in section `[registries.search]` in the given order. In this case, `podman search` command looks for the requested image in registry.access.redhat.com, registry.redhat.io and docker.io in this order.
- The `[registries.insecure]` section adds the registries that do not use TLS (an insecure registry).
• The \texttt{[registries.block]} section disallows the access to the registry from your local system.

As a root user, you can edit the \texttt{/etc/containers/registries.conf} file to change the default, system-wide search settings.

As a regular (rootless) user of \texttt{podman}, you can create your own \texttt{registries.conf} file in your home directory (\$\texttt{HOME/.config/containers/registries.conf}) to override the system-wide settings.

Make sure that you follow the conditions when configuring container registries:

• Each registry must be surrounded by single quotes.

• If there are multiple registries set for the \texttt{registries = value}, you must separate those registries by commas.

• You can identify registries by either IP address or hostname.

• If the registry uses a non-standard port - other than TCP ports 443 for secure and 80 for insecure, enter that port number with the registry name. For example: host.example.com:9999.

• The system searches for registries in the order in which they appear in the \texttt{registries.search} list of the \texttt{registries.conf} file.

Some \texttt{podman search} command examples follow. The first example illustrates the unsuccessful search of all images from quay.io. The forward slash at the end means to search the whole registry for all images accessible to you:

\begin{verbatim}
# podman search quay.io/
ERRO[0000] error searching registry "quay.io": couldn't search registry "quay.io":
unable to retrieve auth token: invalid username/password
\end{verbatim}

To search quay.io registry, log in first:

\begin{verbatim}
# podman login quay.io
Username: johndoe
Password: ***********
Login Succeeded!
# podman search quay.io/
\end{verbatim}

Search all available registries for \texttt{postgresql} images (resulting in more than 40 images found):

\begin{verbatim}
# podman search postgresql-10
\end{verbatim}
To limit your search for `postgresql` to images from registry.redhat.io, type the following command. Notice that by entering the registry and the image name, any repository in the registry can be matched:

```
# podman search registry.redhat.io/postgresql-10
```

<table>
<thead>
<tr>
<th>INDEX</th>
<th>NAME</th>
<th>DESCRIPTION</th>
<th>STARS</th>
<th>OFFICIAL</th>
<th>AUTOMATED</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>registry.redhat.io/rhel8/postgresql-10</td>
<td>This container image ...</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>registry.redhat.io/rhscl/postgresql-10-rhel7</td>
<td>PostgreSQL is an</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To get longer descriptions for each container image, add `--no-trunc` to the command:

```
# podman search --no-trunc registry.redhat.io/rhel8/postgresql-10
```

<table>
<thead>
<tr>
<th>INDEX</th>
<th>NAME</th>
<th>DESCRIPTION</th>
<th>STARS</th>
<th>OFFICIAL</th>
<th>AUTOMATED</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>This container image provides a containerized ...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To access insecure registries, add the fully-qualified name of the registry to the `[registries.insecure]` section of the `/etc/containers/registries.conf` file. For example:

```
[registries.search]
registries = ['myregistry.example.com']

[registries.insecure]
registries = ['myregistry.example.com']
```

Then, search for `myimage` images:

```
# podman search myregistry.example.com/myimage
```

<table>
<thead>
<tr>
<th>INDEX</th>
<th>NAME</th>
<th>DESCRIPTION</th>
<th>STARS</th>
<th>OFFICIAL</th>
<th>AUTOMATED</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>The myimage container executes the ...</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Now you can pull `myimage` image:

```
# podman pull myimage.example.com/myimage
```

## 2.8. DEFINING THE IMAGE SIGNATURE VERIFICATION POLICY

Red Hat delivers signatures for the images in the Red Hat Container Registry. When running as root, `/etc/containers/policy.json`, and the YAML files in the `/etc/containers/registries.d/` directory define the signature verification policy. The trust policy in `/etc/containers/policy.json` describes a registry scope (registry and or or repository) for the trust.

By default, the container tool reads the policy from `$HOME/.config/containers/policy.json`, if it exists, otherwise from `/etc/containers/policy.json`.

Trust is defined using three parameters:
1. The registry or registry/repository name

2. One or more public GPG keys

3. A signature server

Red Hat serves signatures from these URIs:

https://access.redhat.com/webassets/docker/content/sigstore
https://registry.redhat.io/containers/sigstore

**Procedure**

1. Display the `/etc/containers/policy.json` file:

```bash
# cat /etc/containers/policy.json
{
   "default": [ 
     { 
       "type": "insecureAcceptAnything"
     }
   ],
   "transports": 
   { 
     "docker-daemon": 
     { 
       "": [{"type":"insecureAcceptAnything"]}
     }
   }
}
```

2. To update an existing trust scope for the registries registry.access.redhat.com and registry.redhat.io, enter:

```bash
# podman image trust set -f /etc/pki/rpm-gpg/RPM-GPG-KEY-redhat-release registry.access.redhat.com
# podman image trust set -f /etc/pki/rpm-gpg/RPM-GPG-KEY-redhat-release registry.redhat.io
```

3. To verify trust policy configuration, display the `/etc/containers/policy.json` file:

```json
"docker": { 
    "registry.access.redhat.com": [ 
    { 
      "type": "signedBy", 
      "keyType": "GPGKeys", 
      "keyPath": "/etc/pki/rpm-gpg/RPM-GPG-KEY-redhat-release"
    }
    ],
    "registry.redhat.io": [ 
    { 
      "type": "signedBy", 
      "keyType": "GPGKeys", 
      "keyPath": "/etc/pki/rpm-gpg/RPM-GPG-KEY-redhat-release"
    }
    ]
}
```
4. Create the `/etc/containers/registries.d/registry.access.redhat.com.yaml` file to identify the signature store for container images from registry.access.redhat.com registry:

```yaml
docker:
  registry.access.redhat.com:
    sigstore: https://access.redhat.com/webassets/docker/content/sigstore
```

5. Create the `/etc/containers/registries.d/registry.redhat.io.yaml` file with the following content:

```yaml
docker:
  registry.redhat.io:
    sigstore: https://registry.redhat.io/containers/sigstore
```

6. To display the trust configuration, enter:

```bash
# podman image trust show
```

```
default                     accept
registry.access.redhat.com  signedBy                security@redhat.com, security@redhat.com
https://access.redhat.com/webassets/docker/content/sigstore
registry.redhat.io          signedBy                security@redhat.com, security@redhat.com
https://registry.redhat.io/containers/sigstore
insecureAcceptAnything
```

7. To reject the default trust policy, type:

```bash
# podman image trust set -t reject default
```

8. To verify the trust policy configuration, display the `/etc/containers/policy.json`:

```bash
# cat /etc/containers/policy.json
```

```
{
  "default": [
    {
      "type": "reject"
    }
  ]
}
```

You can see that the "default" section has changed from "insecureAcceptAnything" to "reject".

9. Try to pull the minimal Red Hat Universal Base Image 8 (`ubi8-minimal`) image from the registry.access.redhat.com registry:

```bash
# podman --log-level=debug pull registry.access.redhat.com/ubi8-minimal
```

```
DEBU[0000] Using registries.d directory /etc/containers/registries.d for sigstore configuration
```
10. Log in to the registry.redhat.io registry:

   # podman login registry.redhat.io
   Username: username
   Password: ***********
   Login Succeeded!

11. Try to pull the support-tools image from the registry.redhat.io registry:

   # podman --log-level=debug pull registry.redhat.io/rhel8/support-tools
   ...
   DEBU[0000] Using registries.d directory /etc/containers/registries.d for sigstore configuration
   DEBU[0000] Using "docker" namespace registry.redhat.io
   DEBU[0000] Using https://registry.redhat.io/containers/sigstore
   ...

   You can see that the signature storage address registry.redhat.io/containers/sigstore
   matches the address you specified in the /etc/containers/registries.d/registry.redhat.io.yaml.

12. To list all images pulled to your local system, enter:

   # podman images

   REPOSITORY                              TAG     IMAGE ID      CREATED       SIZE
   registry.redhat.io/rhel8/support-tools   latest  5ef2aab09451  13 days ago   254 MB
   registry.access.redhat.com/ubi8-minimal  latest  86c870596572  13 days ago   146 MB

Additional resources

- For more information on the podman image trust, type man podman-image-trust.
- For more information about verifying container images, see article Verifying image signing for Red Hat Container Registry.

2.9. PULLING IMAGES FROM REGISTRIES

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>

The <registry> is a host that provides a container registry service on TCP <port>. Together, <namespace> and <name> identify a particular image controlled by <namespace> at that registry. The <tag> is an additional name to locally-stored image, the default tag is latest. Always 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 which do not allow unknown or anonymous users to create accounts with arbitrary names.
Some registries also support raw `<name>`; for those, `<namespace>` is optional. When it is included, however, the additional level of hierarchy that `<namespace>` provides is useful to distinguish between images with the same `<name>`. For example:

<table>
<thead>
<tr>
<th>Namespace</th>
<th>Examples (<code>&lt;namespace&gt;/&lt;name&gt;</code>)</th>
</tr>
</thead>
<tbody>
<tr>
<td>organization</td>
<td>redhat/kubernetes, google/kubernetes</td>
</tr>
<tr>
<td>login (user name)</td>
<td>alice/application, bob/application</td>
</tr>
<tr>
<td>role</td>
<td>devel/database, test/database, prod/database</td>
</tr>
</tbody>
</table>

The registries that Red Hat provides are registry.redhat.io (requiring authentication), registry.access.redhat.com (requires no authentication), and registry.connect.redhat.com (holds Red Hat Partner Connect program images). 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. For example:

```bash
# podman login registry.redhat.io
Username: myusername
Password: ************
Login Succeeded!
```

Use the pull option to pull an image from a remote registry. To pull the RHEL base image `ubi` and `rsyslog` logging image from the Red Hat registry, type:

```bash
# podman pull registry.redhat.io/ubi8/ubi
# podman pull registry.redhat.io/rhel8/rsyslog
```

An image is identified by a registry name (registry.redhat.io), a namespace name (ubi8) and the image name (ubi). You could also add a tag (which defaults to :latest if not entered). The repository name `ubi`, when passed to the `podman pull` command without the name of a registry preceding it, is ambiguous and could result in the retrieval of an image that originates from an untrusted registry. If there are multiple versions of the same image, adding a tag, such as `latest` to form a name such as `ubi8/ubi:latest`, lets you choose the image more explicitly.

To see the images that resulted from the above `podman pull` command, along with any other images on your system, type `podman images`:

<table>
<thead>
<tr>
<th>REPOSITORY</th>
<th>TAG</th>
<th>IMAGE ID</th>
<th>CREATED</th>
<th>SIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>registry.redhat.io/ubi8/ubi</td>
<td>latest</td>
<td>eb205f07ce7d</td>
<td>2 weeks ago</td>
<td>214MB</td>
</tr>
<tr>
<td>registry.redhat.io/rhel8/rsyslog</td>
<td>latest</td>
<td>85cfba5cd49c</td>
<td>2 weeks ago</td>
<td>234MB</td>
</tr>
</tbody>
</table>

The `ubi` and `rsyslog` images are now available on your local system for you to work with.

### 2.10. Listing Images

You can use `podman images` command to see which images have been pulled to your local system and are available to use.

**Procedure**
To list images in local storage, enter:

```
# podman images
```

<table>
<thead>
<tr>
<th>REPOSITORY</th>
<th>TAG</th>
<th>IMAGE ID</th>
<th>CREATED</th>
<th>VIRTUAL SIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>registry.redhat.io/rhel8/support-tools</td>
<td>latest</td>
<td>b3d6ce4e0043</td>
<td>2 days ago</td>
<td>234MB</td>
</tr>
<tr>
<td>registry.redhat.io/ubi8/ubi-init</td>
<td>latest</td>
<td>779a05997856</td>
<td>2 days ago</td>
<td>225MB</td>
</tr>
<tr>
<td>registry.redhat.io/ubi8/ubi</td>
<td>latest</td>
<td>a80dad1c1953</td>
<td>3 days ago</td>
<td>210MB</td>
</tr>
</tbody>
</table>

### 2.11. INSPECTING LOCAL IMAGES

After you pull an image to your local system and before you run it, it is a good idea to investigate that image. Reasons for investigating an image before you run it include:

- Understanding what the image does
- Checking what software is inside the image

#### Procedure

The `podman inspect` command displays basic information about what an image does. You also have the option of mounting the image to your host system and using tools from the host to investigate what is in the image. Here is an example of investigating what a container image does before you run it.

1. **Inspect an image** Run `podman inspect` to see what command is executed when you run the container image, as well as other information. Here are examples of examining the ubi8/ubi and rhel8/rsyslog container images (with only snippets of information shown here):

   ```
   # podman pull registry.redhat.io/ubi8/ubi
   # podman inspect registry.redhat.io/ubi8/ubi | less
   ...
   "Cmd": [ 
   "/bin/bash"
   ],
   "Labels": {
   "License": "GPLv3",
   "architecture": "x86_64",
   "authoritative-source-url": "registry.redhat.io",
   "build-date": "2018-10-24T16:46:08.916139",
   "com.redhat.build-host": "cpt-0009.osbs.prod.upshift.rdu2.redhat.com",
   "com.redhat.component": "rhel-server-container",
   "description": "The Red Hat Enterprise Linux Base image is designed to be a fully supported..."
   ...
   
   # podman pull registry.redhat.io/rhel8/rsyslog
   # podman inspect registry.redhat.io/rhel8/rsyslog
   "Cmd": [ 
   "/bin/rsyslog.sh"
   ],
   "Labels": {
   "License": "GPLv3",
   "architecture": "x86_64",
   ...
   "install": "podman run --rm --privileged -v /:/host -e HOST=/host -e IMAGE=IMAGE -e NAME=NAME IMAGE /bin/install.sh",
   ```
The ubi8/ubi container will execute the bash shell, if no other argument is given when you start it with `podman run`. If an Entrypoint were set, its value would be used instead of the Cmd value (and the value of Cmd would be used as an argument to the Entrypoint command).

In the second example, the rhel8/rsyslog container image has built-in `install` and `run` labels. Those labels give an indication of how the container is meant to be set up on the system (`install`) and executed (`run`).

2. **Mount a container:** Using the `podman` command, mount an active container to further investigate its contents. This example runs and lists a running `rsyslog` container, then displays the mount point from which you can examine the contents of its file system:

```
# podman run -d registry.redhat.io/rhel8/rsyslog
# podman ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
1cc92aea398d ...rsyslog:latest /bin/rsyslog.sh 37 minutes ago Up 1 day ago myrsyslog
# podman mount 1cc92aea398d /var/lib/containers/storage/overlay/65881e78.../merged
# ls /var/lib/containers/storage/overlay/65881e78*/merged
bin boot dev etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr
var
```

After the `podman mount`, the contents of the container are accessible from the listed directory on the host. Use `ls` to explore the contents of the image.

3. **Check the image’s package list** To check the packages installed in the container, tell the `rpm` command to examine the packages installed on the container’s mount point:

```
# rpm -qa --root=/var/lib/containers/storage/overlay/65881e78.../merged
redhat-release-server-7.6-4.el7.x86_64
filesystem-3.2-25.el7.x86_64
basesystem-10.0-7.el7.noarch
ncurses-base-5.9-14.20130511.el7_4.noarch
glibc-common-2.17-260.el7.x86_64
nspr-4.19.0-1.el7_5.x86_64
libstdc++-4.8.5-36.el7.x86_64
```

### 2.12. INSPECTING REMOTE IMAGES

To inspect a container image before you pull it to your system, you can use the `skopeo inspect` command. With `skopeo inspect`, you can display information about an image that resides in a remote container registry.

**Procedure**
The following command inspects the **ubi8-init** image from the Red Hat registry.

- To inspect the **ubi8-init** from the registry.redhat.io registry, enter:

  ```bash
  # skopeo inspect docker://registry.redhat.io/ubi8/ubi-init
  {
  "Name": "registry.redhat.io/ubi8/ubi-init",
  "Digest": "sha256:53dfe24...",
  "RepoTags": [
    "8.0.0-9",
    "8.0.0",
    "latest"
  ],
  "DockerVersion": "1.13.1",
  "Labels": {
    "architecture": "x86_64",
    "authoritative-source-url": "registry.access.redhat.com",
    "build-date": "2019-05-13T20:49:44.207967",
    "com.redhat.build-host": "cpt-0013.osbs.prod.upshift.rdu2.redhat.com",
    "com.redhat.component": "ubi8-init-container",
    "description": "The Red Hat Enterprise Linux Init image is designed to be..."
  }
  ```

### 2.13. TAGGING IMAGES

You can add names to images to make it more intuitive to understand what they contain. Tagging images can also be used to identify the target registry for which the image is intended. Using the `podman tag` command, you essentially add an alias to the image that can consist of several parts. Those parts can include:

```
registryhost/username/NAME:tag
```

You can add just NAME if you like. For example:

```bash
# podman tag 474ff279782b myrhel8
```

In the previous example, the **rhel8** image had an image ID of 474ff279782b. Using `podman tag`, the name **myrhel8** now also is attached to the image ID. So you could run this container by name (rhel8 or myrhel8) or by image ID. Notice that without adding a :tag to the name, it was assigned :latest as the tag. You could have set the tag to 8.0 as follows:

```bash
# podman tag 474ff279782b myrhel8:8.0
```

To the beginning of the name, you can optionally add a user name and/or a registry name. The user name is actually the repository on Docker.io that relates to the user account that owns the repository. Tagging an image with a registry name was shown in the “Tagging Images” section earlier in this document. Here’s an example of adding a user name:

```bash
# podman tag 474ff279782b jsmith/myrhel8
# podman images | grep 474ff279782b
rhel8     latest  474ff279782b  7 days ago  139.6 MB
myrhel8   latest  474ff279782b  7 months ago 139.6 MB
myrhel8   7.1     474ff279782b  7 months ago  139.6 MB
jsmith/myrhel8 latest  474ff279782b  7 months ago 139.6 MB
```
Above, you can see all the image names assigned to the single image ID.

2.14. SAVING AND LOADING IMAGES

If you want to save a container image you have stored locally, you can use **podman save** to save the image to an archive file or directory and restore it later to another container environment. The archive you save can be in any of several different container image formats: docker-archive, oci-archive, oci-dir (directory with oci manifest type), or docker-dir (directory with v2s2 manifest type). After you save an image, you can store it or send it to someone else, then **load** the image later to reuse it. Here is an example of saving an image as a tarball in the default docker-archive format:

```
# podman save -o myrsyslog.tar registry.redhat.io/rhel8/rsyslog:latest
# file myrsyslog.tar
myrsyslog.tar: POSIX tar archive
```

The **myrsyslog.tar** file is now stored in your current directory. Later, when you are ready to reuse the tarball as a container image, you can import it to another podman environment as follows:

```
# podman load -i myrsyslog.tar
# podman images
REPOSITORY                       TAG    IMAGE ID      CREATED     SIZE
registry.redhat.io/rhel8/rsyslog latest 1f5313131bf0  7 weeks ago 235 MB
```

Instead of using **save** and **load** to store and reload an image, you can make a copy of a container instead, using **podman export** and **podman import**.

2.15. REMOVING IMAGES

To see a list of images that are on your system, run the **podman images** command. To remove images you no longer need, use the **podman rmi** command, with the image ID or name as an option. (You must stop any containers run from an image before you can remove the image.) Here is an example:

```
# podman rmi ubi8-init
7e85c3f4126351cc9d24e492488ba7e49820be08fe53bee02301226f2773293
```

You can remove multiple images on the same command line:

```
# podman rmi registry.redhat.io/rhel8/rsyslog support-tools
46da8e23fa1461b658f9276191b4f1473f366759a6c840f05ed0c9f694aa7c2f
85cfa5c49c84786c773a9f66b8d6fca04582d57b921a308f04bb8ec071205
```

If you want to clear out all your images, you could use a command like the following to remove all images from your local registry (make sure you mean it before you do this!):

```
# podman rmi -a
1ca061b47bd70141d11dcb2272dee0f9ea3f76e9af71dcd121a000f3f5423731
ed904b8f2d5c1b5502dea190977e066b4f76776b98f6d5aa1e389256d5212993
83508706ef1b603e511b1b19afcb5faab565053559942db5d00415fb1ee21e96
```

To remove images that have multiple names (tags) associated with them, you need to add the force option to remove them. For example:

```
# podman rmi -a
```
A container associated with containers/storage, i.e. via Buildah, CRI-O, etc., may be associated with this image: 1de7d7b3f531

# podman rmi -f 1de7d7b3f531
1de7d7b3f531...
CHAPTER 3. WORKING WITH CONTAINERS AND PODS

Containers represent a running or stopped process spawned from the files located in a decompressed container image. Tools for running containers and working with them are described in this section.

3.1. RUNNING CONTAINERS

When you execute a podman run command, you essentially spin up and create a new container from a container image. The command you pass to the podman run command line sees the inside of the container as its running environment so, by default, very little can be seen of the host system. For example, by default, the running application sees:

- The file system provided by the container image.
- A new process table from inside the container (no processes from the host can be seen).

If you want to make a directory from the host available to the container, map network ports from the container to the host, limit the amount of memory the container can use, or expand the CPU shares available to the container, you can do those things from the podman run command line. Here are some examples of podman run command lines that enable different features.

EXAMPLE #1 (Run a quick command): This podman command runs the cat /etc/os-release command to see the type of operating system used as the basis for the container. After the container runs the command, the container exits and is deleted (--rm).

```bash
# podman run --rm registry.redhat.io/ubi8/ubi cat /etc/os-release
NAME="Red Hat Enterprise Linux"
VERSION="8.2 (Ootpa)"
ID="rhel"
ID_LIKE="fedora"
VERSION_ID="8.2"
PLATFORM_ID="platform:el8"
PRETTY_NAME="Red Hat Enterprise Linux 8.2 (Ootpa)"
ANSI_COLOR="0;31"
CPE_NAME="cpe:/o:redhat:enterprise_linux:8.2:GA"
HOME_URL="https://www.redhat.com/"
BUG_REPORT_URL="https://bugzilla.redhat.com/"

REDHAT_BUGZILLA_PRODUCT="Red Hat Enterprise Linux 8"
REDHAT_BUGZILLA_PRODUCT_VERSION=8.2
REDHAT_SUPPORT_PRODUCT="Red Hat Enterprise Linux"
REDHAT_SUPPORT_PRODUCT_VERSION="8.2"
...
```

EXAMPLE #2 (View the Dockerfile in the container): This is another example of running a quick command to inspect the content of a container from the host. All layered images that Red Hat provides include the Dockerfile from which they are built in /root/buildinfo. In this case you do not need to mount any volumes from the host.

```bash
podman run --rm \n    registry.redhat.io/rhel8/rsyslog \n    ls /root/buildinfo
Dockerfile-rhel8-rsyslog-8.2-25
```

Now you know what the Dockerfile is called, you can list its contents:
EXAMPLE #3 (Run a shell inside the container) Using a container to launch a bash shell lets you look inside the container and change the contents. This sets the name of the container to mybash. The -i creates an interactive session and -t opens a terminal session. Without -i, the shell would open and then exit. Without -t, the shell would stay open, but you would not be able to type anything to the shell.

Once you run the command, you are presented with a shell prompt and you can start running commands from inside the container:

```
# podman run --name=mybash -it registry.redhat.io/ubi8/ubi /bin/bash
[root@ed904b8f2d5c/]# yum install procps-ng
[root@ed904b8f2d5c/]# ps -ef
UID      PID  PPID  C STIME TTY          TIME CMD
root          1     0  0 00:46 pts/0    00:00:00 /bin/bash
root         35     1  0 00:51 pts/0    00:00:00 ps -ef
[root@49830c4f9cc4/]# exit
```

Although the container is no longer running once you exit, the container still exists with the new software package still installed. Use `podman ps -a` to list the container:

```
# podman ps -a
CONTAINER ID   IMAGE                   COMMAND   CREATED       STATUS        PORTS NAMES
IS INFRA       1ca061b47bd7 .../ubi8/ubi:latest   /bin/bash 8 minutes ago Exited 12 seconds ago musing_brown false
...
```

You could start that container again using `podman start` with the -ai options. For example:

```
# podman start -ai mybash
[root@ed904b8f2d5c/]#
```

EXAMPLE #4 (Bind mounting log files) One way to make log messages from inside a container available to the host system is to bind mount the host /dev/log device inside the container. This example illustrates how to run an application in a RHEL container that is named `log_test` that generates
log messages (just the logger command in this case) and directs those messages to the /dev/log device that is mounted in the container from the host. The --rm option removes the container after it runs.

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

EXAMPLE #5 (Run a service as a daemon with a static IP address) The following example runs an rsyslog service as a daemon process, so it runs continuously in the background. It also tells podman to set the container network interface to a particular IP address (for example, 10.88.0.44). After that, you can run podman inspect command to check that the IP address was set properly:

```bash
# podman run -d --ip=10.88.0.44 registry.access.redhat.com/rhel7/rsyslog
```

3.2. INVESTIGATING RUNNING AND STOPPED CONTAINERS

After you have some running containers, you can list both those containers that are still running and those that have exited or stopped with the podman ps command. You can also use the podman inspect to look at specific pieces of information within those containers.

3.2.1. Listing containers

Let us say you have one or more containers running on your host. To work with containers from the host system, you can open a shell and try some of the following commands.

**podman ps:** The ps option shows all containers that are currently running:

```bash
# podman run -d registry.redhat.io/rhel8/rsyslog
# podman ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
74b1da000a11 rhel8/rsyslog /bin/rsyslog.sh 2 minutes ago Up About a minute musing_brown
```

If there are containers that are not running, but were not removed (--rm option), the containers are present and can be restarted. The podman ps -a command shows all containers, running or stopped.

```bash
# podman ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES IS INFRA
74b1da000a11 rhel8/rsyslog /bin/rsyslog.sh 2 mins ago Up About a minute musing_brown false
```

3.2.2. Inspecting containers

To inspect the metadata of an existing container, use the podman inspect command. You can show all metadata or just selected metadata for the container. For example, to show all metadata for a selected container, type:

```bash
# podman inspect 74b1da000a11
```
You can also use inspect to pull out particular pieces of information from a container. The information is stored in a hierarchy. So to see the container IP address (IPAddress under NetworkSettings), use the --format option and the identity of the container. For example:

```bash
# podman inspect --format='{{.NetworkSettings.IPAddress}}' 74b1da000a11
10.88.0.31
```

Examples of other pieces of information you might want to inspect include .Path (to see the command run with the container), .Args (arguments to the command), .ConfigExposedPorts (TCP or UDP ports exposed from the container), .State.Pid (to see the process id of the container) and .HostConfig.PortBindings (port mapping from container to host). Here is an example of .State.Pid and .State.StartedAt:

```bash
# podman inspect --format='{{.State.Pid}}' 74b1da000a11
19593
# ps -ef | grep 19593
root 19593 19583  0 10:30 ?        00:00:00 /usr/sbin/rsyslogd -n
# podman inspect --format='{{.State.StartedAt}}' 74b1da000a11
2018-11-13 10:30:35.358175255 -0500 EST
```

In the first example, you can see the process ID of the containerized executable on the host system (PID 19593). The `ps -ef` command confirms that it is the `rsyslogd` daemon running. The second example shows the date and time that the container was run.

### 3.2.3. Investigating within a container

To investigate within a running container, you can use the `podman exec` command. With `podman exec`, you can run a command (such as `/bin/bash`) to enter a running container process to investigate that container.

The reason for using `podman exec`, instead of just launching the container into a bash shell, is that you can investigate the container as it is running its intended application. By attaching to the container as it is performing its intended task, you get a better view of what the container actually does, without necessarily interrupting the container’s activity.

Here is an example using `podman exec` to look into a running `rsyslog`, then look around inside that container.

1. **Launch a container:** Launch a container such the `rsyslog` container image described earlier. Type `podman ps` to make sure it is running:

   ```bash
   # podman ps
   ```
2. Enter the container with `podman exec`: Use the container ID or name to open a bash shell to access the running container. Then you can investigate the attributes of the container as follows:

```
# podman exec -it 74b1da000a11 /bin/bash
[root@74b1da000a11 ]# cat /etc/redhat-release
Red Hat Enterprise Linux release 8.0
[root@74b1da000a11 ]# yum install procps-ng
[root@74b1da000a11 ]# ps -ef
UID   PID  PPID  C STIME TTY          TIME CMD
root   1     0  0 15:30 ?        00:00:00 /usr/sbin/rsyslogd -n
root   8     0  6 16:01 pts/0    00:00:00 /bin/bash
root  21     8  0 16:01 pts/0    00:00:00 ps -ef
[root@74b1da000a11 ]# df -h
Filesystem   Size  Used Avail Use% Mounted on
overlay       39G  2.5G   37G   7% /
 tmpfs         64M     0   64M   0% /dev
 tmpfs         1.5G  8.7M  1.5G   1% /etc/hosts
 shm           63M     0   63M   0% /dev/shm
 tmpfs         1.5G  1.5G  0% /sys/fs/cgroup
 tmpfs         1.5G  1.5G  0% /proc/acpi
 tmpfs         1.5G  1.5G  0% /proc/scsi
 tmpfs         1.5G  1.5G  0% /sys/firmware
[root@74b1da000a11 ]# uname -r
4.18.0-80.1.2.el8_0.x86_64
[root@74b1da000a11 ]# rpm -qa | more
redhat-release-8.0-0.44.el8.x86_64
filesystem-3.8-2.el8.x86_64
basesystem-11-5.el8.noarch
ncurses-base-6.1-7.20180224.el8.noarch
...
bash-4.2# free -m
 total used free shared buff/cache available
 Mem: 1941  560 139 10 1241 1189
 Swap: 1023  15 1008
[root@74b1da000a11 ]# exit
```

The commands just run from the bash shell (running inside the container) show you several things.

- The container was built from a RHEL release 8.0 image.
- The process table (`ps -ef`) shows that the `/usr/sbin/rsyslogd` command is process ID 1.
- Processes running in the host process table cannot be seen from within the container. Although the `rsyslogd` process can be seen on the host process table (it was process ID 19593 on the host).
- There is no separate kernel running in the container (`uname -r` shows the host system’s kernel).
- The `rpm -qa` command lets you see the RPM packages that are included inside the container. In other words, there is an RPM database inside of the container.
• Viewing memory (\texttt{free -m}) shows the available memory on the host (although what the container can actually use can be limited using cgroups).

### 3.3. STARTING AND STOPPING CONTAINERS

If you ran a container, but did not remove it (\texttt{--rm}), that container is stored on your local system and ready to run again. To start a previously run container that was not removed, use the \texttt{start} option. To stop a running container, use the \texttt{stop} option.

#### 3.3.1. Starting containers

A container that does not need to run interactively can sometimes be restarted after being stopped with only the \texttt{start} option and the container ID or name. For example:

```
# podman start myrhel_httpd
myrhel_httpd
```

To start a container so you can work with it from the local shell, use the \texttt{-a} (attach) and \texttt{-i} (interactive) options. Once the bash shell starts, run the commands you want inside the container and type exit to kill the shell and stop the container.

```
# podman start -a -i agitated_hopper
[root@d65aecc325a4 ~]# exit
```

#### 3.3.2. Stopping containers

To stop a running container that is not attached to a terminal session, use the \texttt{stop} option and the container ID or number. For example:

```
# podman stop 74b1da000a11
74b1da000a114015886c557deec8bed9dfb80c888097aa83f30ca4074ff55fb2
```

The \texttt{stop} option sends a \texttt{SIGTERM} signal to terminate a running container. If the container does not stop after a grace period (10 seconds by default), \texttt{podman} sends a \texttt{SIGKILL} signal. You could also use the \texttt{podman kill} command to kill a container (\texttt{SIGKILL}) or send a different signal to a container. Here is an example of sending a \texttt{SIGHUP} signal to a container (if supported by the application, a \texttt{SIGHUP} causes the application to re-read its configuration files):

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

### 3.4. 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.

**Procedure**

1. To create a volume, enter:

   ```bash
   $ podman volume create hostvolume
   ```

2. To display information about the volume, enter:

   ```bash
   $ podman volume inspect hostvolume
   ```

   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}})`.

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

   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 is stored in the `mntPoint` variable:

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

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

   ```bash
   $ 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:

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

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

   ```bash
   # 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`. Steps 10 - 13 are analogous to steps 5 - 8.

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

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

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

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

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

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

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

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

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

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

15. To stop and remove both containers, enter:

    ```
    $ podman stop myubi1
    $ podman stop myubi2
    $ podman rm myubi1
    $ podman rm myubi2
    ```

16. To remove the host volume, enter:

    ```
    $ podman volume rm hostvolume
    ```

17. To check that you deleted the volume, enter:
$ podman volume ls

Additional resources

- For more information on the `podman volume` command, type `man podman-volume`.

3.5. REMOVING CONTAINERS

To see a list of containers that are still hanging around your system, run the `podman ps -a` command. To remove containers you no longer need, use the `podman rm` command, with the container ID or name as an option. You should stop any containers that are still running before removing them. Here is an example:

```
# podman rm goofy_wozniak
```

You can remove multiple containers on the same command line:

```
# podman rm clever_yonath furious_shockley drunk_newton
```

If you want to clear out all your containers, you could use a command like the following to remove all containers (not images) from your local system (make sure you mean it before you do this!):

```
# podman rm -a
```

3.6. CREATING PODS

Containers are the smallest unit that you can manage with Podman, Skopeo and Buildah container tools. A Podman pod is a group of one or more containers. The Pod concept was introduced by Kubernetes. Podman pods are similar to the Kubernetes definition. Pods are the smallest compute units that you can create, deploy, and manage in OpenShift or Kubernetes environments. Every Podman pod includes an infra container. This container holds the namespaces associated with the pod and allows Podman to connect other containers to the pod. It allows you to start and stop containers within the pod and the pod will stay running. The default infra container is based on the Kubernetes `k8s.gcr.io/pause` image.

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

**Procedure**

1. Create an empty pod:

   ```
   $ podman pod create --name mypod
   223df6b390b4ea87a090a4b52017b9b003187a6960bd37631ae9bc12c433aff
   The pod is in the initial state Created.
   ```

   The pod is in the initial state Created.

2. List all pods:
$ podman pod ps

<table>
<thead>
<tr>
<th>POD ID</th>
<th>NAME</th>
<th>STATUS</th>
<th>CREATED</th>
<th># OF CONTAINERS</th>
<th>INFRA ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>223df6b390b4</td>
<td>mypod</td>
<td>Created</td>
<td>Less than a second ago</td>
<td>1</td>
<td>3afdc93de3e</td>
</tr>
</tbody>
</table>

Notice that the pod has one container in it.

3. List all pods and containers associated with them:

$ podman ps -a --pod

<table>
<thead>
<tr>
<th>CONTAINER ID</th>
<th>IMAGE</th>
<th>COMMAND</th>
<th>CREATED</th>
<th>STATUS</th>
<th>PORTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>3afdc93de3e</td>
<td>k8s.gcr.io/pause:3.1</td>
<td>Less than a second ago</td>
<td>Created</td>
<td></td>
<td></td>
</tr>
<tr>
<td>223df6b390b4</td>
<td>223df6b390b4-infra</td>
<td>223df6b390b4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 `k8s.gcr.io/pause` image.

4. To run a container named myubi in the existing pod named mypod, type:

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

5df5c48feea87f60c75822ceab8370548b04c78be9fc156570949013863ccf71

5. List all pods:

$ podman pod ps

<table>
<thead>
<tr>
<th>POD ID</th>
<th>NAME</th>
<th>STATUS</th>
<th>CREATED</th>
<th># OF CONTAINERS</th>
<th>INFRA ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>223df6b390b4</td>
<td>mypod</td>
<td>Running</td>
<td>Less than a second ago</td>
<td>2</td>
<td>3afdc93de3e</td>
</tr>
</tbody>
</table>

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

6. List all pods and containers associated with them:

$ podman ps -a --pod

<table>
<thead>
<tr>
<th>CONTAINER ID</th>
<th>IMAGE</th>
<th>COMMAND</th>
<th>CREATED</th>
<th>STATUS</th>
<th>PORTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>5df5c48feea87</td>
<td>registry.access.redhat.com/ubi8/ubi:latest</td>
<td>/bin/bash</td>
<td>Less than a second ago</td>
<td>Up</td>
<td></td>
</tr>
<tr>
<td></td>
<td>myubi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3afdc93de3e</td>
<td>k8s.gcr.io/pause:3.1</td>
<td>Less than a second ago</td>
<td>Up Less than a second ago</td>
<td></td>
<td></td>
</tr>
<tr>
<td>223df6b390b4</td>
<td>223df6b390b4-infra</td>
<td>223df6b390b4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Additional resources

- For more information on the `podman pod create` command, type `man podman-pod-create`.
- For more information about pods, see article [Podman: Managing pods and containers in a local container runtime](#) by Brent Baude.

### 3.7. DISPLAYING POD INFORMATION

This section provides information on how to display pod information.

Prerequisites
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
    a9f807faacdc frosty_hodgkin    --      3.092MB / 16.7GB    0.02%   -- / --   -- / --    2
    3b33001239ee sleepy_stallman   --      -- / --             --      -- / --   -- / --    --
    ```

  - To display information describing the pod, enter:
    
    ```
    $ podman pod inspect mypod
    {
    "Id": "db99446fa9c6d10b973d1ce55a42a6850357e0cd447d9bac5627bb2516b5b19a",
    "Name": "mypod",
    "Created": "2020-09-08T10:35:07.536541534+02:00",
    "CreateCommand": ["podman", "pod", "create", "--name", "mypod"],
    "State": "Running",
    "Hostname": "mypod",
    "CreateCgroup": false,
    "CgroupParent": "/libpod_parent",
    "CgroupPath": "/libpod_parent/db99446fa9c6d10b973d1ce55a42a6850357e0cd447d9bac5627bb2516b5b19a",
    "CreateInfra": false,
    "InfraContainerID": "891c54f70783dcad596d888040700d93f3ead01921894bc19c10b0a03c738ff7",
    "SharedNamespaces": ["uts", "ipc", "net"],
    "NumContainers": 2,
    "Containers": [
    {
    "Id": "891c54f70783dcad596d888040700d93f3ead01921894bc19c10b0a03c738ff7",
```
You can see information about containers in the pod.

Additional resources

- For more information on the `podman pod top` command, type `man podman-pod-top`.
- For more information on the `podman pod stats` command, type `man podman-pod-stats`.
- For more information on the `podman pod inspect` command, type `man podman-pod-inspect`.

### 3.8. STOPPING PODS

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

**Prerequisites**

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

**Procedure**

1. To stop the pod `mypad`, type:

   ```bash
   $ podman pod stop mypod
   ```

2. List all pods and containers associated with them:

   ```bash
   $ podman ps -a --pod
   CONTAINER ID  IMAGE                               COMMAND    CREATED             STATUS
   PORTS   NAMES               POD ID        PODNAME
   5df5c48f8e577  registry.redhat.io/ubi8/ubi:latest  /bin/bash  About a minute ago  Exited (0) 7
   seconds ago          myubi               223df6b390b4  mypod
   3afcd93de3e k8s.gcr.io/pause:3.2                           About a minute ago  Exited (0) 7
   seconds ago          8a4e6527ac9d-infra  223df6b390b4  mypod
   ```

   You can see that the pod `mypad` and container `myubi` are in "Exited" status.

**Additional resources**

- For more information on the `podman pod stop` command, type `man podman-pod-stop`.
3.9. REMOVING PODS

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

Prerequisites

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

Procedure

1. To remove the pod `mypod`, type:

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

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

2. To check that all containers and pods were removed, type:

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

Additional resources

- For more information on the `podman pod rm` command, type `man podman-pod-rm`. 


CHAPTER 4. ADDING SOFTWARE TO A RUNNING UBI CONTAINER

UBI images are built from Red Hat content. These UBI images also provide a subset of Red Hat Enterprise Linux packages that are freely available to install for use with UBI. To add or update software, UBI images are pre-configured to point to the freely available yum repositories that hold official Red Hat RPMs.

To add packages from UBI repos to running UBI containers:

- On **ubi** images, the **yum** command is installed to let you draw packages.
- On **ubi-minimal** images, the **microdnf** command (with a smaller feature set) is included instead of **yum**.

Keep in mind that installing and working with software packages directly in running containers is just for adding packages temporarily or learning about the repositories. Refer to the "Build a UBI-based image" for more permanent ways of building UBI-based images.

Here are a few issues to consider when working with UBI images:

- Hundreds of RPM packages used in existing Application streams runtime images are stored in the yum repositories packaged with the new UBI images. Feel free to install those RPMs on your UBI images to emulate the runtime (python, php, nodejs, etc.) that interests you.
- Because some language files and documentation have been stripped out of the minimal UBI image (**ubi8/ubi-minimal**), running **rpm -Va** inside that container will show the contents of many packages as being missing or modified. If having a complete list of files inside that container is important to you, consider using a tool such as **Tripwire** to record the files in the container and check it later.
- After a layered image has been created, use **podman history** to check which UBI image it was built on. For example, after completing the webserver example shown earlier, type **podman history johndoe/webserver** to see that the image it was built on includes the image ID of the UBI image you added on the FROM line of the Dockerfile.

When you add software to a UBI container, procedures differ for updating UBI images on a subscribed RHEL host or on an unsubscribed (or non-RHEL) system. Those two ways of working with UBI images are illustrated below.

4.1. ADDING SOFTWARE TO A UBI CONTAINER ON SUBSCRIBED HOST

If you are running a UBI container on a registered and subscribed RHEL host, the main RHEL Server repository is enabled inside the standard UBI container, along with all the UBI repositories. So the full set of Red Hat packages is available. From the UBI minimal container, all UBI repositories are enabled by default, but no repositories are enabled from the host by default.

4.2. ADDING SOFTWARE INSIDE THE STANDARD UBI CONTAINER

To ensure the containers you build can be redistributed, disable non-UBI yum repositories in the standard UBI image when you add software. If you disable all yum repositories except for UBI repositories, only packages from the freely available repositories are used when you add software.

With a shell open inside a standard UBI base image container (**ubi8/ubi**) from a subscribed RHEL host, run the following command to add a package to that container (for example, the **bzip2** package):
# yum install --disablerepo=* --enablerepo=ubi-8-appstream --enablerepo=ubi-8-baseos bzip2

To add software inside a standard UBI container that is in the RHEL server repository, but not in UBI repositories, do not disable any repositories and just install the package:

# yum install zsh

To install a package that is in a different host repository from inside the standard UBI container, you have to explicitly enable the repository you need. For example:

# yum install --enablerepo=rhel-7-server-optional-rpms zsh-html

**WARNING**

Installing Red Hat packages that are not inside the Red Hat UBI repos might limit how widely you can distribute the container outside of subscribed hosts.

### 4.3. ADDING SOFTWARE INSIDE THE MINIMAL UBI CONTAINER

UBI yum repositories are enabled inside the UBI minimal image by default.

To install the same package demonstrated earlier (bzip2) from one of those UBI yum repositories on a subscribed RHEL host from the UBI minimal container, type:

# microdnf install bzip2

To install packages inside a minimal UBI container from repositories available on a subscribed host that are not part of a UBI yum repository, you would have to explicitly enable those repositories. For example:

# microdnf install --enablerepo=rhel-7-server-rpms zsh
# microdnf install --enablerepo=rhel-7-server-rpms \ --enablerepo=rhel-7-server-optional-rpms zsh-html

**WARNING**

Using non-UBI RHEL repositories to install packages in your UBI images could restrict your ability to share those images to run outside of subscribed RHEL systems.

### 4.4. ADDING SOFTWARE TO A UBI CONTAINER ON UNSUBSCRIBED HOST
To add software packages to a running container that is either on an unsubscribed RHEL host or some other Linux system, you do not have to disable any yum repositories. For example:

```
# yum install bzip2
```

To install that package on an unsubscribed RHEL host from the UBI minimal container, type:

```
# microdnf install bzip2
```

As noted earlier, both of these means of adding software to a running UBI container are not intended for creating permanent UBI-based container images. For that, you should build new layers on to UBI images, as described in the following section.

### 4.5. BUILDING AN UBI-BASED IMAGE

You can build UBI-based container images in the same way you build other images, with one exception. You should disable all non-UBI yum repositories when you actually build the images, if you want to be sure that your image only contains Red Hat software that you can redistribute.

Here is an example of creating a UBI-based Web server container from a Dockerfile with the `buildah` utility:

#### NOTE

For `ubi8/ubi-minimal` images, use `microdnf` instead of `yum` below:

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

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

   ```
   FROM registry.access.redhat.com/ubi8/ubi
   USER root
   LABEL maintainer="John Doe"
   # Update image
   RUN yum update --disablerepo=* --enablerepo=ubi-8-appstream --enablerepo=ubi-8-baseos -y && rm -rf /var/cache/yum
   RUN yum install --disablerepo=* --enablerepo=ubi-8-appstream --enablerepo=ubi-8-baseos 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 new image**  While in that directory, use `buildah` to create a new UBI layered image:

   ```bash
   # buildah bud -t johndoe/webserver .
   STEP 1: FROM registry.access.redhat.com/ubi8/ubi:latest
   STEP 2: USER root
   STEP 3: LABEL maintainer="John Doe"
   STEP 4: RUN yum update --disablerepo=* --enablerepo=ubi-8-appstream --enablerepo=ubi-8-baseos
   ```
8-baseos -y

No packages marked for update
STEP 5: RUN yum install --disablerepo=* --enablerepo=ubi-8-appstream --enablerepo=ubi-8-baseos httpd -y
Loading plugins: ovl, product-id, search-disabled-repos
Resolving Dependencies
---> Running transaction check

Package Arch Version Repository Size

Installing:
httpd x86_64 2.4.37-10
  latest-rhubi-8.0-appstream 1.4 M

Transaction Summary

Installing dependencies:
apr x86_64 1.6.3-9.el8 latest-rhubi-8.0-appstream 125 k
apr-util x86_64 1.6.1-6.el8 latest-rhubi-8.0-appstream 105 k
httpd-filesystem noarch 2.4.37-10
  latest-rhubi-8.0-appstream 34 k
httpd-tools x86_64 2.4.37-10.
...

3. Test: Test the UBI layered webserver image:

   # podman run -d -p 80:80 johndoe/webserver
   bbe98c71d18720d966e4567949888dc4fb86eeec7d304e785d5177168a5965f64
   # curl http://localhost/index.html
   The Web Server is Running

4.6. USING APPLICATION STREAM RUNTIME IMAGES

Red Hat Enterprise Linux 8 Application Stream offers another set of container images that you can use as the basis for your container builds. These images are built on RHEL standard base images, with most already updated as UBI images. Each of these images include additional software you might want to use for specific runtime environments.

If you expect to build multiple images that require, for example, php runtime software, you can use provide a more consistent platform for those images by starting with a PHP Application Stream image.

Here are a few examples of Application Stream container images built on UBI base images, that are available from the Red Hat Registry (registry.access.redhat.com or registry.redhat.io):
- **ubi8/php-72**: PHP 7.2 platform for building and running applications
- **ubi8/nodejs-10**: Node.js 10 platform for building and running applications. Used by Node.js 10 Source-To-Image builds
- **ubi8/ruby25**: Ruby 2.5 platform for building and running applications
- **ubi8/python-27**: Python 2.7 platform for building and running applications
- **ubi8/python-36**: Python 3.6 platform for building and running applications
- **ubi8/s2i-core**: Base image with essential libraries and tools used as a base for builder images like perl, python, ruby, and so on
- **ubi8/s2i-base**: Base image for Source-to-Image builds

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. Be sure to use the UBI image names to pull those images.

RHEL 8 Application Stream container images are updated every time RHEL 8 base images are updated. For RHEL 7, these same images (referred to as Red Hat Software Collections images) are updated on a schedule that is separate from RHEL base image updates (as are related images for Dotnet and DevTools). Search the [Red Hat Container Catalog](#) for details on any of these images. For more information on update schedules, see the [Red Hat Container Image Updates](#).

### 4.7. GETTING UBI CONTAINER IMAGE SOURCE CODE

Source code is available for all Red Hat UBI-based images in the form of downloadable containers. Before continuing, be aware about Red Hat source containers:

- 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, instead of using `podman pull` command.
  - Use `skopeo copy` command to copy a source container image to a directory on your local system.
  - Use `skopeo inspect` command to inspect the source container image.

- For more details on `skopeo` command, see [Section 1.5. Using skopeo to work with container registries](#).

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

- Once a source container image is copied to a local directory, you can use a combination of `tar`, `gzip`, and `rpm` commands to work with that content.

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

**Procedure**

1. Use `skopeo copy` command to copy the source container image to a local directory:
$ skopeo copy
  docker://registry.access.redhat.com/ubi8:8.1-397-source
  dir:$HOME/TEST
...
Copying blob 477bc8106765 done
Copying blob c438818481d3 done
Copying blob 26fe858c966c done
Copying blob ba4b5f020b99 done
Copying blob f7d9700cd456 done
Copying blob ade06f94b556 done
Copying blob cc56c782b513 done
Copying blob dc9396fdada done
Copying blob feb6d2ae2524 done
Copying config dd4cd669a4 done
Writing manifest to image destination
Storing signatures

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

```
$ skopeo inspect dir:$HOME/TEST
{
  "Digest": "sha256:7ab721ef3305271bbb629a6db065c59bbaeb87bc53e7cbf88e2953a1217ba7322",
  "RepoTags": [],
  "Created": "2020-02-11T12:14:18.612461174Z",
  "DockerVersion": "",
  "Labels": null,
  "Architecture": "amd64",
  "Os": "linux",
  "Layers": [
    "sha256:1ae73d938ab9f11718d0f6a4148eb07d38ac1c0a70b1d03e751de8bf3c287f",
    "sha256:9fe966885cb8712c47ef5e24a0797a0d5fffb8b119c4bd4b400cc9e255f",
    "sha256:61b2527a4b836a4efbb82d4f49c0556c0f769570a6c0e121f88f8bcd90166",
    ...
    "sha256:cc56c782b513e2bddd2cc2af77b69e13df4ab6244db856c4d086206b46b9e5f",
    "sha256:dc9396fdada4e6c1ce667b306b7f08a83c9e6b39d0955c481b8ea5b2a465b32",
    "sha256:feb6d2ae252402ea6afca8a158a7d32c7e4572db0e6e5a5eab15d4e0777951e"
  ],
  "Env": null
}
```

3. To untar all the content, type:

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

4. To check the results, type:

```
$ find blobs/ rpm_dir/
blobs/
blobs/sha256
blobs/sha256/10914f1fff060ce31388f5ab963871870535aaa551629f5ad182384d60fd82
rpm_dir/
rpm_dir/gzip-1.9-4.el8.src.rpm
```
5. Begin examining and using the content.

4.8. ADDITIONAL RESOURCES

- Red Hat partners and customers can request new features, including package requests, by filling a support ticket through standard methods. Non-Red Hat customers do not receive support, but can file requests through the standard Red Hat Bugzilla for the appropriate RHEL product. For more information, see Red Hat Bugzilla Queue

- Red Hat partners and customers can file support tickets through standard methods when running UBI on a supported Red Hat platform (OpenShift/RHEL). Red Hat support staff will guide partners and customers. For more information, see Open a Support Case
CHAPTER 5. RUNNING SKOPEO AND BUILDAH IN A CONTAINER

With Skopeo, you can inspect images on a remote registry without having to download the entire image with all its layers. You can also use Skopeo for copying images, signing images, syncing images, and converting images across different formats and layer compressions.

Buildah facilitates building OCI container images. With Buildah, you can create a working container, either from scratch or using an image as a starting point. You can create an image either from a working container or via the instructions in a Dockerfile. You can mount and unmount a working container’s root filesystem.

Reasons to run Buildah and Skopeo in a container:

- **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.

- **Both**: You are running an older OS on the host but you want to run the latest version of Skopeo, Buildah, or both. The solution is to run Buildah in a container. For example, this is useful for running the latest version of Skopeo, Buildah, or both provided in RHEL 8 on a RHEL 7 container host which does not have access to the newest versions natively.

- **Both**: 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 both in a container, you can perform these specific tasks as a non-root user.

5.1. RUNNING SKOPEO IN A CONTAINER

This procedure demonstrates how to inspect a remote container image using Skopeo. Running Skopeo in a container means that the container root filesystem is isolated from the host root filesystem. To share or copy files between the host and container, you have to mount files and directories.

**Procedure**

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

   ```
   $ podman login registry.redhat.io
   Username: myuser@mycompany.com
   Password: **********
   Login Succeeded!
   ```

2. Get the `registry.redhat.io/rhel8/skopeo` container image:

   ```
   $ podman pull registry.redhat.io/rhel8/skopeo
   ```

3. Inspect a remote container image `registry.access.redhat.com/ubi8/ubi` using Skopeo:
The `--rm` option removes the `registry.redhat.io/rhel8/skopeo` image after the container exits.

Additional resources

- For more information about how to run Skopeo in a container, see the [How to run skopeo in a container](#) article by Valentin Rothberg.

# 5.2. RUNNING SKOPEO IN A CONTAINER USING CREDENTIALS

Working with container registries requires an authentication to access and alter data. Skopeo supports various ways to specify credentials.

With this approach you can specify credentials on the command line using the `--cred USERNAME[:PASSWORD]` option.

**Procedure**

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

  ```bash
  $ podman run --rm registry.redhat.io/rhel8/skopeo inspect --creds $USER:$PASSWORD docker://$IMAGE
  ```

Additional resources

- For more information about how to run Skopeo in a container, see the [How to run skopeo in a container](#) article by Valentin Rothberg.
5.3. RUNNING SKOPEO IN A CONTAINER USING AUTHFILES

You can use an authentication file (authfile) to specify credentials. The `skopeo login` command logs into the specific registry and stores the authentication token in the authfile. The advantage of using authfiles is preventing the need to repeatedly enter credentials.

When running on the same host, all container tools such as Skopeo, Buildah, and Podman share the same authfile. When running Skopeo in a container, you have to either share the authfile on the host by volume-mounting the authfile in the container, or you have to reauthenticate within the container.

Procedure

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

  ```
  $ podman run --rm -v $AUTHFILE:/auth.json registry.redhat.io/rhel8/skopeo inspect docker://$IMAGE
  ```

  The `-v $AUTHFILE:/auth.json` option volume-mounts an authfile at /auth.json within the container. Skopeo can now access the authentication tokens in the authfile on the host and get secure access to the registry.

  Other Skopeo commands work similarly, for example:

  - Use the `skopeo-copy` command to specify credentials on the command line for the source and destination image using the `--source-creds` and `--dest-creds` options. It also reads the /auth.json authfile.

  - If you want to specify separate authfiles for the source and destination image, use the `--source-authfile` and `--dest-authfile` options and volume-mount those authfiles from the host into the container.

Additional resources

- For more information about how to run Skopeo in a container, see the How to run skopeo in a container article by Valentin Rothberg.

5.4. COPYING CONTAINER IMAGES TO OR FROM THE HOST

Skopeo, Buildah, and Podman share the same local container-image storage. If you want to copy containers to or from the host container storage, you need to mount it into the Skopeo container.

**NOTE**

The path to the host container storage differs between root (`/var/lib/containers/storage`) and non-root users (`$HOME/.local/share/containers/storage`).

Procedure

1. Copy the `registry.access.redhat.com/ubi8/ubi` image into your local container storage:

   ```
   $ podman run --privileged --rm -v
   $HOME/.local/share/containers/storage:/var/lib/containers/storage
   registry.redhat.io/rhel8/skopeo skopeo copy docker://registry.access.redhat.com/ubi8/ubi
   ```
containers-storage:registry.access.redhat.com/ubi8/ubi

- The `--privileged` option disables all security mechanisms. Red Hat recommends only using this option in trusted environments.

- To avoid disabling security mechanisms, export the images to a tarball or any other path-based image transport and mount them in the Skopeo container:
  
  ```
  $ podman save --format oci-archive -o oci.tar $IMAGE
  $ podman run --rm -v oci.tar:/oci.tar registry.redhat.io/rhel8/skopeo copy oci-archive:/oci.tar $DESTINATION
  ```

2. To list images in local storage:

```
$ podman images
REPOSITORY                              TAG     IMAGE ID       CREATED        SIZE
registry.access.redhat.com/ubi8/ubi     latest  ecbc6f53bba0 8 weeks ago 211 MB
```

Additional resources

- For more information about how to run Skopeo in a container, see the [How to run skopeo in a container](#) article by Valentin Rothberg.

## 5.5. Running Buildah in a Container

The procedure demonstrates how to run Buildah in a container and create a working container based on an image.

### Procedure

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

   ```
   $ podman login registry.redhat.io
   Username: myuser@mycompany.com
   Password: ***********
   Login Succeeded!
   ```

2. Pull and run the `registry.redhat.io/rhel8/buildah` image:

   ```
   # podman run --rm --device /dev/fuse -it registry.redhat.io/rhel8/buildah  /bin/bash
   ```

   - The `--rm` option removes the `registry.redhat.io/rhel8/buildah` image after the container exits.

   - The `--device` option adds a host device to the container.

3. Create a new container using a `registry.access.redhat.com/ubi8` image:

   ```
   # buildah from registry.access.redhat.com/ubi8
   ...
   ubi8-working-container
   ```

4. Run the `ls /` command inside the `ubi8-working-container` container:
To list all images in a local storage, enter:

```
# buildah images
```

<table>
<thead>
<tr>
<th>REPOSITORY</th>
<th>TAG</th>
<th>IMAGE ID</th>
<th>CREATED</th>
<th>SIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>registry.access.redhat.com/ubi8</td>
<td>latest</td>
<td>ecbc6f53bba0</td>
<td>5 weeks ago</td>
<td>211 MB</td>
</tr>
</tbody>
</table>

To list the working containers and their base images, enter:

```
# buildah containers
```

<table>
<thead>
<tr>
<th>CONTAINER ID</th>
<th>BUILDER</th>
<th>IMAGE ID</th>
<th>IMAGE NAME</th>
<th>CONTAINER NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>0aaba7192762</td>
<td>*</td>
<td>ecbc6f53bba0</td>
<td>registry.access.redhat.com/ubi8</td>
<td>ubi8-working-container</td>
</tr>
</tbody>
</table>

To push the `registry.access.redhat.com/ubi8` image to the a local registry located on `registry.example.com`:

```
# buildah push ecbc6f53bba0 registry.example.com:5000/ubi8/ubi
```

Additional resources

- For more information about how to run Buildah in a container, see the Best practices for running Buildah in a container article by Daniel Walsh.
CHAPTER 6. RUNNING SPECIAL CONTAINER IMAGES

Use this chapter to learn about some special types of container images. These include:

- **Toolbox**: Instead of burdening a host system by installing tools needed to debug problems or monitor features, you can run the `toolbox` command. Toolbox starts a `support-tools` container image that holds tools you can use to run reports or diagnose problems on the host.

- **Runlabels**: Some container images have labels built in that allow you to run those containers with preset options and arguments. The `podman container runlabel <label>` command, allows you to execute the command defined in that `<label>` for the container image. Supported labels are `install`, `run` and `uninstall`.

### 6.1. TROUBLESHOOTING CONTAINER HOSTS WITH TOOLBOX

Instead of installing troubleshooting tools directly on your RHEL 8 system, the `toolbox` utility offers a way to temporarily add those tools, then easily discard them when you are done. The `toolbox` utility works by:

- Pulling the `registry.redhat.io/rhel8/support-tools` image to your local system.

- Starting up a container from the image, then running a shell inside the container from which you can access the host system.

The `support-tools` container allows you to:

- Run commands that may not be installed on the host system, such as `sosreport`, `strace`, or `tcpdump`, in a way that lets them act on the host system.

- Install more software inside the container to use on the host system.

- Discard the container when you are done.

The following illustrates a typical `toolbox` session.

**Procedure**

1. Ensure that the `toolbox` and `podman` packages are installed:

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

2. Run the toolbox command to pull and run the `support-tools` image (inserting your Red Hat Customer Portal credentials when prompted):

   ```
   # toolbox
   Trying to pull registry.redhat.io/rhel8/support-tools...
   ...
   Would you like to authenticate to registry: ‘registry.redhat.io’ and try again? [y/N] y
   Username: johndoe
   Password: *************
   Login Succeeded!
   ```
Trying to pull registry.redhat.io/rhel8/support-tools...Getting image source signatures...Storing signatures
30e261462851238d38f4ef2afdaf55f1f8187775c5ca373b43e0f55722faaf97
Spawning a container 'toolbox-root' with image 'registry.redhat.io/rhel8/support-tools'
Detected RUN label in the container image. Using that as the default...
command: podman run -it --name toolbox-root --privileged --ipc=host --net=host --pid=host -e HOST=/host -e NAME=toolbox-root -e IMAGE=registry.redhat.io/rhel8/support-tools:latest -v /run:/run -v /var/log:/var/log -v /etc/machine-id:/etc/machine-id -v /etc/localtime:/etc/localtime -v /:/host registry.redhat.io/rhel8/support-tools:latest

3. Open a bash shell to run commands inside the container:

```
# bash-4.4#
```

4. From inside the container, the root file system on the host is available from the /host directory. The other directories shown are all inside the container.

```
# ls /
bin   dev  home  lib lost+found mnt proc run  srv  tmp  var
boot  etc  host  lib64 media     opt  root  sbin  sys  usr
```

5. Try to run a command inside your container. The sosreport command allows you to generate information about your system to send to Red Hat support:

```
bash-4.4# sosreport
```

sosreport (version 3.6)
This command will collect diagnostic and configuration information from this Red Hat Enterprise Linux system and installed applications.

An archive containing the collected information will be generated in 
/host/var/tmp/sosreport-rhel81beta-12345678-2019-10-29-pmgjncg.tar.xz and may be provided to a Red Hat support representative.

...Press ENTER to continue, or CTRL-C to quit. <Press ENTER>
...Your sosreport has been generated and saved in:
/host/var/tmp/sosreport-rhel81beta-12345678-2019-10-29-pmgjncg.tar.xz
The checksum is: c4e1f3e45f78a17ab4e45a05842ed
Please send this file to your support representative.

Note that sosreport command saves the report to the host (/host/var/tmp/sosreport-<ID>).

6. Install a software package inside the container, to add tools that are not already in the container. For example, to get a core dump of a running process on the host, install the procps and gcore packages, use ps to get the process ID of a running daemon, then use gcore to get a core dump:

```
bash-4.4# yum install procps gdb -y
bash-4.4# ps -ef | grep chronyd
994  809  10 Oct28 ?   00:00:00 /usr/sbin/chronyd
bash-4.4# gcore -o /host/tmp/chronyd.core 809
Missing separate debuginfo for target:/usr/sbin/chronyd
Try: dnf --enablerepo="*debug*" install /usr/lib/debug/.build-
id/96/0789a8a3bf28932b093e94b816be379f16a56a.debug
... Saved corefile /host/tmp/chronyd.core.809
[Inferior 1 (process 809) detached]

7. To leave the container and return to the host, type **exit**. The file is saved to /host/tmp/chronyd.core.809 and is available from /tmp/chronyd.core.809 on the host.

8. To remove the toolbox-root container, type:

```
# podman rm toolbox-root
```

You can change the registry, image, or container name used by toolbox by adding the following:

- **REGISTRY**: Change the registry from which the toolbox image is pulled. For example: `REGISTRY=registry.example.com`
- **IMAGE**: Change the image that is used. For example, `IMAGE=mysupport-tools`
- **TOOLBOX_NAME**: Change the name assigned to the running container. For example, `TOOLBOX_NAME=mytoolbox`

The next time you run `toolbox`, the new values from the `.toolboxrc` file are used.

### 6.1.1. Opening privileges to the host

When you run other commands from within the support-tools container (or any privileged container), they can behave differently than when run in a non-privileged container. Although `sosreport` can tell when it is running in a container, other commands need to be told to act on the host system (the `/host` directory). Here are examples of features that may or not be open to the host from a container:

- **Privileges**: A privileged container (`--privileged`) runs applications as root user on the host by default. The container has this ability because it runs with an `unconfined_t` SELinux security context. So you can, for example, delete files and directories mounted from the host that are owned by the root user.

- **Process tables**: Unlike a regular container that only sees the processes running inside the container, running a `ps -e` command within a privileged container (with `--pid=host` set) lets you see every process 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>`). With some commands, however, permissions issues could occur when they try to access processes from the container.

- **Network interfaces**: By default, a container has only one external network interface and one loopback network interface. With network interfaces open to the host (`--net=host`), you can access those 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.

### 6.2. RUNNING CONTAINERS WITH RUNLABELS

Some Red Hat images include labels that provide pre-set command lines for working with those images. Using the `podman container runlabel <label>` command, you can tell `podman` to execute the command defined in that `<label>` for the image. Existing runlabels include:
• `install`: Sets up the host system before executing the image. Typically, this results in creating files and directories on the host that the container can access when it is run later.

• `run`: Identifies podman command line options to use when running the container. Typically, the options will open privileges on the host and mount the host content the container needs to remain permanently on the host.

• `uninstall`: Cleans up the host system after you are done running the container.

Red Hat images that have one or more runlabels include the `rsyslog` and `support-tools` images. The following procedure illustrates how to use those images.

### 6.2.1. Running rsyslog with runlabels

The `rhel8/rsyslog` container image is made to run a containerized version of the `rsyslogd` daemon. Inside the `rsyslog` image are `install`, `run`, and `uninstall` runlabels. The following procedure steps you through installing, running, and uninstalling the `rsyslog` image:

#### Procedure

1. Pull the `rsyslog` image:

   ```bash
   # podman pull registry.redhat.io/rhel8/rsyslog
   ```

2. Display (but do not yet run) the `install` runlabel for `rsyslog`:

   ```bash
   # podman container runlabel install --display rhel8/rsyslog
   command: podman run --rm --privileged -v /:/host -e HOST=/host -e IMAGE=registry.redhat.io/rhel8/rsyslog:latest -e NAME=rsyslog registry.redhat.io/rhel8/rsyslog:latest /bin/install.sh
   ```

   This shows that the command will open privileges to the host, mount the host root filesystem on `/host` in the container, and run an `install.sh` script.

3. Run the `install` runlabel for `rsyslog`:

   ```bash
   # podman container runlabel install rhel8/rsyslog
   command: podman run --rm --privileged -v /:/host -e HOST=/host -e IMAGE=registry.redhat.io/rhel8/rsyslog:latest -e NAME=rsyslog registry.redhat.io/rhel8/rsyslog:latest /bin/install.sh
   ```

   Creating directory at `/host/etc/pki/rsyslog`
   Creating directory at `/host/etc/rsyslog.d`
   Installing file at `/host/etc/rsyslog.conf`
   Installing file at `/host/etc/sysconfig/rsyslog`
   Installing file at `/host/etc/logrotate.d/syslog`

   This creates files on the host system that the `rsyslog` image will use later.

4. Display the `run` runlabel for `rsyslog`:

   ```bash
   # podman container runlabel run --display rhel8/rsyslog
   command: podman run -d --privileged --name rsyslog --net=host --pid=host -v /etc/pki/rsyslog:/etc/pki/rsyslog -v /etc/rsyslog.conf:/etc/rsyslog.conf -v /etc/sysconfig/rsyslog:/etc/sysconfig/rsyslog -v /etc/syslog.d:/etc/syslog.d -v /var/log:/var/log
   ```
5. Execute the run runlabel for rsyslog:

```
# podman container runlabel run rhel8/rsyslog
command: podman run -d --privileged --name rsyslog --net=host --pid=host -v /etc/pki/rsyslog:/etc/pki/rsyslog -v /etc/rsyslog.conf:/etc/rsyslog.conf -v /etc/sysconfig/rsyslog:/etc/sysconfig/rsyslog -v /etc/rsyslog.d -v /var/log -v /var/lib/rsyslog:/var/lib/rsyslog -v /run:/run -v /etc/machine-id:/etc/machine-id -v /etc/localtime:/etc/localtime -e IMAGE=registry.redhat.io/rhel8/rsyslog:latest -e NAME=rsyslog --restart=always registry.redhat.io/rhel8/rsyslog:latest /bin/rsyslog.sh
```

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

6. Display the uninstall runlabel for rsyslog:

```
# podman container runlabel uninstall --display rhel8/rsyslog
command: podman run --rm --privileged -v /:/host -e HOST=/host -e IMAGE=registry.redhat.io/rhel8/rsyslog:latest -e NAME=rsyslog registry.redhat.io/rhel8/rsyslog:latest /bin/uninstall.sh
```

In this case, the `uninstall.sh` script just removes the `/etc/logrotate.d/syslog` file. Note that it does not clean up the configuration files.

6.2.2. Running support-tools with runlabels

The `rhel8/support-tools` container image is made to run tools such as `sosreport` and `sos-collector` to help you analyze your host system. To simplify running the `support-tools` image, it includes a run runlabel. The following procedure describes how to run the `support-tools` image:

**Procedure**

1. Pull the `support-tools` image:

```
# podman pull registry.redhat.io/rhel8/support-tools
```

2. Display (but do not yet run) the run runlabel for `support-tools`:
This shows that the command mounts directories and opens privileges and namespaces (ipc, net, and pid) to the host system. It assigns the host’s root file system to the /host directory in the container.

3. Execute the run runlabel for support-tools:

```
# podman container runlabel run --display rhel8/support-tools
command: podman run -it --name support-tools --privileged --ipc=host --net=host --pid=host -e HOST=/host -e NAME=support-tools -e IMAGE=registry.redhat.io/rhel8/support-tools:latest -v /run:/run -v /var/log:/var/log -v /etc/machine-id:/etc/machine-id -v /etc/localtime:/etc/localtime -v /:/host registry.redhat.io/rhel8/support-tools:latest
```

This opens a bash shell inside the support-tools container. You can now run reports or debug tools against the host system (/host).

4. To leave the container and return to the host, type exit.

```
# exit
```
CHAPTER 7. PORTING CONTAINERS TO OPENSSHIFT USING PODMAN

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

The YAML files are:

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

Reasons to use YAML files:

1. You can re-run a local orchestrated set of containers and pods with minimal input required which can be useful for iterative development.

2. You can run the same containers and pods on another machine. For example, to run an application in an OpenShift environment and to ensure that the application is working correctly. You can use `podman generate kube` command to generate a Kubernetes YAML file. Then, you can use `podman play` command to test the creation of pods and containers on your local system before you transfer the generated YAML files to the Kubernetes or OpenShift environment. Using the `podman play` command, you can also recreate pods and containers originally created in OpenShift or Kubernetes environments.

7.1. GENERATING A KUBERNETES YAML FILE USING PODMAN

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

**Prerequisites**

- The pod has been created. For details, see 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
   5df5c48feea87 registry.access.redhat.com/ubi8/ubi:latest /bin/bash Less than a second ago Up Less than a second ago myubi 223df6b390b4
   3afdc93d93e3e k8s.gcr.io/pause:3.1 Less than a second ago Up Less than a second ago 223df6b390b4-infra 223df6b390b4
   ```

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

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

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

3. Display the mypod.yaml file:

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

4. To stop the pod mypod:

$ podman pod stop mypod

5. List all pods and containers associated with them:

$ podman ps -a --pod
CONTAINER ID IMAGE COMMAND CREATED STATUS
PORTS NAMES POD ID PODNAME
12fc1ada3d4f registry.redhat.io/ubi8/ubi:latest /bin/bash About a minute ago Exited (0) 7
Here, the pod mypod and container myubi are in "Exited" status.

6. To remove the pod mypod:

   $ podman pod rm mypod

   8a4e6527ac9d2276e8a6b9c2670608866dbcb5da3efbd06f70ec2ecc88e247eb

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

7. To check that all containers and pods were removed:

   $ podman ps
   $ podman pod ps

Additional resources

- The podman-generate-kube man page.
- Podman: Managing pods and containers in a local container runtime article by Brent Baude.

7.2. GENERATING A KUBERNETES YAML FILE IN OPENSSHIFT ENVIRONMENT

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

Procedure

- Generate the YAML file for your myapp application:

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

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

  **NOTE**

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

7.3. STARTING CONTAINERS AND PODS WITH PODMAN

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

Procedure
1. Create the pod and the container from the `mypad.yaml` file:

   ```
   $ podman play kube mypod.yaml
   Pod:
   b8c5b99ba846cccf76c3ef257e5761c2d8a5ca4d7fa3880531aeec79c0dacb22
   Container:
   848179395ebd33dd91d14ffbd7ae273158d9695a081468f487af4e356888ece
   ```

2. List all pods:

   ```
   $ podman pod ps
   POD ID     NAME    STATUS     CREATED             # OF CONTAINERS   INFRA ID
   b8c5b99ba846 mypod    Running   19 seconds ago     2                 aa4220eaf4bb
   ```

3. List all pods and containers associated with them:

   ```
   $ podman ps -a --pod
   CONTAINER ID  IMAGE                                           COMMAND       CREATED             STATUS                      PORTS   NAMES               POD
   848179395ebd  registry.access.redhat.com/ubi8/ubi:latest   /bin/bash    About a minute ago  Up About a minute ago myubi b8c5b99ba846
   a4220eaf4bb  k8s.gcr.io/pause:3.1                           About a minute ago  Up About a minute ago b8c5b99ba846
   ```

   The pod IDs from `podman ps` command matches the pod ID from the `podman pod ps` command.

Additional resources

- The `podman-play-kube` man page.
- `Podman can now ease the transition to Kubernetes and CRI-O` article by Brent Baude.

### 7.4. STARTING CONTAINERS AND PODS IN OPENSIFT 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.
Podman (Pod Manager) is a fully featured container engine that is a simple daemonless tool. Podman provides a Docker-CLI comparable command line that eases the transition from other container engines and allows the management of pods, containers and images. It was not originally designed to bring up an entire Linux system or manage services for such things as start-up order, dependency checking, and failed service recovery. That is the job of a full-blown initialization system like systemd. Red Hat has become a leader in integrating containers with systemd, so that OCI and Docker-formatted containers built by Podman can be managed in the same way that other services and features are managed in a Linux system. You can use systemd initialization service to work with pods and containers. You can use the podman generate systemd command to generate a systemd unit file for containers and pods.

With systemd unit files, you can:

- Set up a container or pod to start as a systemd service.
- Define the order in which the containerized service runs and check for dependencies (for example making sure another service is running, a file is available or a resource is mounted).
- Control the state of the systemd system using the systemctl command.

This chapter provides you with information on how to generate portable descriptions of containers and pods using systemd unit files.

### 8.1. ENABLING SYSTEMD SERVICES

When enabling the service, you have different options.

**Procedure**

- Enable the service:
  - To enable a service at system start, no matter if user is logged in or not, enter:
    ```bash
    # 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:
    ```bash
    $ 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:
    ```bash
    # loginctl enable-linger <username>
    ```

**Additional resources**

- For more information on the systemctl and loginctl commands, enter man systemctl or man loginctl, respectively.
To learn more about configuring services with systemd, refer to the Configuring basic system settings guide chapter called Managing services with systemd.

8.2. GENERATING A SYSTEMD UNIT FILE USING PODMAN

Podman allows systemd to control and manage container processes. You can generate a systemd unit file for the existing containers and pods using `podman generate systemd` command. It is recommended to use `podman generate systemd` because the generated units files change frequently (via updates to Podman) and the `podman generate systemd` ensures that you get the latest version of unit files.

Procedure

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

   ```
   $ podman create -d --name myubi registry.access.redhat.com/ubi8:latest top
   0280afe98bb75a5c5e713b28de4b7c5cb49f156f1ccee4a208f13flee2f75cb453
   ```

2. Use the container name or ID to generate the systemd unit file and direct it into the `~/.config/systemd/user/container-myubi.service` file:

   ```
   $ podman generate systemd --name myubi > ~/.config/systemd/user/container-myubi.service
   ```

Verification steps

- To display the content of generated systemd unit file, enter:

  ```
  $ cat ~/.config/systemd/user/container-myubi.service
  # container-myubi.service
  # autogenerated by Podman 2.0.0
  # Tue Aug 11 10:51:04 CEST 2020
  
  [Unit]
  Description=Podman container-myubi.service
  Documentation=man:podman-generate-systemd(1)
  Wants=network.target
  After=network-online.target
  
  [Service]
  Environment=PODMAN_SYSTEMD_UNIT=%n
  Restart=on-failure
  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/0280afe98bb75a5c5e713b28de4b7c5cb49f156f1ccee4a208f13flee2f75cb453/userdata/conmon.pid
  KillMode=none
  Type=forking
  
  [Install]
  WantedBy=multi-user.target default.target
  ```
The `Restart=on-failure` line sets the restart policy and instructs systemd to restart when the service cannot be started or stopped cleanly, or when the process exits non-zero.

- The `ExecStart` line describes how we start the container.
- The `ExecStop` line describes how we stop and remove the container.

Additional resources

- Running containers with Podman and shareable systemd services article by Valentin Rothberg.

### 8.3. AUTO-GENERATING A SYSTEMD UNIT FILE USING PODMAN

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

**Procedure**

1. Pull the image you want to use on your system. For example, to pull the `busybox` image:

```
# podman pull busybox:latest
```

2. List all images available on your system:

```
# podman images
Repository                  Tag          Image ID            Created       Size
docker.io/library/busybox  latest       c7c37e472d31 3 weeks ago 1.45 MB
```

3. Create the `busybox` container:

```
# podman create --name busybox busybox:latest
```

4. To verify the container has been created, list all containers:

```
# podman ps -a
CONTAINER ID  IMAGE                             COMMAND  CREATED        STATUS   PORTS NAMES
1e12cf95e305  docker.io/library/busybox:latest  sh       7 seconds ago Created busybox
```

5. Generate a systemd unit file for the `busybox` container:

```
# podman generate systemd --new --files --name busybox /root/container-busybox.service
```

6. Display the content of the generated `container-busybox.service` systemd unit file:

```
# vim container-busybox.services

# container-busybox.service
# autogenerated by Podman 2.0.0-rc7
# Mon Jul 27 11:06:32 CEST 2020
```
[Unit]
Description=Podman container-busybox.service
Documentation=man:podman-generate-systemd(1)
Wants=network.target
After=network-online.target

[Service]
Environment=PODMAN_SYSTEMD_UNIT=%n
Restart=on-failure
ExecStartPre=/usr/bin/rm -f %t/container-busybox.pid %t/container-busybox.ctr-id
ExecStart=/usr/bin/podman run --conmon-pidfile %t/container-busybox.pid --cidfile %t/container-busybox.ctr-id --cgroups=no-conmon -d --replace --name busybox busybox:latest
ExecStop=/usr/bin/podman stop --ignore --cidfile %t/container-busybox.ctr-id -t 10
ExecStopPost=/usr/bin/podman rm --ignore -f --cidfile %t/container-busybox.ctr-id
PIDFile=%t/container-busybox.pid
KillMode=none
Type=forking

[Install]
WantedBy=multi-user.target default.target

Note that unit files generated using the --new option do not expect containers and pods to exist. Therefore, they perform the podman run command when starting the service (see the ExecStart line) instead of the podman start command. For example, see Section 7.2. Generating a systemd unit file using Podman.

- The podman run command uses the following command-line options:
  - The --conmon-pidfile option points to a path to store the process ID for the conmon process running on the host. The conmon process terminates with the same exit status as the container, which allows systemd to report the correct service status and restart the container if needed.
  - The --cidfile option points to the path that stores the container ID.
  - The %t is the path to the run time directory root, for example /run/user/$UserID.
  - The %n is the full name of the service.

7. Copy unit files to /usr/lib/systemd/system for installing them as a root user:

```bash
# cp -Z container-busybox.service /usr/lib/systemd/system
Created symlink /etc/systemd/system/multi-user.target.wants/container-busybox.service → /usr/lib/systemd/system/container-busybox.service.
Created symlink /etc/systemd/system/default.target.wants/container-busybox.service → /usr/lib/systemd/system/container-busybox.service.
```

Additional resources

- Improved Systemd Integration with Podman 2.0 article by Valentin Rothberg and Dan Walsh.
- To learn more about configuring services with systemd, refer to the Configuring basic system settings guide chapter called Managing services with systemd.
8.4. AUTO-STARTING CONTAINERS USING SYSTEMD

You can control the state of the systemd system and service manager using the `systemctl` command. This section shows the general procedure on how to enable, start, stop the service as a non-root user. To install the service as a root user, omit the `--user` option.

Procedure

1. Reload systemd manager configuration:
   
   ```
   # systemctl --user daemon-reload
   ```

2. Enable the service `container.service` and start it at boot time:
   
   ```
   # systemctl --user enable container.service
   ```

3. To start the service immediately:
   
   ```
   # systemctl --user start container.service
   ```

4. Check the status of the service:
   
   ```
   $ systemctl --user status container.service
   container.service - Podman container.service
   Loaded: loaded (/home/user/.config/systemd/user/container.service; enabled; vendor preset: enabled)
   Active: active (running) since Wed 2020-09-16 11:56:57 CEST; 8s ago
   Docs: man:podman-generate-systemd(1)
   Process: 80602 ExecStart=/usr/bin/podman run --conmon-pidfile
   //run/user/1000/container.service-pid --cidfile //run/user/1000/container.service-cid -d ubi8-minimal:
   Process: 80601 ExecStartPre=/usr/bin/rm -f //run/user/1000/container.service-pid
   //run/user/1000/container.service-cid (code=exited, status=0/SUCCESS)
   Main PID: 80617 (conmon)
   CGroup: /user.slice/user-1000.slice/user@1000.service/container.service
   ├─ 2870 /usr/bin/podman
   │ └─ 80612 /usr/bin/slirp4netns --disable-host-loopback --mtu 65520 --enable-sandbox -
   │    -enable-seccomp -c -e 3 -r 4 --netns-type=path /run/user/1000/netns/cni->
   │     └─ 80614 /usr/bin/fuse-overlayfs -o
   │        lowerdir=/home/user/.local/share/containers/storage/overlay/l/YJSPGXM2OCDZPLMLXJOW3NRF6Q:/home/user/.local/share/containers/storage/overlay/l/
   │    ├── 80617 /usr/bin/common --api-version 1 -c
   │    │   cbc75d6031508dfd3d78a74a03e4ace1732b51223e72a2ce4aa3bfe10a78e4fa -u
   │    │   cbc75d6031508dfd3d78a74a03e4ace1732b51223e72a2ce4aa3bfe10a78e4fa
   │    │     └─ cbc75d6031508dfd3d78a74a03e4ace1732b51223e72a2ce4aa3bfe10a78e4fa
   │    │     └─ 80626 /usr/bin/coreutils --coreutils-prog-shebang=sleep /usr/bin/sleep 1d
   │    └─ 80617 /usr/bin/common --api-version 1 -c
   │        cbc75d6031508dfd3d78a74a03e4ace1732b51223e72a2ce4aa3bfe10a78e4fa
   │        └─ 80626 /usr/bin/coreutils --coreutils-prog-shebang=sleep /usr/bin/sleep 1d
   ```

You can check if the service is enabled using the `systemctl is-enabled container.service` command.

Verification steps

- List containers that are running or have exited:
# podman ps

<table>
<thead>
<tr>
<th>CONTAINER ID</th>
<th>IMAGE</th>
<th>COMMAND</th>
<th>CREATED</th>
<th>STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>f20988d59920</td>
<td>registry.access.redhat.com/ubi8-minimal:latest</td>
<td>top</td>
<td>12 seconds ago</td>
<td>Up 11 seconds ago</td>
</tr>
</tbody>
</table>

**NOTE**

To stop `container.service`, enter:

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

### Additional resources

- For more information on the `systemctl` command, type `man systemctl`.
- For more information, see Running containers with Podman and shareable systemd services article by Valentin Rothberg.
- To learn more about configuring services with systemd, refer to the Configuring basic system settings guide chapter called Managing services with systemd.

## 8.5. AUTO-STARTING PODS USING SYSTEMD

You can start multiple containers as systemd services. Note that the `systemctl` command should only be used on the pod and you should not start or stop containers individually via `systemctl`, as they are managed by the pod service along with the internal infra-container.

### Procedure

1. Create an empty pod, for example named `systemd-pod`:

   ```bash
   $ podman pod create --name systemd-pod
   11d4646ba41b1fffa51c108cbdf97cfab3213f7bd9b3e1ca52fe81b90fed577
   ```

2. List all pods:

   ```bash
   $ podman pod ps
   POD ID       NAME       STATUS          CREATED     # OF CONTAINERS   INFRA ID
   11d4646ba41b systemd-pod Created 40 seconds ago 1 8a428b257111
   11d4646ba41b1fffa51c108cbdf97cfab3213f7bd9b3e1ca52fe81b90fed577
   ```

3. Create two containers in the empty pod. For example, to create `container0` and `container1` in `systemd-pod`:

   ```bash
   $ podman create --pod systemd-pod --name container0 registry.access.redhat.com/ubi8 top
   $ podman create --pod systemd-pod --name container1 registry.access.redhat.com/ubi8 top
   ```

4. List all pods and containers associated with them:

   ```bash
   $ podman ps -a --pod
   CONTAINER ID  IMAGE                                      COMMAND | CREATED | STATUS
   ```
PORTS     NAMES               POD ID        PODNAME
24666f47d9b2  registry.access.redhat.com/ubi8:latest  top      3 minutes ago  Created
56eb1bf0cde8 k8s.gcr.io/pause:3.2                             4 minutes ago  Created
62118d170e43 registry.access.redhat.com/ubi8:latest  top      3 seconds ago  Created

5. Generate the systemd unit file for the new pod:

```bash
$ 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:

```bash
$ cat pod-systemd-pod.service
```

```
[Unit]
Description=Podman pod-systemd-pod.service
Documentation=man:podman-generate-systemd(1)
Wants=network.target
After=network-online.target
Requires=container_CONTAINER0.service container_CONTAINER1.service
Before=container_CONTAINER0.service container_CONTAINER1.service

[Service]
Environment=PODMAN_SYSTEMD_UNIT=%n
Restart=on-failure
ExecStart=/usr/bin/podman start c852fba568-infra
ExecStop=/usr/bin/podman stop -t 10 c852fba568-infra
ExecStopPost=/usr/bin/podman stop -t 10 c852fba568-infra
PIDFile=/run/user/1000/containers/overlay-containers/a7f863b2608add27a03ac2166d5d0164199f01eadf80b68b06a406c195105fc/userd ata/conmon.pid
KillMode=none
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 2.0.3
# Tue Jul 28 14:00:46 EDT 2020

[Unit]
Description=Podman container-container0.service
Documentation=man:podman-generate-systemd(1)
Wants=network.target
After=network-online.target
BindsTo=pod-systemd-pod.service
After=pod-systemd-pod.service

[Service]
Environment=PODMAN_SYSTEMD_UNIT=%n
Restart=on-failure
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/12e8537f12854b8283f791974494a02aa6c92630d76d1050237839b61508a008/user data/conmon.pid
KillMode=none
Type=forking

[Install]
WantedBy=multi-user.target default.target
```

- The `BindsTo` line in the `[Unit]` section defines the dependency on the `pod-systemd-pod.service` unit file.
- The `ExecStart` and `ExecStop` lines in the `[Service]` section start and stop the `container0` respectively.

8. Display `container-container1.service` unit file:

```
$ cat container-container1.service
```

9. Copy all the generated files to `$HOME/.config/systemd/user` for installing as a non-root user:

```
$ cp pod-systemd-pod.service container-container0.service container-container1.service
$HOME/.config/systemd/user
```

10. Enable the service and start at user login:

```
$ systemctl enable --user pod-systemd-pod.service
Created symlink /home/user1/.config/systemd/user/multi-user.target.wants/pod-systemd-pod.service → /home/user1/.config/systemd/user/pod-systemd-pod.service.
Created symlink /home/user1/.config/systemd/user/default.target.wants/pod-systemd-pod.service → /home/user1/.config/systemd/user/pod-systemd-pod.service.
```

Note that the service stops at user logout.
Verification steps

- Check if the service is enabled:
  
  ```
  $ systemctl is-enabled pod-systemd-pod.service
  enabled
  ```

Additional resources

- For more information on the `podman create` command, type `man podman-create`.

- For more information on the `podman generate systemd` command, type `man podman-generate-systemd`.

- For more information on the `systemctl` command, type `man systemctl`.

- For more information, see the Running containers with Podman and shareable systemd services article by Valentin Rothberg.

- To learn more about configuring services with systemd, refer to the Configuring basic system settings guide chapter called Managing services with systemd.
CHAPTER 9. BUILDING CONTAINER IMAGES WITH BUILDAH

The buildah command lets you create container images from a working container, a Dockerfile, or from scratch. The resulting images are OCI compliant, so they will work on any container runtime that meets the OCI Runtime Specification (such as Docker and CRI-O).

This section describes how to use the buildah command to create and otherwise work with containers and container images.

9.1. UNDERSTANDING BUILDAH

Using Buildah is different from building images with the docker command in the following ways:

- **No Daemon!** Bypasses the Docker daemon! So no container runtime (Docker, CRI-O, or other) is needed to use Buildah.
- **Base image or scratch** Lets you not only build an image based on another container, but also lets you start with an empty image (scratch).
- **Build tools external** Doesn’t include build tools within the image itself. As a result, Buildah:
  - Reduces the size of images you build
  - Makes the image more secure by not having the software used to build the container (like gcc, make, and yum) within the resulting image.
  - Creates images that require fewer resources to transport the images (because they are smaller).

Buildah is able to operate without Docker or other container runtimes by storing data separately and by including features that let you not only build images, but run those images as containers as well. By default, Buildah stores images in an area identified as containers-storage (/var/lib/containers).

NOTE

The containers-storage location that the buildah command uses by default is the same place that the CRI-O container engine uses for storing local copies of images. So images pulled from a registry by either CRI-O or Buildah, or committed by the buildah command, will be stored in the same directory structure. Currently, however, CRI-O and Buildah cannot share containers, though they can share images.

There are more than a dozen options to use with the buildah command. Some of the main activities you can do with the buildah command include:

- **Build a container from a Dockerfile** Use a Dockerfile to build a new container image (buildah bud).
- **Build a container from another image or scratch** Build a new container, starting with an existing base image (buildah from <imagename>) or from scratch (buildah from scratch)
- **Inspecting a container or image** View metadata associated with the container or image (buildah inspect)
- **Mount a container** Mount a container’s root filesystem to add or change content (buildah mount).
• **Create a new container layer**: Use the updated contents of a container’s root filesystem as a filesystem layer to commit content to a new image (**buildah commit**).

• **Unmount a container**: Unmount a mounted container (**buildah umount**).

• **Delete a container or an image**: Remove a container (**buildah rm**) or a container image (**buildah rmi**).

For more details on Buildah, see the GitHub Buildah page. The GitHub Buildah site includes man pages and software that might be more recent than is available with the RHEL version. Here are some other articles on Buildah that might interest you:

• Buildah Tutorial 1: Building OCI container images

• Buildah Tutorial 2: Using Buildah with container registries

• Buildah Blocks - Getting Fit

### 9.1.1. Installing Buildah

The buildah package is available with the container-tools module in RHEL 8 (**yum module install container-tools**). You can install the buildah package separately by typing:

```
# yum -y install buildah
```

With the buildah package installed, you can refer to the man pages included with the buildah package for details on how to use it. To see the available man pages and other documentation, then open a man page, type:

```
# rpm -qd buildah
# man buildah
```

The following sections describe how to use **buildah** to get containers, build a container from a Dockerfile, build one from scratch, and manage containers in various ways.

### 9.2. GETTING IMAGES WITH BUILDAH

To get a container image to use with **buildah**, use the **buildah from** command. Notice that if you are using RHEL 8.0, you may encounter problems with authenticating to the repository, see [bug](#). Here’s how to get a RHEL 8 image from the Red Hat Registry as a working container to use with the **buildah** command:

```
# buildah from registry.redhat.io/ubi8/ubi
```

Getting image source signatures
Copying blob...
Writing manifest to image destination
Storing signatures
ubi-working-container
# buildah images
Notice that the result of the `buildah` command is an image (registry.redhat.io/ubi8/ubi:latest) and a working container that is ready to run from that image (ubi-working-container). Here’s an example of how to execute a command from that container:

```
# buildah run ubi-working-container cat /etc/redhat-release
Red Hat Enterprise Linux release 8.0
```

The image and container are now ready for use with Buildah.

### 9.3. BUILDING AN IMAGE FROM A DOCKERFILE WITH BUILDAH

With the `buildah` command, you can create a new image from a Dockerfile. The following steps show how to build an image that includes a simple script that is executed when the image is run.

This simple example starts with two files in the current directory: Dockerfile (which holds the instructions for building the container image) and myecho (a script that echoes a few words to the screen):

```
# ls
Dockerfile  myecho
# cat Dockerfile
FROM registry.redhat.io/ubi8/ubi
ADD myecho /usr/local/bin
ENTRYPOINT "/usr/local/bin/myecho"
# cat myecho
echo "This container works!"
# chmod 755 myecho
# ./myecho
This container works!
```

With the Dockerfile in the current directory, build the new container as follows:

```
# buildah bud -t myecho .
STEP 1: FROM registry.redhat.io/ubi8/ubi
STEP 2: ADD myecho /usr/local/bin
STEP 3: ENTRYPOINT "/usr/local/bin/myecho"
```

The `buildah bud` command creates a new image named myecho. To see that new image, type:

```
# buildah images
```

```
IMAGE NAME        IMAGE TAG  IMAGE ID      CREATED AT          SIZE
localhost/myecho  latest     a3882af49784  Jun 21, 2019 12:21  216 MB
```

Next, you can run the image, to make sure it is working.

### 9.3.1. Running the image you built

To check that the image you built previously works, you can run the image using `podman run`:
# podman run localhost/myecho
This container works!

9.3.2. Inspecting a container with Buildah

With **buildah inspect**, you can show information about a container or image. For example, to build and inspect the **myecho** image, type:

```
# buildah from localhost/myecho
# buildah inspect localhost/myecho | less
{
  "Type": "buildah 0.0.1",
  "FromImage": "docker.io/library/myecho:latest",
  "FromImage-ID": "e2b190ac8...",
  "Config": "{"created":":2018-11-13...

  "Entrypoint": [
      "/usr/local/bin/myecho"
  ],
  "WorkingDir": "/",
  "Labels": {
      "architecture": "x86_64",
      "authoritative-source-url": "registry.access.redhat.com",
      "build-date": "2018-09-19T20:46:28.459833",

To inspect a container from that same image, type the following:

```
# buildah inspect myecho-working-container | less
{
  "Type": "buildah 0.0.1",
  "FromImage": "docker.io/library/myecho:latest",
  "FromImage-ID": "e2b190a...",
  "Config": "{"created":":2018-11-13T19:5...
...
  "Container": "myecho-working-container",
  "ContainerID": "c0cd2e494d...",
  "MountPoint": "",
  "ProcessLabel": "system_u:system_r:svirt_lxc_net_t:s0:c89,c921",
  "MountLabel": "",
```

Note that the container output has added information, such as the container name, container id, process label, and mount label to what was in the image.

9.4. MODIFYING A CONTAINER TO CREATE A NEW IMAGE WITH BUILDAH

There are several ways you can modify an existing container with the **buildah** command and commit those changes to a new container image:

- Mount a container and copy files to it
- Use **buildah copy** and **buildah config** to modify a container

Once you have modified the container, use **buildah commit** to commit the changes to a new image.
9.4.1. Using `buildah mount` to modify a container

After getting an image with `buildah from`, you can use that image as the basis for a new image. The following text shows how to create a new image by mounting a working container, adding files to that container, then committing the changes to a new image.

Type the following to view the working container you used earlier:

```
# buildah containers
CONTAINER ID BUILDER IMAGE ID     IMAGE NAME  CONTAINER NAME
 dc8f21af4a47   *     1456eedf8101 registry.redhat.io/ubi8/ubi:latest
                  ubi-working-container
 6d1ffcc557d   *     ab230ac5aba3 docker.io/library/myecho:latest
                  myecho-working-container
```

Mount the container image and set the mount point to a variable (`$mymount`) to make it easier to deal with:

```
# mymount=$(buildah mount myecho-working-container)
# echo $mymount
/var/lib/containers/storage/devicemapper/mnt/176c273fe28c23e5319805a2c48559305a57a706cc7ae7beec7da4cd79ed3c02/rootfs
```

Add content to the script created earlier in the mounted container:

```
# echo 'echo "We even modified it."' >> $mymount/usr/local/bin/myecho
```

To commit the content you added to create a new image (named `myecho`), type the following:

```
# buildah commit myecho-working-container containers-storage:myecho2
```

To check that the new image includes your changes, create a working container and run it:

```
# buildah images
```

You can see that the new `echo` command added to the script displays the additional text.

When you are done, you can unmount the container:

```
# buildah umount myecho-working-container
```

9.4.2. Using `buildah copy` and `buildah config` to modify a container

With `buildah copy`, you can copy files to a container without mounting it first. Here’s an example, using `Red Hat Enterprise Linux 8 Building, running, and managing containers`
With **buildah copy**, you can copy files to a container without mounting it first. Here’s an example, using the **myecho-working-container** created (and unmounted) in the previous section, to copy a new script to the container and change the container’s configuration to run that script by default.

Create a script called **newecho** and make it executable:

```bash
# cat newecho
echo "I changed this container"
# chmod 755 newecho
```

Create a new working container:

```bash
# buildah from myecho:latest
myecho-working-container-2
```

Copy **newecho** to /usr/local/bin inside the container:

```bash
# buildah copy myecho-working-container-2 newecho /usr/local/bin
```

Change the configuration to use the **newecho** script as the new entrypoint:

```bash
# buildah config --entrypoint "/bin/sh -c /usr/local/bin/newecho" myecho-working-container-2
```

Run the new container, which should result in the **newecho** command being executed:

```bash
# buildah run myecho-working-container-2
I changed this container
```

If the container behaved as you expected it would, you could then commit it to a new image (mynewecho):

```bash
# buildah commit myecho-working-container-2 containers-storage:mynewecho
```

### 9.5. CREATING IMAGES FROM SCRATCH WITH BUILDAH

Instead of starting with a base image, you can create a new container that holds no content and only a small amount of container metadata. This is referred to as a **scratch** container. Here are a few issues to consider when choosing to create an image starting from a scratch container with the **buildah** command:

- When building a scratch container you can copy executable with no dependencies into the scratch image and make a few configuration settings to get a minimal container to work.

- To use tools like **yum** or **rpm** packages to populate the scratch container, you need to at least initialize an RPM database in the container and add a release package. The example below shows how to do that.

- If you end up adding a lot of RPM packages, consider using the **ubi** or **ubi-minimal** base images instead of a scratch image. Those base images have had documentation, language packs, and other components trimmed out, which can ultimately result in your image being smaller.

This example adds a Web service (httpd) to a container and configures it to run. To begin, create a scratch container:
# buildah from scratch

working-container

This creates just an empty container (no image) that you can mount as follows:

```bash
# scratchmnt=$(buildah mount working-container)
# echo $scratchmnt
/var/lib/containers/storage/devicemapper/mnt/cc92011e9a2b077d03a97c0809f113e7f6f0f29bdc6ab5e86b85430ec77b2bf6/rootfs
```

Initialize an RPM database within the scratch image and add the redhat-release package (which includes other files needed for RPMs to work):

```bash
# yum install -y --releasever=8 --installroot=$scratchmnt redhat-release
```

Install the httpd service to the scratch directory:

```bash
# yum install -y --setopt=reposdir=/etc/yum.repos.d --installroot=$scratchmnt --setopt=cachedir=/var/cache/dnf httpd
```

Add some text to an index.html file in the container, so you will be able to test it later:

```bash
# echo "Your httpd container from scratch worked." > $scratchmnt/var/www/html/index.html
```

Instead of running httpd as an init service, set a few buildah config options to run the httpd daemon directly from the container:

```bash
# buildah config --cmd "/usr/sbin/httpd -DFOREGROUND" working-container
# buildah config --port 80/tcp working-container
# buildah commit working-container localhost/myhttpd:latest
```

For now, you can use the Image ID to run the new image as a container with the podman command:

```bash
# podman images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
localhost/myhttpd   latest              47c0795d7b0e        9 minutes ago       665.6 MB
# podman run -p 8080:80 -d --name httpd-server 47c0795d7b0e
# curl localhost:8080
Your httpd container from scratch worked.
```

## 9.6. REMOVING IMAGES OR CONTAINERS WITH BUILDAH

When you are done with particular containers or images, you can remove them with buildah rm or buildah rmi, respectively. Here are some examples.

To remove the container created in the previous section, you could type the following to see the mounted container, unmount it and remove it:

```bash
# buildah containers
CONTAINER ID  BUILDER  IMAGE ID     IMAGE NAME                       CONTAINER NAME
05387e29ab93     *     c37e14066ac7 docker.io/library/myecho:latest  myecho-working-container
```
To remove the image you created previously, you could type the following:

```
# buildah rmi docker.io/library/myecho:latest
untagged: docker.io/library/myecho:latest
ab230ac5aba3b5a0a7c3d2c5e0793280c1a1b4d2457a75a01b70a4b7a9ed415a
```

### 9.7. USING CONTAINER REGISTRIES WITH BUILDAH

With Buildah, you can push and pull container images between your local system and public or private container registries. The following examples show how to:

- Push containers to and pull them from a private registry with buildah.
- Push and pull container between your local system and the Docker Registry.
- Use credentials to associate your containers with a registry account when you push them.

Use the skopeo command, in tandem with the `buildah` command, to query registries for information about container images.

#### 9.7.1. Pushing containers to a private registry

Pushing containers to a private container registry with the `buildah` command works much the same as pushing containers with the `docker` command. You need to:

- Set up a private registry (OpenShift provides a container registry or you can set up a Red Hat Quay container registry).
- Create or acquire the container image you want to push.
- Use `buildah push` to push the image to the registry.

To push an image from your local Buildah container storage, check the image name, then push it using the `buildah push` command. Remember to identify both the local image name and a new name that includes the location. For example, a registry running on the local system that is listening on TCP port 5000 would be identified as localhost:5000.

```
# buildah images
IMAGE ID     IMAGE NAME                       CREATED AT          SIZE
cb702d492ee9 docker.io/library/myecho2:latest Nov 12, 2018 16:50     3.143 KB

# buildah push --tls-verify=false myecho2:latest localhost:5000/myecho2:latest
Getting image source signatures
Copying blob sha256:e4efd0...
...  Writing manifest to image destination
Storing signatures
```
Use the `curl` command to list the images in the registry and `skopeo` to inspect metadata about the image:

```bash
# curl http://localhost:5000/v2/_catalog
{"repositories":["myatomic","myecho2"]}
# curl http://localhost:5000/v2/myecho2/tags/list
{"name":"myecho2","tags":["latest"]}
# skopeo inspect --tls-verify=false docker://localhost:5000/myecho2:latest | less
{
  "Name": "localhost:5000/myecho2",
  "Digest": "sha256:8999ff6050...",
  "RepoTags": [
    "latest"
  ],
  "Created": "2017-11-21T16:50:25.830343Z",
  "DockerVersion": "",
  "Labels": {
    "architecture": "x86_64",
    "authoritative-source-url": "registry.redhat.io",
  }
}
```

At this point, any tool that can pull container images from a container registry can get a copy of your pushed image. For example, on a RHEL 7 system you could start the docker daemon and try to pull the image so it can be used by the `docker` command as follows:

```bash
# systemctl start docker
# docker pull localhost:5000/myecho2
# docker run localhost:5000/myecho2
This container works!
```

### 9.7.2. Pushing containers to the Docker Hub

You can use your Docker Hub credentials to push and pull images from the Docker Hub with the `buildah` command. For this example, replace the username and password (testaccountXX:My00P@sswd) with your own Docker Hub credentials:

```bash
# buildah push --creds testaccountXX:My00P@sswd \
  docker.io/library/myecho2:latest docker://testaccountXX/myecho2:latest
```

As with the private registry, you can then get and run the container from the Docker Hub with the `podman`, `buildah` or `docker` command:

```bash
# podman run docker.io/textaccountXX/myecho2:latest
This container works!
# buildah from docker.io/textaccountXX/myecho2:latest
myecho2-working-container-2
# podman run myecho2-working-container-2
This container works!
```
CHAPTER 10. MONITORING CONTAINERS

This chapter focuses on useful Podman commands that allow you to manage a Podman environment, including determining the health of the container, displaying system and pod information, and monitoring Podman events.

10.1. PERFORMING A HEALTHCHECK ON A CONTAINER

The healthcheck allows you to determine the health or readiness of the process running inside the container. A healthcheck consists of five basic components:

- **Command**
  Podman executes the command inside the target container and waits for the exit code.

- **Retries**
  Defines the number of consecutive failed healthchecks that need to occur before the container is marked as "unhealthy". A successful healthcheck resets the retry counter.

- **Interval**
  Describes the time between running the healthcheck command. Note that small intervals cause your system to spend a lot of time running healthchecks. The large intervals cause struggles with catching timeouts.

- **Start-period**
  Describes the time between when the container starts and when you want to ignore healthcheck failures.

- **Timeout**
  Describes the period of time the healthcheck must complete before being considered unsuccessful.

Healthchecks run inside the container. Healthcheck only make sense if you know what is a health state of the service and can differentiate between a successful and unsuccessful health check.

**Procedure**

1. Define a healthcheck:

```
$ sudo podman run -dt --name hc1 --health-cmd="curl http://localhost || exit 1" --health-interval=0 quay.io/libpod/alpine_nginx:latest
D25ee6f6aaf6e5e12c09e734b1ac675385fe4d4e8b52504dd01a60e1b726e3ed
```

- The **--health-cmd** option sets a healthcheck command for the container.
• The `-health-interval=0` option with 0 value indicates that you want to run healthcheck manually.

2. Run the healthcheck manually:

   ```bash
   $ sudo podman healthcheck run hc1
   Healthy
   ```

3. Optionally, you can check the exit status of last command:

   ```bash
   $ echo $?
   0
   ```

   The "0" value means success.

**Additional resources**

- For more information on the `podman run` command, type `man podman-run`.
- For more information, see article [Monitoring container vitality and availability with Podman](https://example.com) by Brent Baude.

### 10.2. DISPLAYING PODMAN SYSTEM INFORMATION

The `podman system` command allows you to manage the Podman systems. This section provides information on how to display Podman system information.

**Procedure**

- Display Podman system information:
  - To show Podman disk usage, enter:
    ```bash
    $ podman system df
    TYPE            TOTAL   ACTIVE   SIZE    RECLAIMABLE
    Images          3       1        255MB   255MB (100%)
    Containers      1       0        0B      0B (0%)
    Local Volumes   0       0        0B      0B (0%)
    ```
  - To show detailed information on space usage, enter:
    ```bash
    $ podman system df -v
    Images space usage:
    REPOSITORY                            TAG      IMAGE ID       CREATED         SIZE     SHARED
    SIZE   UNIQUE SIZE   CONTAINERS
    docker.io/library/alpine              latest   e7d92c0c71fe   3 months ago    5.86MB   0B 5.86MB 0
    registry.access.redhat.com/ubi8/ubi   latest   8121a9f5303b   6 weeks ago     240MB    0B 240MB 1
    quay.io/libpod/alpine_nginx           latest   3ef70f7291f4   18 months ago  9.21MB   0B 9.21MB 0
    ```
    - Containers space usage:
<table>
<thead>
<tr>
<th>CONTAINER ID</th>
<th>IMAGE</th>
<th>COMMAND</th>
<th>LOCAL VOLUMES</th>
<th>SIZE</th>
<th>CREATED</th>
<th>STATUS</th>
<th>NAMES</th>
</tr>
</thead>
<tbody>
<tr>
<td>ff0167c6c271</td>
<td>8121</td>
<td>/bin/bash</td>
<td>0</td>
<td>0B</td>
<td>10 seconds ago</td>
<td>exited</td>
<td>playTest</td>
</tr>
</tbody>
</table>

Local Volumes space usage:

<table>
<thead>
<tr>
<th>VOLUME NAME</th>
<th>LINKS</th>
<th>SIZE</th>
</tr>
</thead>
</table>

- To display information about the host, current storage stats, and build of Podman, enter:

```bash
$ podman system info
```

**host:**
- arch: amd64
- buildahVersion: 1.15.0
- cgroupVersion: v1
- conmon:
  - package: conmon-2.0.18-1.module+el8.3.0+7084+c16098dd.x86_64
  - path: /usr/bin/conmon
  - version: 'conmon version 2.0.18, commit: 7fd3f71a218f8d3a7202e464252aebe1e942d17eb'
- cpus: 1
- distribution:
  - distribution: "rhel"
  - version: "8.3"
- 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: 4.18.0-227.el8.x86_64
- linkmode: dynamic
- memFree: 69713920
- memTotal: 1376636928
- ociRuntime:
  - name: runc
  - package: runc-1.0.0-66.rc10.module+el8.3.0+7084+c16098dd.x86_64
  - path: /usr/bin/runc
  - version: 'runc version spec: 1.0.1-dev'
- os: linux
- remoteSocket:
  - path: /run/user/1000/podman/podman.sock
- rootless: true
- slirp4netns:
To remove all unused containers, images and volume data, enter:

```
$ podman system prune
```

WARNING! This will remove:
- all stopped containers
- all stopped pods
The **podman system prune** command removes all unused containers (both dangling and unreferenced), pods and optionally, volumes from local storage.

- Use the **--all** option to delete all unused images. Unused images are dangling images and any image that does not have any containers based on it.

- Use the **--volume** option to prune volumes. By default, volumes are not removed to prevent important data from being deleted if there is currently no container using the volume.

**Additional resources**

- For more information on the **podman system df** command, type `man podman-system-df`.

- For more information on the **podman system info** command, type `man podman-system-info`.

- For more information on the **podman system prune** command, type `man podman-system-prune`.

### 10.3. PODMAN EVENT TYPES

You can monitor events that occur in Podman. Several event types exist and each event type reports different statuses.

The **container** event type reports the following statuses:

- attach
- checkpoint
- cleanup
- commit
- create
- exec
- export
- import
- init
- kill
- mount
- pause
- prune
- remove
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
10.4. MONITORING PODMAN EVENTS

You can monitor and print events that occur in Podman. Each event will include a timestamp, a type, a status, name (if applicable), and image (if applicable).

Procedure

- Show Podman events:
  - To show all Podman events, enter:
    ```bash
    $ podman events
    2020-05-14 10:33:42.312377447 -0600 CST container create 34503c192940 (image=registry.access.redhat.com/ubi8/ubi:latest, name=keen_colden)
    2020-05-14 10:33:46.958768077 -0600 CST container init 34503c192940 (image=registry.access.redhat.com/ubi8/ubi:latest, name=keen_colden)
    2020-05-14 10:33:46.973661968 -0600 CST container start 34503c192940 (image=registry.access.redhat.com/ubi8/ubi:latest, name=keen_colden)
    2020-05-14 10:33:50.833761479 -0600 CST container stop 34503c192940 (image=registry.access.redhat.com/ubi8/ubi:latest, name=keen_colden)
    2020-05-14 10:33:51.047104966 -0600 CST container cleanup 34503c192940 (image=registry.access.redhat.com/ubi8/ubi:latest, name=keen_colden)
    ```
  - To exit logging, press CTRL+c.
  - To show only Podman create events, enter:
    ```bash
    $ podman events --filter event=create
    2020-05-14 10:36:01.375685062 -0600 CST container create 20dc581f6fbf (image=registry.access.redhat.com/ubi8/ubi:latest)
    2019-03-02 10:36:08.561188337 -0600 CST container create 58e7e002344c (image=registry.access.redhat.com/ubi8/ubi-minimal:latest)
    2020-05-14 10:33:50.833761479 -0600 CST container stop 34503c192940 (image=registry.access.redhat.com/ubi8/ubi:latest, name=keen_colden)
    2020-05-14 10:33:51.047104966 -0600 CST container cleanup 34503c192940 (image=registry.access.redhat.com/ubi8/ubi:latest, name=keen_colden)
    ```

Additional resources

- For more information on the `podman events` command, type `man podman-events`.

For more information on the `podman events` command, type `man podman-events`.
CHAPTER 11. USING THE CONTAINER-TOOLS CLI

11.1. PODMAN

The podman command (which stands for Pod Manager) lets you run containers as standalone entities, without requiring that Kubernetes, the Docker runtime, or any other container runtime be involved. It is a tool that can act as a replacement for the docker command, implementing the same command-line syntax, while it adds even more container management features. The podman features include:

- **Based on the Docker interface**: Because podman syntax mirrors the docker command, transitioning to podman should be easy for those familiar with docker.

- **Managing containers and images**: Both Docker- and OCI-compatible container images can be used with podman to:
  - Run, stop and restart containers
  - Create and manage container images (push, commit, configure, build, and so on)

- **Managing pods**: Besides running individual containers, podman can run a set of containers grouped in a pod. A pod is the smallest container unit that Kubernetes manages.

- **Working with no runtime**: No runtime environment is used by podman to work with containers.

Here are a few implementation features of Podman you should know about:

- 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, however. Eventually those features will be able to share containers.

- The podman command, like the docker command, can build container images from a Dockerfile.

- The podman command can be a useful troubleshooting tool when the CRI-O service is unavailable.

- 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.

- To interact programmatically with podman, you can use the Podman v2.0 RESTful API, it works in both a rootful and a rootless environment. For more information, see chapter Using the container tools API.

11.1.1. Using podman commands

If you are used to using the docker command to work with containers, you will find most of the features and options match those of podman. Table 1 shows a list of commands you can use with podman (type podman -h to see this list):

Table 11.1. Commands supported by podman
<table>
<thead>
<tr>
<th>podman command</th>
<th>Description</th>
<th>podman command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attach</td>
<td>Attach to a running container</td>
<td>commit</td>
<td>Create new image from changed container</td>
</tr>
<tr>
<td>build</td>
<td>Build an image using Dockerfile instructions</td>
<td>create</td>
<td>Create, but do not start, a container</td>
</tr>
<tr>
<td>diff</td>
<td>Inspect changes on container’s filesystems</td>
<td>exec</td>
<td>Run a process in a running container</td>
</tr>
<tr>
<td>export</td>
<td>Export container’s filesystem contents as a tar archive</td>
<td>help, h</td>
<td>Shows a list of commands or help for one command</td>
</tr>
<tr>
<td>history</td>
<td>Show history of a specified image</td>
<td>images</td>
<td>List images in local storage</td>
</tr>
<tr>
<td>import</td>
<td>Import a tarball to create a filesystem image</td>
<td>info</td>
<td>Display system information</td>
</tr>
<tr>
<td>inspect</td>
<td>Display the configuration of a container or image</td>
<td>kill</td>
<td>Send a specific signal to one or more running containers</td>
</tr>
<tr>
<td>load</td>
<td>Load an image from an archive</td>
<td>login</td>
<td>Login to a container registry</td>
</tr>
<tr>
<td>logout</td>
<td>Logout of a container registry</td>
<td>logs</td>
<td>Fetch the logs of a container</td>
</tr>
<tr>
<td>mount</td>
<td>Mount a working container’s root filesystem</td>
<td>pause</td>
<td>Pauses all the processes in one or more containers</td>
</tr>
<tr>
<td>ps</td>
<td>List containers</td>
<td>port</td>
<td>List port mappings or a specific mapping for the container</td>
</tr>
<tr>
<td>pull</td>
<td>Pull an image from a registry</td>
<td>push</td>
<td>Push an image to a specified destination</td>
</tr>
<tr>
<td>restart</td>
<td>Restart one or more containers</td>
<td>rm</td>
<td>Remove one or more containers from host. Add <code>-f</code> if running.</td>
</tr>
<tr>
<td>rmi</td>
<td>removes one or more images from local storage</td>
<td>run</td>
<td>run a command in a new container</td>
</tr>
</tbody>
</table>
11.1.2. Creating SELinux policies for containers

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

11.1.3. Using podman with MPI

You can use Podman with Open MPI (Message Passing Interface) to run containers in a High Performance Computing (HPC) environment.

The example is based on the ring.c program taken from Open MPI. In this example, a value is passed around by all processes in a ring-like fashion. Each time the message passes rank 0, the value is decremented. When each process receives the 0 message, it passes it on to the next process and then quits. By passing the 0 first, every process gets the 0 message and can quit normally.

Procedure

1. Install Open MPI:

   $ sudo yum install openmpi

2. To activate the environment modules, type:

   $ . /etc/profile.d/modules.sh

3. Load the mpi/openmpi-x86_64 module:

   $ module load mpi/openmpi-x86_64
Optionally, to automatically load `mpi/openmpi-x86_64` module, add this line to the `.bashrc` file:

```bash
$ echo "module load mpi/openmpi-x86_64" >> .bashrc
```

4. To combine `mpirun` and `podman`, create a container with the following definition:

```bash
$ cat Containerfile
FROM registry.access.redhat.com/ubi8/ubi
RUN yum -y install openmpi-devel wget && \
    yum clean all
RUN wget https://raw.githubusercontent.com/open-mpi/ompi/master/test/simple/ring.c && \
    /usr/lib64/openmpi/bin/mpicc ring.c -o /home/ring && \
    rm -f ring.c
```

5. Build the container:

```bash
$ podman build --tag=mpi-ring .
```

6. Start the container. On a system with 4 CPUs this command starts 4 containers:

```bash
$ mpirun \
   --mca orte_tmpdir_base /tmp/podman-mpirun \
   podman run --env-host \
   -v /tmp/podman-mpirun:/tmp/podman-mpirun \
   --userns=keep-id \
   --net=host --pid=host --ipc=host \
   mpi-ring /home/ring

Rank 2 has cleared MPI_Init
Rank 2 has completed ring
Rank 2 has completed MPI_Barrier
Rank 3 has cleared MPI_Init
Rank 3 has completed ring
Rank 3 has completed MPI_Barrier
Rank 1 has cleared MPI_Init
Rank 1 has completed ring
Rank 1 has completed MPI_Barrier
Rank 0 has cleared MPI_Init
Rank 0 has completed ring
Rank 0 has completed MPI_Barrier
```

As a result, `mpirun` starts up 4 Podman containers and each container is running one instance of the `ring` binary. All 4 processes are communicating over MPI with each other.

The following `mpirun` options are used to start the container:

- `--mca orte_tmpdir_base /tmp/podman-mpirun` line tells Open MPI to create all its temporary files in `/tmp/podman-mpirun` and not in `/tmp`. If using more than one node this directory will be named differently on other nodes. This requires mounting the complete `/tmp` directory into the container which is more complicated.

The `mpirun` command specifies the command to start, the `podman` command. The following `podman` options are used to start the container:
run command runs a container.

--env-host option copies all environment variables from the host into the container.

-v /tmp/podman-mpirun:/tmp/podman-mpirun line tells Podman to mount the directory where Open MPI creates its temporary directories and files to be available in the container.

--userns=keep-id line ensures the user ID mapping inside and outside the container.

--net=host --pid=host --ipc=host line sets the same network, PID and IPC namespaces.

mpi-ring is the name of the container.

/home/ring is the MPI program in the container.

For more information, see the article Podman in HPC environments by Adrian Reber.

11.1.4. Creating and restoring container checkpoints

Checkpoint/Restore In Userspace (CRIU) is a software that enables you to set a checkpoint on a running container or an individual application and store its state to disk. You can use data saved to restore the container after a reboot at the same point in time it was checkpointed.

11.1.4.1. Creating and restoring a container checkpoint locally

This example is based on a Python based web server which returns a single integer which is incremented after each request.

Procedure

1. Create a Python based server:

```python
# cat counter.py
#!/usr/bin/python3
import http.server

counter = 0
class handler(http.server.BaseHTTPRequestHandler):
    def do_GET(s):
        global counter
        s.send_response(200)
        s.send_header('Content-type', 'text/html')
        s.end_headers()
        s.wfile.write(b'%d
' % counter)
        counter += 1

server = http.server.HTTPServer(('', 8088), handler)
server.serve_forever()
```

2. Create a container with the following definition:

```bash
# cat Containerfile
```
FROM registry.access.redhat.com/ubi8/ubi
COPY counter.py /home/counter.py
RUN useradd -ms /bin/bash counter
RUN yum -y install python3 && chmod 755 /home/counter.py
USER counter
ENTRYPOINT /home/counter.py

The container is based on the Universal Base Image (UBI 8) and uses a Python based server.

3. Build the container:

```
# podman build . --tag counter
```

Files `counter.py` and `Containerfile` are the input for the container build process (`podman build`). The built image is stored locally and tagged with the tag `counter`.

4. Start the container as root:

```
# podman run --name criu-test --detach counter
```

5. To list all running containers, enter:

```
# podman ps
CONTAINER ID  IMAGE  COMMAND  CREATED   STATUS  PORTS NAMES
e4f82fd84d48  localhost/counter:latest  5 seconds ago  Up 4 seconds ago  criu-test
```

6. Display IP address of the container:

```
# podman inspect criu-test --format "{{.NetworkSettings.IPAddress}}"
10.88.0.247
```

7. Send requests to the container:

```
# curl 10.88.0.247:8080
0
# curl 10.88.0.247:8080
1
```

8. Create a checkpoint for the container:

```
# podman container checkpoint criu-test
```

9. Reboot the system.

10. Restore the container:

```
# podman container restore --keep criu-test
```

11. Send requests to the container:

```
The result now does not start at 0 again, but continues at the previous value.

This way you can easily save the complete container state through a reboot.

For more information, see the article Adding checkpoint/restore support to Podman by Adrian Reber.

11.1.4.2. Reducing startup time using container restore

You can use container migration to reduce startup time of containers which require a certain time to initialize. Using a checkpoint, you can restore the container multiple times on the same host or on different hosts. This example is based on the container from the Creating and restoring a container checkpoint locally section.

Procedure

1. Create a checkpoint of the container, and export the checkpoint image to a tar.gz file:

   ```bash
   # podman container checkpoint criu-test --export /tmp/chkpt.tar.gz
   ```

2. Restore the container from the tar.gz file:

   ```bash
   # podman container restore --import /tmp/chkpt.tar.gz --name counter1
   # podman container restore --import /tmp/chkpt.tar.gz --name counter2
   # podman container restore --import /tmp/chkpt.tar.gz --name counter3
   ```

   The --name (-n) option specifies a new name for containers restored from the exported checkpoint.

3. Display ID and name of each container:

   ```bash
   # podman ps -a --format "{{.ID}} {{.Names}}"
   a8b2e50d463c counter3
   faabc5c27362 counter2
   2ce648af11e5 counter1
   ```

4. Display IP address of each container:

   ```bash
   # podman inspect counter1 --format "{{.NetworkSettings.IPAddress}}"
   10.88.0.248
   # podman inspect counter2 --format "{{.NetworkSettings.IPAddress}}"
   10.88.0.249
   # podman inspect counter3 --format "{{.NetworkSettings.IPAddress}}"
   10.88.0.250
   ```

5. Send requests to each container:
# curl 10.88.0.248:8080
4
# curl 10.88.0.249:8080
4
# curl 10.88.0.250:8080
4

Note, that the result is 4 in all cases, because you are working with different containers restored from the same checkpoint.

Using this approach, you can quickly start up stateful replicas of the initially checkpointed container.

For more information, see the article Container migration with Podman on RHEL by Adrian Reber.

11.1.4.3. Migrating containers among systems

This procedure shows the migration of running containers from one system to another, without losing the state of the applications running in the container. This example is based on the container from the Creating and restoring a container checkpoint locally section tagged with counter.

Prerequisites

The following steps are not necessary if the container is pushed to a registry as Podman will automatically download the container from a registry if it is not available locally. This example does not use a registry, you have to export previously built and tagged container (see Creating and restoring a container checkpoint locally section) locally and import the container on the destination system of this migration.

- Export previously built container:

  # podman save --output counter.tar counter

- Copy exported container image to the destination system (other_host):

  # scp counter.tar other_host

- Import exported container on the destination system:

  # ssh other_host podman load --input counter.tar

Now the destination system of this container migration has the same container image stored in its local container storage.

Procedure

1. Start the container as root:

   # podman run --name criu-test --detach counter

2. Display IP address of the container:

   # podman inspect criu-test --format "{{.NetworkSettings.IPAddress}}"
   10.88.0.247
3. Send requests to the container:

   # curl 10.88.0.247:8080
   0
   # curl 10.88.0.247:8080
   1

4. Create a checkpoint of the container, and export the checkpoint image to a `tar.gz` file:

   # podman container checkpoint criu-test --export /tmp/chkpt.tar.gz

5. Copy the checkpoint archive to the destination host:

   # scp /tmp/chkpt.tar.gz other_host:/tmp/

6. Restore the checkpoint on the destination host (`other_host`):

   # podman container restore --import /tmp/chkpt.tar.gz

7. Send a request to the container on the destination host (`other_host`):

   # curl 10.88.0.247:8080
   2

As a result, the stateful container has been migrated from one system to another without losing its state.

For more information, see the article Container migration with Podman on RHEL by Adrian Reber.

### 11.2. RUNC

The “runC” container runtime is a lightweight, portable implementation of the Open Container Initiative (OCI) container runtime specification. The runC unites a lot of the low-level features that make running containers possible. It shares a lot of low-level code with Docker but it is not dependent on any of the components of the Docker platform.

runC supports Linux namespaces, live migration, and has portable performance profiles. It also provides full support for Linux security features such as SELinux, control groups (cgroups), seccomp, and others. You can build and run images with runC, or you can run OCI-compatible images with runC.

#### 11.2.1. Running containers with runc

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

To create a bundle, run:

```
$ runc spec
```

This command creates a `config.json` file that only contains a bare-bones structure that you will need to edit. Most importantly, you will need to change the `args` parameter to identify the executable to run. By default, `args` is set to `sh`.
As an example, you can download the Red Hat Enterprise Linux base image (`ubi8/ubi`) using podman then export it, create a new bundle for it with runc, and edit the `config.json` file to point to that image. You can then create the container image and run an instance of that image with runc. Use the following commands:

```
# podman pull registry.redhat.io/ubi8/ubi
# podman export $(podman create registry.redhat.io/ubi8/ubi) > rhel.tar
# mkdir -p rhel-runc/rootfs
# tar -C rhel-runc/rootfs -xf rhel.tar
# runc spec -b rhel-runc
# vi rhel-runc/config.json  Change any setting you like
# runc create -b rhel-runc/ rhel-container
# runc start rhel-container
```

In this example, the name of the container instance is `rhel-container`. Running that container, by default, starts a shell, so you can begin looking around and running commands from inside that container. Type `exit` when you are done.

The name of a container instance must be unique on the host. To start a new instance of a container:

```
# runc start <container_name>
```

You can provide the bundle directory using the `-b` option. By default, the value for the bundle is the current directory.

You will need root privileges to start containers with runc. To see all commands available to runc and their usage, run `runc --help`.

### 11.3. SKOPEO

With the `skopeo` command, you can work with container images from registries without using the docker daemon or the `docker` command. Registries can include the Docker Registry, your own local registries, Red Hat Quay or OpenShift registries. Activities you can do with skopeo include:

- **inspect**: The output of a `skopeo inspect` command is similar to what you see from a `docker inspect` command: low-level information about the container image. That output can be in json format (default) or raw format (using the `--raw` option).

- **copy**: With `skopeo copy` you can copy a container image from a registry to another registry or to a local directory.

- **layers**: The `skopeo layers` command lets you download the layers associated with images so that they are stored as tarballs and associated manifest files in a local directory.

Like the `buildah` command and other tools that rely on the containers/image library, the `skopeo` command can work with images from container storage areas other than those associated with Docker. Available transports to other types of container storage include: containers-storage (for images stored by `buildah` and CRI-O), ostree (for atomic and system containers), oci (for content stored in an OCI-compliant directory), and others. See the `skopeo man page` for details.
To try out skopeo, you could set up a local registry, then run the commands that follow to inspect, copy, and download image layers. If you want to follow along with the examples, start by doing the following:

- Install a local registry (such as Red Hat Quay. Container registry software available in the docker-distribution package for RHEL 7, is not available for RHEL 8.
- Pull the latest RHEL image to your local system (podman pull ubi8/ubi).
- Retag the RHEL image and push it to your local registry as follows:
  
  ```
  # podman tag ubi8/ubi localhost/myubi8
  # podman push localhost/myubi8
  ```

  The rest of this section describes how to inspect, copy and get layers from the RHEL image.

  **NOTE**

  The **skopeo** tool by default requires a TLS connection. It fails when trying to use an unencrypted connection. To override the default and use an http registry, prepend http: to the `<registry>/<image>` string.

### 11.3.1. Inspecting container images with skopeo

When you inspect a container image from a registry, you need to identify the container format (such as docker), the location of the registry (such as docker.io or localhost), and the repository/image (such as ubi8/ubi).

The following example inspects the mariadb container image from the Docker Registry:

```
# skopeo inspect docker://docker.io/library/mariadb
{
  "Name": "docker.io/library/mariadb",
  "Tag": "latest",
  "Digest": "sha256:d3f56b143b62690b400ef42e876e628eb5e488d2d0d2a35d6438a4aa841d89c4",
  "RepoTags": ["10.0.15",
               "10.0.16",
               "10.0.17",
               "10.0.19",
               ...
               "Created": "2018-06-10T01:53:48.812217692Z",
               "DockerVersion": "1.10.3",
               "Labels": {},
               "Architecture": "amd64",
               "Os": "linux",
               "Layers": [ ...
               ...
  }
```

Assuming you pushed a container image tagged **localhost/myubi8** to a container registry running on your local system, the following command inspects that image:

```
# skopeo inspect docker://localhost/myubi8
{
  "Name": "localhost/myubi8",
  "Tag": "latest",
  ```
11.3.2. Copying container images with skopeo

This command copies the myubi8 container image from a local registry into a directory on the local system:

```
# skopeo copy docker://localhost/myubi8 dir:/root/test/
```

The result of the `skopeo copy` command is a tarball (16d*.tar) and a manifest.json file representing the image being copied to the directory you identified. If there were multiple layers, there would be multiple tarballs. The `skopeo copy` command can also copy images to another registry. If you need to provide a signature to write to the destination registry, you can do that by adding a `--sign-by=` option to the command line, followed by the required key-id.

11.3.3. Getting image layers with skopeo

The `skopeo layers` command is similar to `skopeo copy`, with the difference being that the `copy` option can copy an image to another registry or to a local directory, while the `layers` option just drops the layers (tarballs and manifest.json file) in the current directory. For example

```
# skopeo layers docker://localhost/myubi8
```

The result of the `skopeo layers` command is a tarball (16d*.tar) and a manifest.json file representing the image being copied to the directory you identified. If there were multiple layers, there would be multiple tarballs. The `skopeo layers` command can also copy images to another registry. If you need to provide a signature to write to the destination registry, you can do that by adding a `--sign-by=` option to the command line, followed by the required key-id.
As you can see from this example, a new directory is created (layers-myubi8-latest-698503105) and, in this case, a single layer tarball and a manifest.json file are copied to that directory.
The new REST based Podman 2.0 API replaces the old remote API for Podman that used the varlink library. The new API works in both a rootful and a rootless environment.

The Podman v2.0 RESTful API consists of the Libpod API providing support for Podman, and Docker-compatible API.

With this new REST API, you can call Podman from platforms such as cURL, Postman, Google’s Advanced REST client, and many others.

### 12.1. ENABLING THE PODMAN API USING SYSTEMD IN ROOT MODE

This procedure shows how to do the following:

1. Use systemd to activate the Podman API socket.
2. Use a Podman client to perform basic commands.

**Prerequisites**

- The `podman-remote` package is installed.

```
# yum install podman-remote
```

**Procedure**

1. Configure the systemd unit file for Podman socket:

   ```
   # cat /usr/lib/systemd/system/podman.socket
   [Unit]
   Description=Podman API Socket
   Documentation=man:podman-api(1)
   
   [Socket]
   ListenStream=%t/podman/podman.sock
   SocketMode=0660
   
   [Install]
   WantedBy=sockets.target
   ```

2. Reload the systemd manager configuration:

   ```
   # systemctl daemon-reload
   ```

3. Start the service immediately:

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

4. To enable the link to `var/lib/docker.sock` using the `docker-podman` package:

   ```
   # yum install podman-docker
   ```
Verification steps

- Display system information of Podman:
  
  ```shell
  # podman-remote info
  ```

- Verify the link:
  
  ```shell
  # 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

- For more information about the Podman 2.0 API, see the Podman v2.0 RESTful API documentation.

- For more examples on how to use Podman 2.0 API, see the A First Look At Podman 2.0 API article by Scott McCarty.

- For yet more examples on how to use Podman 2.0 API, see the Sneak peek: Podman's new REST API article by Tom Sweeney.

12.2. ENABLING THE PODMAN API USING SYSTEMD IN ROOTLESS MODE

This procedure shows how to use systemd to activate the Podman API socket and podman API service.

Prerequisites

- The `podman-remote` package is installed.
  
  ```shell
  # yum install podman-remote
  ```

Procedure

1. Create a rootless socket for Podman:

   ```shell
   $ vim .config/systemd/user/podman.socket
   ```

   ```ini
   [Unit]
   Description=Podman API Socket
   Documentation=man:podman-api(1)
   
   [Socket]
   ListenStream=/home/username/podman.sock
   SocketMode=0660
   
   [Install]
   WantedBy=sockets.target
   ```

2. Create a rootless service for Podman:

   ```shell
   $ vim .config/systemd/user/podman.service
   ```
The **After** line in the [Unit] sections defines the dependency on the podman.socket unit file. The podman.socket unit started up before the configured podman.service.

3. Reload the systemd manager configuration:

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

4. Enable and start the service immediately:

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

5. To enable programs using Docker to interact with the rootless Podman socket:

   ```
   $ export DOCKER_HOST=unix:///var/run/<username>/podman.sock
   ```

**Verification steps**

- Display system information of Podman:

  ```
  $ podman-remote info
  ```

**Additional resources**

- For more information about the Podman 2.0 API, see the [Podman v2.0 RESTful API documentation](#).

- For more examples on how to use Podman 2.0 API, see the [A First Look At Podman 2.0 API](#) article by Scott McCarty.

- For yet more examples on how to use Podman 2.0 API, see the [Sneak peek: Podman’s new REST API](#) article by Tom Sweeney.

- For examples of using the Podman 2.0 API with Python and Bash, see the [Exploring Podman RESTful API using Python and Bash](#) article by Jhon Honce.
12.3. RUNNING THE PODMAN API MANUALLY

This procedure describes how to run the Podman API. This is useful for debugging API calls, especially when using the Docker compatibility layer.

Prerequisites

- The podman-remote package is installed.

  # yum install podman-remote

Procedure

1. Run the service for the REST API:

   # podman system service -t 0 --log-level=debug

   The value of 0 means no timeout. The default endpoint for a rootful service is unix:/run/podman/podman.sock.

   - The --log-level <level> option sets the logging level. The standard logging levels are debug, info, warn, error, fatal, and panic.

2. In another terminal, display system information of Podman. The podman-remote command, unlike the regular podman command, communicates through the Podman socket:

   # podman-remote info

3. To troubleshoot the Podman API and display request and responses, use the curl command. To get the information about the Podman installation on the Linux server in JSON format:

   # curl -s --unix-socket /run/podman/podman.sock http://d/v1.0.0/libpod/info | jq

   "host": {
       "arch": "amd64",
       "buildahVersion": "1.15.0",
       "cgroupVersion": "v1",
       "conmon": {
           "package": "conmon-2.0.18-1.module+el8.3.0+7084+c16098dd.x86_64",
           "path": "/usr/bin/conmon",
           "version": "conmon version 2.0.18, commit: 7fd3f71a218f8d3a7202e464252aeb1e942d17eb"
       },
       ...
   }
A jq utility is a command-line JSON processor.

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

```bash
# curl -XPOST --unix-socket /run/podman/podman.sock -v 'http://d/v1.0.0/images/create?fromImage=registry.access.redhat.com%2Fubi8%2Fubi'
* Trying /run/podman/podman.sock...
* Connected to d (/run/podman/podman.sock) port 80 (#0)
> POST /v1.0.0/images/create?fromImage=registry.access.redhat.com%2Fubi8%2Fubi HTTP/1.1
> Host: d
> User-Agent: curl/7.61.1
> Accept: /
> HTTP/1.1 200 OK
< Content-Type: application/json
< Date: Tue, 20 Oct 2020 13:58:37 GMT
< Content-Length: 231
<
{"status":"pulling image () from registry.access.redhat.com/ubi8/ubi:latest, registry.redhat.io/ubi8/ubi:latest","error":null,"progress":null,"progressDetail":null,"id":"ecbc6f53bba0d1923ca9e92b3f747da8353a070fccbae93625bd8b47dbee772e"}
* Connection #0 to host d left intact
```

5. Display the pulled image:

```bash
# curl --unix-socket /run/podman/podman.sock -v 'http://d/v1.0.0/libpod/images/json' | jq
* Trying /run/podman/podman.sock...
* User-Agent: curl/7.61.1
> GET /v1.0.0/libpod/images/json HTTP/1.1
> Host: d
> Accept: */*
< HTTP/1.1 200 OK
< Content-Type: application/json
< Date: Tue, 20 Oct 2020 13:59:55 GMT
< Transfer-Encoding: chunked
< 
{"Id":"ecbc6f53bba0d1923ca9e92b3f747da8353a070fccbae93625bd8b47dbee772e","RepoTags":null,"Created":null,"Size":null,"Labels":null,"Summary":null,"Names":null}
<
* Connection #0 to host d left intact
```

Additional resources

For more information about the Podman 2.0 API, see the Podman v2.0 RESTful API.
For more information about the Podman 2.0 API, see the Podman v2.0 RESTful API documentation.

For yet more examples on how to use Podman 2.0 API, see the Sneak peek: Podman's new REST API article by Tom Sweeney.

For examples of using the Podman 2.0 API with Python and Bash, see the Exploring Podman RESTful API using Python and Bash article by Jhon Honce.

For more information on podman system service command, see the podman-system-service man page.
CHAPTER 13. ADDITIONAL RESOURCES

- Buildah - a tool for building OCI container images
- Podman - a tool for running and managing containers
- Skopeo - a tool for copying and inspecting container images