Red Hat Enterprise Linux 8

Building, running, and managing containers

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

Last Updated: 2021-07-13

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

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

Legal Notice

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.

Table of Contents

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. RHEL implements Linux containers using core technologies such as:

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

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

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

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

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

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

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

1.1. RUNNING CONTAINERS WITHOUT DOCKER

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

NOTE

Docker is not supported in RHEL 8.

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

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

- Running in rootless mode - rootless containers are much more secure, as they run without any added privileges
- No daemon required - these tools have much lower resource requirements at idle, since when you are not running containers, Podman is not running versus having a daemon always running
- Native systemd integration - Podman allows you to create systemd unit files and run containers as system services

If you still want to use Docker in RHEL, 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 in Podman, you can set up an alias so that typing **docker** causes **podman** to run.

Installing the **podman-docker** package sets up this alias. As a result, every time you run a **docker** command, it actually runs **podman** command.

Podman also supports the Docker Socket API, so the **podman-docker** package also sets up a link between **/var/run/docker.sock** and **/var/run/podman/podman.sock**. As a result, you can continue to run your Docker API commands with **docker-py** and **docker-compose** tools without requiring the Docker daemon. Podman will service the requests.

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

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

Additional resources

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

1.3. GETTING CONTAINER TOOLS

This procedure shows how you can install the **container-tools** module which contains the Podman, Buildah, Skopeo, and runc tools.

Procedure

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

```
# subscription-manager register
Registering to: subscription.rhsm.redhat.com:443/subscription
```

```
Username: ********
Password: **********
```

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

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

   - To subscribe to RHEL by Pool ID:

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

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

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

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

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

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

1.4. SETTING UP ROOTLESS CONTAINERS

Running the container tools such as Podman, Skopeo, or Buildah as a user with superuser privileges (root user) is the best way to ensure that your containers have full access to any feature available on your system. However, with the feature called "Rootless Containers" generally available as of 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. As a result, regular users can make requests through their containers that can harm the system. By setting up rootless container users, system administrators prevent potentially damaging container activities from regular users, while still allowing those users to safely run most container features under their own accounts.

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

Prerequisites

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

Procedure

1. Install RHEL.
2. Install the **podman** and **slirp4netns** packages:

   ```
   # yum install slirp4netns podman -y
   ```

3. Create a new user account:

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

   The user is automatically configured to be able to use rootless Podman. The **useradd** command automatically sets the range of accessible user and group IDs automatically in the **/etc/subuid** and **/etc/subgid** files.

4. Connect to the user:

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

   NOTE

   Do not use **su** or **su -** commands because these commands do not set the correct environment variables. Use your Red Hat Customer Portal credentials.

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

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

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

   ```
   $ podman run --name=myubi registry.access.redhat.com/ubi8/ubi cat
     /etc/os-release
   NAME="Red Hat Enterprise Linux"
   VERSION="8.4 (Ootpa)"
   ```

Additional resources

- Rootless containers with Podman: The basics

1.5. UPGRADING TO ROOTLESS CONTAINERS

This section shows how to upgrade to rootless containers from RHEL 7. You must configure user and group IDs manually.

Here are some things to consider when upgrading to rootless containers from RHEL 7:

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

Prerequisites

- The user account has been created.

Procedure

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

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

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

2. Optional: Check that the UIDs and GIDs are set properly:

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

1.6. SPECIAL CONSIDERATIONS FOR ROOTLESS CONTAINERS

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

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

  ```
  $ 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 cannot access a port numbered less than 1024. Inside the rootless container namespace it can, for example, start a service that exposes port 80 from an httpd service from the container, but it is not accessible outside of the namespace:

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

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

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

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

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

Additional resources

- Shortcomings of Rootless Podman

CHAPTER 2. TYPES OF CONTAINER IMAGES

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

There are two types of container images:

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

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

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

2.1. GENERAL CHARACTERISTICS OF RHEL CONTAINER IMAGES

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

In general, RHEL container images are:

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

2.2. CHARACTERISTICS OF UBI IMAGES

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

Following characteristics apply to UBI images:

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

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

Additional resources

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

2.3. UNDERSTANDING THE UBI STANDARD IMAGES

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

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

2.4. UNDERSTANDING THE UBI INIT IMAGES

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

NOTE

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

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

2.5. UNDERSTANDING THE UBI MINIMAL IMAGES

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

The key features of UBI minimal images include:

- **Small size**: Minimal images are about 92M on disk and 32M, when compressed. This makes it less than half the size of the standard images.
- **Software installation (microdnf)**: Instead of including the fully-developed **yum** facility for working with software repositories and RPM software packages, the minimal images includes the **microdnf** utility. The **microdnf** is a scaled-down version of **dnf** allowing you to enable and disable repositories, remove and update packages, and clean out cache after packages have been installed.
- **Based on RHEL packaging**: Minimal images incorporate regular RHEL software RPM packages, with a few features removed. Minimal images do not include initialization and service management system, such as systemd or System V init, Python run-time environment, and some shell utilities. You can rely on RHEL repositories for building your images, while carrying the smallest possible amount of overhead.
- **Modules for microdnf are supported**: Modules used with **microdnf** command let you install multiple versions of the same software, when available. You can use **microdnf module enable**, **microdnf module disable**, and **microdnf module reset** to enable, disable, and reset a module stream, respectively.
  - For example, to enable the **nodejs:14** module stream inside the UBI minimal container, enter:

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

```
Enabling module streams:
    nodejs:14

Running transaction test...
```

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

2.6. UNDERSTANDING THE UBI MICRO IMAGES

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

2.7. USING THE UBI INIT IMAGES

This procedure shows how to build a container using a **Containerfile** that installs and configures a Web server (**httpd**) to start automatically by the systemd service (**/sbin/init**) when the container is run on a host system. The **podman build** command uses a **Containerfile** if found in the context directory, if it is not found the **podman build** command will use a **Dockerfile**; otherwise any file can be specified with the **--file** option. The available commands that are usable inside a **Containerfile** and a **Dockerfile** are equivalent.

Procedure

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

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

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

2. Build the container:

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

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

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

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

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

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

NOTE

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

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

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

```
$ sudo sysctl net.ipv4.ip_unprivileged_port_start=80
```

5. Check that the container is running:

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

6. Test the web server:

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

Additional resources

- Shortcomings of Rootless Podman

2.8. USING THE UBI MICRO IMAGES

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

Prerequisites

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

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

Procedure

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

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

2. Mount a working container root filesystem:

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

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

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

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

```
# buildah umount $microcontainer
```

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

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

Verification steps

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

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

CHAPTER 3. WORKING WITH CONTAINER IMAGES

This explains how to work with container images. How to pull the image, inspect, tag, save, load, redistribute, define the image signature.

3.1. CONFIGURING CONTAINER REGISTRIES

Using the **podman search** command you can search selected container registries for images. 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**. As a root user, edit the **/etc/containers/registries.conf** file to change the default, system-wide search settings.

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

```
[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 the **[registries.search]** section in the given order. In this case, the command searches the requested image in **registry.access.redhat.com**, **registry.redhat.io**, and **docker.io** in this order.
- The **[registries.insecure]** section defines registries that do not use TLS encryption for the communication with the registry, what makes them insecure.
- The **[registries.block]** section defines registries the local system is not allowed to access.

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 **registries = key**, you must separate those registries by commas.
- You can identify registries by either IP address or host name.
- If the registry uses a non-standard port - other than the TCP ports **443** for encrypted and **80** for unencrypted connections, 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 **registries.search** list.

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

```
[registries.insecure]
registries = ['insecure-registry.example.com']
```

3.2. SEARCHING FOR CONTAINER IMAGES

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

Prerequisites

- The registry is configured.

Procedure

1. Authenticate to the registry:

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

2. Search for the image:

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

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

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

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

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

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

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

3.3. 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>
```

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

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

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

Procedure

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

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

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

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

Verification steps

- To list all images pulled to your local system:

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

3.4. PULLING AN IMAGE USING PODMAN

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 that do not allow unknown or anonymous users to create accounts with arbitrary names.

For example, a user wants to pull the **example** container image from **example.registry.com** registry. If **example.registry.com** is not first in the search list, an attacker could place a different **example** image at a registry earlier in the search list. The user would accidentally pull and run the attacker image rather than the intended content.

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:

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

The registries that Red Hat provides are:

- registry.redhat.io (requiring authentication)

- registry.access.redhat.com (requires no authentication)
- 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 using your RHEL Subscription credentials.

3.5. LISTING IMAGES

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

Prerequisites

- A pulled image is available on the local system.

Procedure

- List all images in the local storage:

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

3.6. INSPECTING LOCAL IMAGES

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

Prerequisites

- A pulled image is available on the local system.

Procedure

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

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

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

3.7. INSPECTING REMOTE IMAGES

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

Procedure

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

```
# skopeo inspect docker://registry.redhat.io/ubi8/ubi-init
{
    "Name": "registry.redhat.io/ubi8/ubi8-init",
    "Digest": "sha256:c6d1e50ab...",
    "RepoTags": [
        "8.2-13-source",
        "8.0-15",
        "8.1-28",
        ...
        "latest"
    ],
   "Created": "2020-12-10T07:16:37.250312Z",
    "DockerVersion": "1.13.1",
    "Labels": {
        "architecture": "x86_64",
        "build-date": "2020-12-10T07:16:11.378348",
        "com.redhat.build-host": "cpt-1007.osbs.prod.upshift.rdu2.redhat.com",
        "com.redhat.component": "ubi8-init-container",
        "com.redhat.license_terms": "https://www.redhat.com/en/about/red-hat-end-user-
license-agreements#UBI",
        "description": "The Universal Base Image Init is designed to run an init system as PID 1
for running multi-services inside a container
        ...
```

3.8. TAGGING IMAGES

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

Prerequisites

- A pulled image is available on the local system.

Procedure

1. List all images:

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

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

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

   - The image ID:

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

     Both commands give you the same result.
3. List all images:

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

   Notice that the default tag is **latest** for both images. You can see all the image names are assigned to the single image ID 3269c37eae33.
4. Add the **8.4** tag to the **registry.redhat.io/ubi8/ubi** image using either:
   - The image name:

     ```
     $ podman tag registry.redhat.io/ubi8/ubi myubi:8.4
     ```

   - The image ID:

     ```
     $ podman tag 3269c37eae33 myubi:8.4
     ```

     Both commands give you the same result.
5. List all images:

   ```
   $ podman images
   REPOSITORY                   TAG     IMAGE ID      CREATED        SIZE
   registry.redhat.io/ubi8/ubi  latest  3269c37eae33  2 months ago  208 MB
   localhost/myubi            latest  3269c37eae33  2 months ago  208 MB
   localhost/myubi            8.4     3269c37eae33  2 months ago  208 MB
   ```

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

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

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

- by name (**localhost/myubi:8.4**)

3.9. SAVING AND LOADING IMAGES

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

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

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

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

Prerequisites

- A pulled image is available on the local system.

Procedure

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

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

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

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

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

2. Check the file type of **myrsyslog.tar**:

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

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

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

3.10. REDISTRIBUTING UBI IMAGES

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

Prerequisites

- A pulled image is available on the local system.

Procedure

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

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

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

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

IMPORTANT

While there are few restrictions on how you use these images, there are some restrictions about how you can refer to them. For example, you cannot call those images Red Hat certified or Red Hat supported unless you certify it through the Red Hat Partner Connect Program, either with Red Hat Container Certification or Red Hat OpenShift Operator Certification.

3.11. DEFINING THE IMAGE SIGNATURE VERIFICATION POLICY

Red Hat delivers signatures for the images in the Red Hat Container Registry. The YAML files in the **/etc/containers/registries.d/** directory and the **/etc/containers/policy.json** file define the signature verification policy when running as root.

The trust policy in the **/etc/containers/policy.json** file describes a registry scope (registry and or or repository) for the trust.

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

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 distributes 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:

```
# 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:

```
# 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:

```
        "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"
                }
            ]
        },
```

You can see that sections **"registry.access.redhat.com"** and **"registry.redhat.io"** are added.

4. Create the **/etc/containers/registries.d/registry.access.redhat.com.yaml** file with the following content to identify the signature store for container images from the registry.access.redhat.com registry:

```
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:

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

6. To display the trust configuration, enter:

```
# 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:

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

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

```
# cat /etc/containers/policy.json
{
    "default": [
    {
        "type": "reject"
    }
...
}
```

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

9. Pull the minimal Red Hat Universal Base Image 8 (**ubi8-minimal**) image from the registry.access.redhat.com registry:

```
# podman --log-level=debug pull registry.access.redhat.com/ubi8-minimal
....
DEBU[0000] Using registries.d directory /etc/containers/registries.d for sigstore configuration
DEBU[0000]  Using "docker" namespace registry.access.redhat.com
DEBU[0000]   Using https://access.redhat.com/webassets/docker/content/sigstore
...
```

You see that the signature storage address **access.redhat.com/webassets/docker/content/sigstore** matches the address you specified in the **/etc/containers/registries.d/registry.access.redhat.com.yaml**.

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

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

11. 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

- **man podman-image-trust** man page
- Verifying image signing for Red Hat Container Registry .

3.12. REMOVING IMAGES

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

Procedure

1. List all images on your local system:

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

2. List all containers:

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

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

3. Stop the **mysyslog** container:

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

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

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

   - To remove multiple images:

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

   - To remove all images from your system:

     ```
     $ podman rmi -a
     ```

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

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

CHAPTER 4. WORKING WITH CONTAINERS

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

4.1. PODMAN RUN COMMAND

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

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

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

Basic options are:

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

4.2. RUNNING COMMANDS IN A CONTAINER FROM THE HOST

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

Prerequisites

- The Podman tool is installed.

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

Procedure

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

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

2. Optional: List all containers.

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

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

Additional resources

- **man podman-run**

4.3. RUNNING COMMANDS INSIDE THE CONTAINER

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

Prerequisites

- The Podman tool is installed.

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

Procedure

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

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

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

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

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

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

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

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

```
# exit
```

5. Optional: List all containers:

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

You can see that the container is in Exited status.

Additional resources

- **man podman-run**

4.4. LISTING CONTAINERS

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

Prerequisites

- The Podman tool is installed.

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

Procedure

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

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

2. List all containers:

   - To list all running containers:

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

   - To list all containers, running or stopped:

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

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

Additional resources

- **man podman-ps**

4.5. STARTING CONTAINERS

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

Prerequisites

- The Podman tool is installed.

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

- At least one container has been stopped.

Procedure

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

     ```
     $ podman start myubi
     ```

     Alternatively, you can use **podman start 1984555a2c27**.
   - In the interactive mode, use **-a** (**--attach**) and **-t** (**--interactive**) options to work with container bash shell:

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

     Alternatively, you can use **podman start -a -i 1984555a2c27**.
2. Enter **exit** to exit the container and return to the host:

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

Additional resources

- **man podman-start**

4.6. INSPECTING CONTAINERS FROM THE HOST

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

Prerequisites

- The Podman tool is installed.

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

Procedure

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

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

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

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

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

Examples of other items you might want to inspect include:

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

Additional resources

- **man podman-inspect**

4.7. MOUNTING DIRECTORY ON LOCALHOST TO THE CONTAINER

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

Prerequisites

- The Podman tool is installed.

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

Procedure

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

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

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

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

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

Additional resources

- **man podman-run**

4.8. MOUNTING A CONTAINER FILESYSTEM

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

Prerequisites

- The Podman tool is installed.

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

Procedure

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

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

2. Optional: List all containers:

```
# podman ps -a
CONTAINER ID  IMAGE                              COMMAND       CREATED       STATUS
PORTS   NAMES
```

```
c56ef6a256f8  registry.redhat.io/rhel8/rsyslog:latest  /bin/rsyslog.sh  20 minutes ago  Up 20
minutes ago                 mysyslog
```

3. Mount the **mysyslog** container:

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

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

```
# ls
/var/lib/containers/storage/overlay/990b5c6ddcdeed4bde7b245885ce4544c553d108310e2b797
d7be46750894719/merged
bin  boot  dev  etc  home  lib  lib64  lost+found  media  mnt  opt  proc  root  run  sbin  srv  sys
tmp  usr  var
```

5. Display the OS version:

```
# cat
/var/lib/containers/storage/overlay/990b5c6ddcdeed4bde7b245885ce4544c553d108310e2b797
d7be46750894719/merged/etc/os-release
NAME="Red Hat Enterprise Linux"
VERSION="8.3 (Ootpa)"
ID="rhel"
ID_LIKE="fedora"
VERSION_ID="8.3"
```

Additional resources

- **man podman-mount**

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

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

Prerequisites

- The Podman tool is installed.

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

Procedure

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

```
# podman run -d --ip=10.88.0.44 registry.access.redhat.com/rhel7/rsyslog
efde5f0a8c723f70dd5cb5dc3d5039df3b962fae65575b08662e0d5b5f9fbe85
```

2. Check that the IP address is set properly:

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

Additional resources

- **man podman-run**

4.10. EXECUTING COMMANDS INSIDE A RUNNING CONTAINER

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

Prerequisites

- The Podman tool is installed.

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

- The container is running.

Procedure

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

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

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

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

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

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

4. Inspect the container:

   - To list every process on the system:

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

   - To display file system disk space usage:

```
# df -h
Filesystem      Size  Used Avail Use% Mounted on
fuse-overlayfs   27G  7.1G   20G  27% /
tmpfs            64M     0   64M   0% /dev
tmpfs           269M  936K  268M   1% /etc/hosts
shm              63M     0   63M   0% /dev/shm
...
```

- To display system information:

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

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

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

Additional resources

- **man podman-exec**

4.11. SHARING FILES BETWEEN TWO CONTAINERS

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

Main advantages are:

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

Prerequisites

- The Podman tool is installed.

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

Procedure

1. Create a volume:

```
$ podman volume create hostvolume
```

2. Display information about the volume:

```
$ podman volume inspect hostvolume
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Additional resources

- **man podman-volume**

4.12. EXPORTING AND IMPORTING CONTAINERS

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

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

Prerequisites

- The Podman tool is installed.

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

Procedure

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

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

2. Optional: List all containers:

   ```
   $ podman ps -a
   CONTAINER ID  IMAGE                                    COMMAND       CREATED     STATUS
   PORTS   NAMES
   a6a6d4896142  registry.access.redhat.com/ubi8:latest   /bin/bash       7 seconds ago  Up 7
   seconds ago          myubi
   ```

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

   ```
   $ podman attach myubi
   ```

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

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

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

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

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

7. Optional: List the current directory content:

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

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

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

```
├── srv
├── sys
├── testfile
├── tmp
├── usr
└── var

20 directories, 1 file
```

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

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

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

10. List all images:

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

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

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

Additional resources

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

4.13. STOPPING CONTAINERS

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

Prerequisites

- The Podman tool is installed.

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

- At least one container is running.

Procedure

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

    ```
    $ podman stop myubi
    ```

  - Using the container ID:

    ```
    $ podman stop 1984555a2c27
    ```

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

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

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

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

Additional resources

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

4.14. REMOVING CONTAINERS

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

Prerequisites

- The Podman tool is installed.

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

- At least one container has been stopped.

Procedure

1. List all containers, running or stopped:

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

2. Remove the containers:

   - To remove the **peaceful_hopper** container:

     ```
     $ podman rm peaceful_hopper
     ```

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

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

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

     NOTE

     - To remove multiple containers:

       ```
       $ podman rm clever_yonath furious_shockley
       ```

     - To remove all containers from your local system:

       ```
       $ podman rm -a
       ```

Additional resources

- **man podman-rm**

CHAPTER 5. WORKING WITH PODS

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

5.1. CREATING PODS

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

Prerequisites

- The Podman tool is installed.

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

Procedure

1. Create an empty pod:

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

   The pod is in the initial state Created.

2. Optional: List all pods:

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

   Notice that the pod has one container in it.

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

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

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

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

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

5. Optional: List all pods:

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

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

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

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

Additional resources

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

5.2. DISPLAYING POD INFORMATION

This procedure provides information on how to display pod information.

Prerequisites

- The Podman tool is installed.

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

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

Procedure

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

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

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

```
$ podman pod stats -a --no-stream
```

```
ID            NAME              CPU %   MEM USAGE / LIMIT   MEM %   NET IO    BLOCK IO
PIDS
a9f807ffaacd   frosty_hodgkin    --      3.092MB / 16.7GB    0.02%   -- / --   -- / --    2
3b33001239ee   sleepy_stallman   --      -- / --             --      -- / --   -- / --    --
```

- To display information describing the pod, enter:

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

You can see information about containers in the pod.

Additional resources

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

5.3. STOPPING PODS

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

Prerequisites

- The Podman tool is installed.

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

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

Procedure

1. Stop the pod **mypod**:

   ```
   $ podman pod stop mypod
   ```

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

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

   3afdcd93de3e  registry.access.redhat.com/ubi8/pause                        About a minute ago
   Exited (0) 7 seconds ago         8a4e6527ac9d-infra  223df6b390b4  mypod
   ```

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

Additional resources

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

5.4. REMOVING PODS

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

Prerequisites

- The Podman tool is installed.

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

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

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

Procedure

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

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

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

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

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

Additional resources

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

CHAPTER 6. 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.

6.1. 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.

6.2. 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.

6.3. 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.

6.4. 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.

6.5. 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:

```
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:

```
# 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 -y
. . .
No packages marked for update
STEP 5: RUN yum install --disablerepo=* --enablerepo=ubi-8-appstream --enablerepo=ubi-
8-baseos httpd -y
Loaded 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
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.
...

Transaction Summary
...
Complete!
STEP 6: RUN echo "The Web Server is Running" > /var/www/html/index.html
STEP 7: EXPOSE 80
STEP 8: CMD ["-D", "FOREGROUND"]
STEP 9: ENTRYPOINT ["/usr/sbin/httpd"]
STEP 10: COMMIT
...
Writing manifest to image destination
Storing signatures
--> 36a604cc0dd3657b46f8762d7ef69873f65e16343b54c63096e636c80f0d68c7
```

3. **Test**: Test the UBI layered webserver image:

```
# podman run -d -p 80:80 johndoe/webserver
bbe98c71d18720d966e4567949888dc4fb86eec7d304e785d5177168a5965f64
# curl http://localhost/index.html
The Web Server is Running
```

6.6. 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 Red Hat Container Image Updates .

6.7. 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 f7d970ccd456 done
Copying blob ade06f94b556 done
Copying blob cc56c782b513 done
Copying blob dcf9396fdada 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":
"sha256:7ab721ef3305271bbb629a6db065c59bbeb87bc53e7cbf88e2953a1217ba7322",
   "RepoTags": [],
   "Created": "2020-02-11T12:14:18.612461174Z",
   "DockerVersion": "",
   "Labels": null,
   "Architecture": "amd64",
   "Os": "linux",
   "Layers": [
      "sha256:1ae73d938ab9f11718d0f6a4148eb07d38ac1c0a70b1d03e751de8bf3c2c87fa",
      "sha256:9fe966885cb8712c47efe5ecc2eaa0797a0d5ffb8b119c4bd4b400cc9e255421",
      "sha256:61b2527a4b836a4efbb82dfd449c0556c0f769570a6c02e112f88f8bbcd90166",
      ...
      "sha256:cc56c782b513e2bdd2cc2af77b69e13df4ab624ddb856c4d086206b46b9b9e5f",
      "sha256:dcf9396fdada4e6c1ce667b306b7f08a83c9e6b39d0955c481b8ea5b2a465b32",

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

3. 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/sha256/10914f1fff060ce31388f5ab963871870535aaaa551629f5ad182384d60fdf82
rpm_dir/
rpm_dir/gzip-1.9-4.el8.src.rpm
```

5. Begin examining and using the content.

6.8. 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 7. RUNNING SKOPEO, BUILDAH, AND PODMAN IN A CONTAINER

This chapter describes how you can run Skopeo, Buildah, and Podman in a container.

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

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

With Podman, you can manage containers and images, volumes mounted into those containers, and pods made from groups of containers. Podman is based on a **libpod** library for container lifecycle management. The **libpod** library provides APIs for managing containers, pods, container images, and volumes.

Reasons to run Buildah, Skopeo, and Podman in a container:

- **CI/CD system**:
  - **Podman and Skopeo**: You can run a CI/CD system inside of Kubernetes or use OpenShift to build your container images, and possibly distribute those images across different container registries. To integrate Skopeo into a Kubernetes workflow, you need to run it in a container.
  - **Buildah**: You want to build OCI/container images within a Kubernetes or OpenShift CI/CD systems that are constantly building images. Previously, people used a Docker socket to connect to the container engine and perform a **docker build** command. This was the equivalent of giving root access to the system without requiring a password which is not secure. For this reason, Red Hat recommends using Buildah in a container.
- **Different versions**:
  - **All**: You are running an older OS on the host but you want to run the latest version of Skopeo, Buildah, or Podman. The solution is to run the container tools in a container. For example, this is useful for running the latest version of the container tools provided in RHEL 8 on a RHEL 7 container host which does not have access to the newest versions natively.
- **HPC environment**:
  - **All**: A common restriction in HPC environments is that non-root users are not allowed to install packages on the host. When you run Skopeo, Buildah, or Podman in a container, you can perform these specific tasks as a non-root user.

7.1. RUNNING SKOPEO IN A CONTAINER

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

Prerequisites

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

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

Procedure

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

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

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

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

3. Inspect a remote container image **registry.access.redhat.com/ubi8/ubi** using Skopeo:

```
$ podman run --rm registry.redhat.io/rhel8/skopeo skopeo inspect
docker://registry.access.redhat.com/ubi8/ubi
{
    "Name": "registry.access.redhat.com/ubi8/ubi",
    ...
    "Labels": {
        "architecture": "x86_64",
        ...
        "name": "ubi8",
        ...
        "summary": "Provides the latest release of Red Hat Universal Base Image 8.",
        "url":
"https://access.redhat.com/containers/#/registry.access.redhat.com/ubi8/images/8.2-347",
        ...
    },
    "Architecture": "amd64",
    "Os": "linux",
    "Layers": [
    ...
    ],
    "Env": [
        "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
        "container=oci"
    ]
}
```

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

Additional resources

- How to run skopeo in a container article

7.2. RUNNING SKOPEO IN A CONTAINER USING CREDENTIALS

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

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

Prerequisites

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

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

Procedure

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

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

Additional resources

- How to run skopeo in a container article

7.3. RUNNING SKOPEO IN A CONTAINER USING AUTHFILES

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

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

Prerequisites

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

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

Procedure

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

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

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

Other Skopeo commands work similarly, for example:

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

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

Additional resources

- How to run skopeo in a container article

7.4. COPYING CONTAINER IMAGES TO OR FROM THE HOST

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

NOTE

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

Prerequisites

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

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

Procedure

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

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

   - The **--privileged** option disables all security mechanisms. Red Hat recommends only using this option in trusted environments.
   - To avoid disabling security mechanisms, export the images to a tarball or any other path-based image transport and mount them in the Skopeo container:
     - **$ podman save --format oci-archive -o oci.tar $IMAGE**
     - **$ podman run --rm -v oci.tar:/oci.tar registry.redhat.io/rhel8/skopeo copy oci-archive:/oci.tar $DESTINATION**

2. Optional: List images in local storage:

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

Additional resources

- How to run skopeo in a container article

7.5. RUNNING BUILDAH IN A CONTAINER

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

Prerequisites

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

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

Procedure

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

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

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

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

   - The **--rm** option removes the **registry.redhat.io/rhel8/buildah** image after the container exits.
   - The **--device** option adds a host device to the container.

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

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

4. Run the **ls** / command inside the **ubi8-working-container** container:

   ```
   # buildah run  --isolation=chroot ubi8-working-container ls /
   bin  boot  dev  etc  home  lib  lib64  lost+found  media  mnt  opt  proc  root  run  sbin  srv
   ```

5. Optional: List all images in a local storage:

   ```
   # buildah images
   REPOSITORY                        TAG      IMAGE ID       CREATED       SIZE
   registry.access.redhat.com/ubi8   latest   ecbc6f53bba0   5 weeks ago   211 MB
   ```

6. Optional: List the working containers and their base images:

   ```
   # buildah containers
   CONTAINER ID  BUILDER  IMAGE ID     IMAGE NAME                       CONTAINER NAME
   0aaba7192762     *     ecbc6f53bba0 registry.access.redhat.com/ub... ubi8-working-container
   ```

7. Optional: Push the **registry.access.redhat.com/ubi8** image to the a local registry located on **registry.example.com**:

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

Additional resources

- Best practices for running Buildah in a container article

7.6. RUNNING PODMAN IN A CONTAINER

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

Prerequisites

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

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

- You are logged in to the registry.redhat.io registry:

  ```
  # podman login registry.redhat.io
  ```

Procedure

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

   ```
   # podman run --privileged --name podman_container -it registry.redhat.io/rhel8/podman
   /bin/bash
   ```

   - The **--it** option specifies that you want to run an interactive bash shell within a container.
   - The **--privileged** option adds additional capabilities necessary to run Podman in Podman.

2. Create a **Containerfile** inside the **podman_container** container:

   ```
   # vi Containerfile
   FROM registry.access.redhat.com/ubi8/ubi
   RUN yum install -y https://dl.fedoraproject.org/pub/epel/epel-release-latest-8.noarch.rpm
   RUN yum -y install moon-buggy && yum clean all
   CMD ["/usr/bin/moon-buggy"]
   ```

   The commands in the **Containerfile** cause the following build command to:

   - Build a container from the **registry.access.redhat.com/ubi8/ubi** image.
   - Install the **epel-release-latest-8.noarch.rpm** package.
   - Install the **moon-buggy** package.
   - Set the container command.

3. Build a new container image named **moon-buggy** using the **Containerfile**:

   ```
   # podman build -t moon-buggy .
   ```

4. Optional: List all images:

   ```
   # podman images
   REPOSITORY                  TAG      IMAGE ID       CREATED          SIZE
   localhost/moon-buggy  latest  c97c58abb564  13 seconds ago  1.67 GB
   registry.access.redhat.com/ubi8/ubi latest 4199acc83c6a  132seconds ago 213 MB
   ```

5. Run a new container based on a **moon-buggy** container:

   ```
   # podman run -it --name moon moon-buggy
   ```

6. Optional: Tag the **moon-buggy** image:

   ```
   # podman tag moon-buggy registry.example.com/moon-buggy
   ```

7. Optional: Push the **moon-buggy** image to the registry:

   ```
   # podman push registry.example.com/moon-buggy
   ```

Additional resources

- Technology preview: Running a container inside a container article

CHAPTER 8. 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**.

8.1. 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
   ```

   To install the full set of container tools, type:

   ```
   # 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/sos.u82evisb 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: c4e1fd3ee45f78a17afb4e45a05842ed
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     1  0 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.

8.1.1. 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.

8.2. 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.

8.2.1. 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:

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

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

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

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

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

```
# podman container runlabel install rhel8/rsyslog
command: podman run --rm --privileged -v /:/host -e HOST=/host -e
IMAGE=registry.redhat.io/rhel8/rsyslog:latest -e NAME=rsyslog
registry.redhat.io/rhel8/rsyslog:latest /bin/install.sh
Creating directory at /host//etc/pki/rsyslog
Creating directory at /host//etc/rsyslog.d
Installing file at /host//etc/rsyslog.conf
Installing file at /host//etc/sysconfig/rsyslog
Installing file at /host//etc/logrotate.d/syslog
```

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

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

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

```
-v /var/lib/rsyslog:/var/lib/rsyslog -v /run:/run -v /etc/machine-id:/etc/machine-id -v
/etc/localtime:/etc/localtime -e IMAGE=registry.redhat.io/rhel8/rsyslog:latest -e
NAME=rsyslog --restart=always registry.redhat.io/rhel8/rsyslog:latest /bin/rsyslog.sh
```

This shows that the command opens privileges to the host and mount specific files and directories from the host inside the container, when it launches the **rsyslog** container to run the **rsyslogd** daemon.

5. Execute the **run** runlabel for **rsyslog**:

```
# podman container runlabel run rhel8/rsyslog
command: podman run -d --privileged --name rsyslog --net=host --pid=host -v
/etc/pki/rsyslog:/etc/pki/rsyslog -v /etc/rsyslog.conf:/etc/rsyslog.conf -v
/etc/sysconfig/rsyslog:/etc/sysconfig/rsyslog -v /etc/rsyslog.d:/etc/rsyslog.d -v /var/log:/var/log
-v /var/lib/rsyslog:/var/lib/rsyslog -v /run:/run -v /etc/machine-id:/etc/machine-id -v
/etc/localtime:/etc/localtime -e IMAGE=registry.redhat.io/rhel8/rsyslog:latest -e
NAME=rsyslog --restart=always registry.redhat.io/rhel8/rsyslog:latest /bin/rsyslog.sh
28a0d719ff179adcea81eb63cc90fcd09f1755d5edb121399068a4ea59bd0f53
```

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

6. Display the **uninstall** runlabel for **rsyslog**:

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

7. Run the **uninstall** runlabel for **rsyslog**:

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

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.

8.2.2. 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**:

```
# 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 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 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
bash-4.4#
```

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 9. PORTING CONTAINERS TO OPENSHIFT USING PODMAN

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

The YAML files are:

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

Reasons to use YAML files:

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

9.1. 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
5df5c48fea87  registry.access.redhat.com/ubi8/ubi:latest  /bin/bash  Less than a second ago
Up Less than a second ago         myubi               223df6b390b4
3afdcd93de3e  k8s.gcr.io/pause:3.1                                Less than a second ago  Up Less
than a second ago         223df6b390b4-infra  223df6b390b4
```

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

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

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

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

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

Additional resources

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

9.2. GENERATING A KUBERNETES YAML FILE IN OPENSHIFT ENVIRONMENT

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

Procedure

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

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

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

NOTE

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

9.3. STARTING CONTAINERS AND PODS WITH PODMAN

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

Procedure

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

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

2. List all pods:

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

3. List all pods and containers associated with them:

   ```
   $ podman ps -a --pod
   CONTAINER ID  IMAGE                                    COMMAND    CREATED            STATUS
   PORTS  NAMES              POD
   848179395ebd  registry.access.redhat.com/ubi8/ubi:latest  /bin/bash  About a minute ago  Up
   About a minute ago         myubi              b8c5b99ba846
   aa4220eaf4bb  k8s.gcr.io/pause:3.1                               About a minute ago  Up About a
   minute ago         b8c5b99ba846-infra  b8c5b99ba846
   ```

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

Additional resources

- **man podman-play-kube**
- Podman can now ease the transition to Kubernetes and CRI-O article

9.4. STARTING CONTAINERS AND PODS IN OPENSHIFT ENVIRONMENT

You can use the **oc create** command to create pods and containers in the OpenShift environment.

Procedure

- Create a pod from the YAML file in the OpenShift environment:

```
$ oc create -f mypod.yaml
```

NOTE

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

CHAPTER 10. PORTING CONTAINERS TO SYSTEMD USING PODMAN

Podman (Pod Manager) is a fully featured container engine that is a simple daemonless tool. Podman provides a Docker-CLI comparable command line that eases the transition from other container engines and allows the management of pods, containers and images.

Podman was not originally designed to bring up an entire Linux system or manage services for such things as start-up order, dependency checking, and failed service recovery. That is the job of a full-blown initialization system like systemd. Red Hat has become a leader in integrating containers with systemd, so that OCI and Docker-formatted containers built by Podman can be managed in the same way that other services and features are managed in a Linux system. You can use the systemd initialization service to work with pods and containers. You can use the **podman generate systemd** command to generate a systemd unit file for containers and pods.

With systemd unit files, you can:

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

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

10.1. ENABLING SYSTEMD SERVICES

When enabling the service, you have different options.

Procedure

- Enable the service:
  - To enable a service at system start, no matter if user is logged in or not, enter:

    ```
    # systemctl enable <service>
    ```

    You have to copy the systemd unit files to the **/etc/systemd/system** directory.
  - To start a service at user login and stop it at user logout, enter:

    ```
    $ systemctl --user enable <service>
    ```

    You have to copy the systemd unit files to the **$HOME/.config/systemd/user** directory.
  - To enable users to start a service at system start and persist over logouts, enter:

    ```
    # loginctl enable-linger <username>
    ```

Additional resources

- **man systemctl**

- **man loginctl**
- Managing services with systemd chapter

10.2. 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
0280afe98bb75a5c5e713b28de4b7c5cb49f156f1cce4a208f13fee2f75cb453
```

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

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

Verification steps

- 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/0280afe98bb75a5c5e713b28de4b7c5cb49f156f1cce4a208f13fee2f75cb453/userdat
a/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

10.3. 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
1e12cf95e305435c0001fa7d4a14cf1d52f737c1118328937028c0bd2fdec5ca
```

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

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

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

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

```
# cp -Z container-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
- Managing services with systemd chapter

10.4. 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/YJSPGXM2OCDZPLMLXJOW3N
RF6Q:/home/user/.local/share/contain>
           ├─80617 /usr/bin/conmon --api-version 1 -c
cbc75d6031508dfd3d78a74a03e4ace1732b51223e72a2ce4aa3bfe10a78e4fa -u
cbc75d6031508dfd3d78a74a03e4ace1732b51223e72>
           └─cbc75d6031508dfd3d78a74a03e4ace1732b51223e72a2ce4aa3bfe10a78e4fa
             └─80626 /usr/bin/coreutils --coreutils-prog-shebang=sleep /usr/bin/sleep 1d
```

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

Verification steps

- List containers that are running or have exited:

```
# podman ps
CONTAINER ID  IMAGE                              COMMAND  CREATED        STATUS
PORTS  NAMES
f20988d59920  registry.access.redhat.com/ubi8-minimal:latest  top      12 seconds ago  Up 11
seconds ago        funny_zhukovsky
```

NOTE

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

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

Additional resources

- **man systemctl**
- Running containers with Podman and shareable systemd services article
- Managing services with systemd chapter

10.5. AUTO-STARTING PODS USING SYSTEMD

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

Procedure

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

```
$ podman pod create --name systemd-pod
11d4646ba41b1fffa51c108cbdf97cfab3213f7bd9b3e1ca52fe81b90fed5577
```

2. List all pods:

```
$ podman pod ps
POD ID        NAME         STATUS   CREATED          # OF CONTAINERS  INFRA ID
11d4646ba41b  systemd-pod  Created  40 seconds ago  1                8a428b257111
11d4646ba41b1fffa51c108cbdf97cfab3213f7bd9b3e1ca52fe81b90fed5577
```

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

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

4. List all pods and containers associated with them:

```
$ podman ps -a --pod
CONTAINER ID  IMAGE                                   COMMAND  CREATED        STATUS
PORTS   NAMES             POD ID        PODNAME
24666f47d9b2  registry.access.redhat.com/ubi8:latest  top      3 minutes ago  Created
```

```
container0          3130f724e229  systemd-pod
56eb1bf0cdfe  k8s.gcr.io/pause:3.2                           4 minutes ago  Created
3130f724e229-infra  3130f724e229  systemd-pod
62118d170e43  registry.access.redhat.com/ubi8:latest  top       3 seconds ago  Created
container1          3130f724e229  systemd-pod
```

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

```
$ podman generate systemd --files --name systemd-pod
/home/user1/pod-systemd-pod.service
/home/user1/container-container0.service
/home/user1/container-container1.service
```

Note that three systemd unit files are generated, one for the **systemd-pod** pod and two for the containers **container0** and **container1**.

6. Display **pod-systemd-pod.service** unit file:

```
$ cat pod-systemd-pod.service
# pod-systemd-pod.service
# autogenerated by Podman 2.0.3
# Tue Jul 28 14:00:46 EDT 2020


[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 c852fbaba568-infra
ExecStop=/usr/bin/podman stop -t 10 c852fbaba568-infra
ExecStopPost=/usr/bin/podman stop -t 10 c852fbaba568-infra
PIDFile=/run/user/1000/containers/overlay-
containers/a7ff86382608add27a03ac2166d5d0164199f01eadf80b68b06a406c195105fc/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/12e85378f2854b8283f791974494a02aa6c92630d76d1050237839b61508a008/user
data/conmon.pid
KillMode=none
Type=forking

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

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

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

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

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

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

10. Enable the service and start at user login:

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

Note that the service stops at user logout.

Verification steps

- Check if the service is enabled:

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

Additional resources

- **man podman-create**
- **man podman-generate-systemd**
- **man systemctl**
- Running containers with Podman and shareable systemd services article
- Managing services with systemd chapter

10.6. AUTO-UPDATING CONTAINERS USING PODMAN

The **podman auto-update** command allows you to automatically update containers according to their auto-update policy. The **podman auto-update** command updates services when the container image is updated on the registry. To use auto-updates, containers must be created with the **--label "io.containers.autoupdate=image"** label and run in a systemd unit generated by **podman generate systemd --new** command.

Podman searches for running containers with the **"io.containers.autoupdate"** label set to **"image"** and communicates to the container registry. If the image has changed, Podman restarts the corresponding systemd unit to stop the old container and create a new one with the new image. As a result, the container, its environment, and all dependencies, are restarted.

Prerequisites

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

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

Procedure

1. Start a **myubi** container based on the **registry.access.redhat.com/ubi8/ubi-init** image:

```
# podman run --label "io.containers.autoupdate=image" \
--name myubi  -dt registry.access.redhat.com/ubi8/ubi-init top
bc219740a210455fa27deacc96d50a9e20516492f1417507c13ce1533dbdcd9d
```

2. Optional: List containers that are running or have exited:

```
# podman ps -a
CONTAINER ID  IMAGE                                         COMMAND  CREATED         STATUS
PORTS   NAMES
76465a5e2933  registry.access.redhat.com/ubi8/ubi-init:latest  top      24 seconds ago  Up 23
seconds ago          myubi
```

3. Generate a systemd unit file for the **myubi** container:

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

4. Copy unit files to **/usr/lib/systemd/system** for installing it as a root user:

```
# cp -Z ~/container-myubi.service /usr/lib/systemd/system
```

5. Reload systemd manager configuration:

```
# systemctl daemon-reload
```

6. Start and check the status of a container:

```
# systemctl start container-myubi.service
# systemctl status container-myubi.service
```

7. Auto-update the container:

```
# podman auto-update
```

Additional resources

- Improved Systemd Integration with Podman 2.0 article
- Running containers with Podman and shareable systemd services article
- Managing Services with systemd chapter

10.7. AUTO-UPDATING CONTAINERS USING SYSTEMD

As mentioned in section Auto-updating containers using Podman, you can update the container using the **podman auto-update** command. It integrates into custom scripts and can be invoked when needed. Another way to auto update the containers is to use the pre-installed **podman-auto-update.timer** and **podman-auto-update.service** systemd service. The **podman-auto-update.timer** can be configured to trigger auto updates at a specific date or time. The **podman-auto-update.service** can further be started by the **systemctl** command or be used as a dependency by other systemd services. As a result, auto updates based on time and events can be triggered in various ways to meet individual needs and use cases.

Prerequisites

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

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

Procedure

1. Display the **podman-auto-update.service** unit file:

```
# cat /usr/lib/systemd/system/podman-auto-update.service

[Unit]
```

```
Description=Podman auto-update service
Documentation=man:podman-auto-update(1)
Wants=network.target
After=network-online.target

[Service]
Type=oneshot
ExecStart=/usr/bin/podman auto-update

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

2. Display the **podman-auto-update.timer** unit file:

```
# cat /usr/lib/systemd/system/podman-auto-update.timer

[Unit]
Description=Podman auto-update timer

[Timer]
OnCalendar=daily
Persistent=true

[Install]
WantedBy=timers.target
```

In this example, the **podman auto-update** command is launched daily at midnight.

3. Enable the **podman-auto-update.timer** service at system start:

```
# systemctl enable podman-auto-update.timer
```

4. Start the systemd service:

```
# systemctl start podman-auto-update.timer
```

5. Optional: List all timers:

```
# systemctl list-timers --all
NEXT                         LEFT     LAST                          PASSED      UNIT
ACTIVATES
Wed 2020-12-09 00:00:00 CET  9h left   n/a                          n/a          podman-auto-
update.timer     podman-auto-update.service
```

You can see that **podman-auto-update.timer** activates the **podman-auto-update.service**.

Additional resources

- Improved Systemd Integration with Podman 2.0 article
- Running containers with Podman and shareable systemd services article
- Managing Services with systemd chapter

CHAPTER 11. 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.

11.1. 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

11.1.1. 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
buildah(1)       General Commands Manual       buildah(1)

NAME
 Buildah - A command line tool that facilitates building OCI container images.
...
```

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.

11.2. 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
```

```
IMAGE ID      IMAGE NAME                         CREATED AT         SIZE
3da40a1670b5  registry.redhat.io/ubi8/ubi:latest  May 8, 2019 21:55  214 MB
# buildah containers
CONTAINER ID  BUILDER  IMAGE ID     IMAGE NAME         CONTAINER NAME
c6c9279ecc0f     *     3da40a1670b5 ...ubi8/ubi:latest ubi-working-container
```

Notice that the result of the **buildah from** 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.

11.3. 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.

11.3.1. 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!
```

11.3.2. 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.

11.4. 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.

11.4.1. 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
6d1ffccb557d   *     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/176c273fe28c23e5319805a2c48559305a57a706cc7ae7b
ec7da4cd79edd3c02/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
IMAGE ID     IMAGE NAME     CREATED AT          SIZE
a7e06d3cd0e2 docker.io/library/myecho2:latest
                        Oct 12, 2017 15:15  3.144 KB
# buildah from docker.io/library/myecho2:latest
myecho2-working-container
# podman run docker.io/library/myecho2
This container works!
We even modified it.
```

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
```

11.4.2. 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 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:

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

Create a new working container:

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

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

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

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

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

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

```
# 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):

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

11.5. 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:

```
# scratchmnt=$(buildah mount working-container)
# echo $scratchmnt
/var/lib/containers/storage/devicemapper/mnt/cc92011e9a2b077d03a97c0809f1f3e7fef0f29bdc6ab5e86
b85430ec77b2bf6/rootfs
```

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

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

Install the httpd service to the scratch directory:

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

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

```
# 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:

```
# 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:

```
# 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.
```

11.6. 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:

```
# buildah containers
CONTAINER ID  BUILDER  IMAGE ID     IMAGE NAME                       CONTAINER NAME
05387e29ab93     *     c37e14066ac7 docker.io/library/myecho:latest  myecho-working-container
```

```
# buildah mount
05387e29ab93 /var/lib/containers/storage/devicemapper/mnt/9274181773a.../rootfs
# buildah umount 05387e29ab93
# buildah rm 05387e29ab93
05387e29ab93151cf52e9c85c573f3e8ab64af1592b1ff9315db8a10a77d7c22
```

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
```

11.7. 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.

11.7.1. 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:

```
# 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:

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

11.7.2. 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:

```
# 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:

```
# 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 12. 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.

12.1. 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
- Retries
- Interval
- Start-period
- Timeout

The description of healthcheck components follows.

Command

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

The other four components are related to the scheduling of the healthcheck and they are optional.

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 time outs.

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
D25ee6faaf6e5e12c09e734b1ac675385fe4d4e8b52504dd01a60e1b726e3edb
```

   - 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:

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

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

```
$ echo $?
0
```

   The "0" value means success.

Additional resources

- **man podman-run**
- Monitoring container vitality and availability with Podman article

12.2. 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:

```
$ 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:

```
$ podman system df -v
Images space usage:

REPOSITORY                          TAG       IMAGE ID       CREATED         SIZE     SHARED
SIZE   UNQUE SIZE   CONTAINERS
docker.io/library/alpine            latest   e7d92cdc71fe   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:
```

```
CONTAINER ID   IMAGE   COMMAND    LOCAL VOLUMES   SIZE   CREATED
STATUS   NAMES
ff0167c6c271    8121    /bin/bash   0             0B     10 seconds ago   exited   playTest

Local Volumes space usage:

VOLUME NAME   LINKS   SIZE
```

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

```
$ 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:
7fd3f71a218f8d3a7202e464252aeb1e942d17eb'
  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:
    executable: /usr/bin/slirp4netns
    package: slirp4netns-1.1.1-1.module+el8.3.0+7084+c16098dd.x86_64
    version: |-
      slirp4netns version 1.1.1
      commit: bbf27c5acd4356edb97fa639b4e15e0cd56a39d5
      libslirp: 4.3.0
      SLIRP_CONFIG_VERSION_MAX: 3
  swapFree: 1833693184
  swapTotal: 2147479552
  uptime: 145h 19m 14.55s (Approximately 6.04 days)
registries:
  search:
  - registry.access.redhat.com
  - registry.redhat.io
  - docker.io
store:
  configFile: /home/user/.config/containers/storage.conf
  containerStore:
    number: 1
    paused: 0
    running: 0
    stopped: 1
  graphDriverName: overlay
  graphOptions:
    overlay.mount_program:
      Executable: /usr/bin/fuse-overlayfs
      Package: fuse-overlayfs-1.1.1-1.module+el8.3.0+7121+472bc0cf.x86_64
      Version: |-
        fuse-overlayfs: version 1.1.0
        FUSE library version 3.2.1
        using FUSE kernel interface version 7.26
  graphRoot: /home/user/.local/share/containers/storage
  graphStatus:
    Backing Filesystem: xfs
    Native Overlay Diff: "false"
    Supports d_type: "true"
    Using metacopy: "false"
  imageStore:
    number: 15
  runRoot: /run/user/1000/containers
  volumePath: /home/user/.local/share/containers/storage/volumes
version:
  APIVersion: 1
  Built: 0
  BuiltTime: Thu Jan  1 01:00:00 1970
  GitCommit: ""
  GoVersion: go1.14.2
  OsArch: linux/amd64
  Version: 2.0.0
```

- To remove all unused containers, images and volume data, enter:

```
$ podman system prune
WARNING! This will remove:
        - all stopped containers
        - all stopped pods
        - all dangling images
        - all build cache
Are you sure you want to continue? [y/N] y
```

- The **podman system prune** command removes all unused containers (both dangling and unreferenced), pods and optionally, volumes from local storage.
- Use the **--all** option to delete all unused images. Unused images are dangling images and any image that does not have any containers based on it.
- Use the **--volume** option to prune volumes. By default, volumes are not removed to prevent important data from being deleted if there is currently no container using the volume.

Additional resources

- **man podman-system-df**
- **man podman-system-info**
- **man podman-system-prune**

12.3. 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

- restart
- restore
- start
- stop
- sync
- unmount
- unpause

The *pod* event type reports the following statuses:

- create
- kill
- pause
- remove
- start
- stop
- unpause

The *image* event type reports the following statuses:

- prune
- push
- pull
- save
- remove
- tag
- untag

The *system* type reports the following statuses:

- refresh
- renumber

The *volume* type reports the following statuses:

- create
- prune

- remove

Additional resources

- **man podman-events**

12.4. 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:

      ```
      $ podman events
      2020-05-14 10:33:42.312377447 -0600 CST container create 34503c192940
      (image=registry.access.redhat.com/ubi8/ubi:latest, name=keen_colden)
      2020-05-14 10:33:46.958768077 -0600 CST container init 34503c192940
      (image=registry.access.redhat.com/ubi8/ubi:latest, name=keen_colden)
      2020-05-14 10:33:46.973661968 -0600 CST container start 34503c192940
      (image=registry.access.redhat.com/ubi8/ubi:latest, name=keen_colden)
      2020-05-14 10:33:50.833761479 -0600 CST container stop 34503c192940
      (image=registry.access.redhat.com/ubi8/ubi:latest, name=keen_colden)
      2020-05-14 10:33:51.047104966 -0600 CST container cleanup 34503c192940
      (image=registry.access.redhat.com/ubi8/ubi:latest, name=keen_colden)
      ```

      To exit logging, press CTRL+c.

    - To show only Podman create events, enter:

      ```
      $ podman events --filter event=create
      2020-05-14 10:36:01.375685062 -0600 CST container create 20dc581f6fbf
      (image=registry.access.redhat.com/ubi8/ubi:latest)
      2019-03-02 10:36:08.561188337 -0600 CST container create 58e7e002344c
      (image=registry.access.redhat.com/ubi8/ubi-minimal:latest)
      2019-03-02 10:36:29.978806894 -0600 CST container create d81e30f1310f
      (image=registry.access.redhat.com/ubi8/ubi-init:latest)
      ```

Additional resources

- **man podman-events**

CHAPTER 13. USING THE CONTAINER-TOOLS CLI

13.1. 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.

13.1.1. 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 13.1. Commands supported by podman

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

save	Save image to an archive	**search**	search registry for image
start	Start one or more containers	**stats**	Display percentage of CPU, memory, network I/O, block I/O and PIDs for one or more containers
stop	Stop one or more containers	**tag**	Add an additional name to a local image
top	Display the running processes of a container	**umount, unmount**	Unmount a working container's root filesystem
unpause	Unpause the processes in one or more containers	**version**	Display podman version information
wait	Block on one or more containers		

13.1.2. Using the crun container runtime

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

Additional features of **crun** include:

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

13.1.3. Temporarily using crun as the container runtime

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

Prerequisites

- The Podman tool is installed.

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

Procedure

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

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

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

```
$ podman run --name=myubi -dt --runtime=crun ubi8 bash
e4654eb4df12ac031f1d0f2657dc4ae6ff8eb0085bf114623b66cc664072e69b
```

3. Optional: List all images:

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

Verification steps

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

```
$ podman inspect myubi --format "{{.OCIRuntime}}"
crun
```

Additional resources

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

13.1.4. Permanently using crun as the container runtime

You can set the container runtime and its options in the **containers.conf** configuration file.

Procedure

1. Change the runtime from **runc** to **crun** in the **containers.conf** file:

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

2. Run the container named **myubi**:

```
# podman run --name=myubi -dt ubi8 bash
```

Verification steps

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

```
# podman inspect myubi --format "{{.OCIRuntime}}"
crun
```

Additional resources

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

13.1.5. 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.

13.1.6. 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:

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

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

   ```
   $ cat Containerfile
   FROM registry.access.redhat.com/ubi8/ubi

   RUN yum -y install openmpi-devel wget && \
       yum clean all
   ```

```
RUN wget https://raw.githubusercontent.com/open-mpi/ompi/master/test/simple/ring.c && \
    /usr/lib64/openmpi/bin/mpicc ring.c -o /home/ring && \
    rm -f ring.c
```

5. Build the container:

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

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

```
$ mpirun \
   --mca orte_tmpdir_base /tmp/podman-mpirun \
   podman run --env-host \
     -v /tmp/podman-mpirun:/tmp/podman-mpirun \
     --userns=keep-id \
     --net=host --pid=host --ipc=host \
     mpi-ring /home/ring
Rank 2 has cleared MPI_Init
Rank 2 has completed ring
Rank 2 has completed MPI_Barrier
Rank 3 has cleared MPI_Init
Rank 3 has completed ring
Rank 3 has completed MPI_Barrier
Rank 1 has cleared MPI_Init
Rank 1 has completed ring
Rank 1 has completed MPI_Barrier
Rank 0 has cleared MPI_Init
Rank 0 has completed ring
Rank 0 has completed MPI_Barrier
```

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

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.

13.1.7. 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.

13.1.7.1. Creating and restoring a container checkpoint locally

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

Procedure

1. Create a Python based server:

```
# cat counter.py
#!/usr/bin/python3

import http.server

counter = 0

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


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

2. Create a container with the following definition:

```
# cat Containerfile
FROM registry.access.redhat.com/ubi8/ubi

COPY counter.py /home/counter.py

RUN useradd -ms /bin/bash counter

RUN yum -y install python3 && chmod 755 /home/counter.py

USER counter
ENTRYPOINT /home/counter.py
```

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

3. Build the container:

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

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

4. Start the container as root:

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

5. To list all running containers, enter:

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

6. Display IP address of the container:

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

7. Send requests to the container:

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

8. Create a checkpoint for the container:

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

9. Reboot the system.

10. Restore the container:

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

11. Send requests to the container:

    ```
    # curl 10.88.0.247:8080
    2
    # curl 10.88.0.247:8080
    3
    # curl 10.88.0.247:8080
    4
    ```

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

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

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

13.1.7.2. 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:

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

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

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

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

3. Display ID and name of each container:

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

4. Display IP address of each container:

   ```
   #  podman inspect counter1 --format "{{.NetworkSettings.IPAddress}}"
   10.88.0.248

   #  podman inspect counter2 --format "{{.NetworkSettings.IPAddress}}"
   10.88.0.249

   #  podman inspect counter3 --format "{{.NetworkSettings.IPAddress}}"
   10.88.0.250
   ```

5. Send requests to each container:

   ```
   #  curl 10.88.0.248:8080
   4
   #  curl 10.88.0.249:8080
   4
   #  curl 10.88.0.250:8080
   4
   ```

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

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

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

13.1.7.3. 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.

13.2. 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.

13.2.1. 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**.

```
    "args": [
      "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
sh-4.2#
```

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**.

13.3. 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.

13.3.1. 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",
    "Digest": "sha256:4e09c308a9ddf56c0ff6e321d135136eb04152456f73786a16166ce7cba7c904",
    "RepoTags": [
        "latest"
    ],
    "Created": "2018-06-16T17:27:13Z",
    "DockerVersion": "1.7.0",
    "Labels": {
        "Architecture": "x86_64",
        "Authoritative_Registry": "registry.access.redhat.com",
        "BZComponent": "rhel-server-docker",
        "Build_Host": "rcm-img01.build.eng.bos.redhat.com",
        "Name": "myubi8",
        "Release": "75",
        "Vendor": "Red Hat, Inc.",
        "Version": "8.0"
    },
    "Architecture": "amd64",
    "Os": "linux",
    "Layers": [
        "sha256:16dc1f96e3a1bb628be2e00518fec2bb97bd5933859de592a00e2eb7774b6ecf"
    ]
}
```

13.3.2. 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/
INFO[0000] Downloading
myubi8/blobs/sha256:16dc1f96e3a1bb628be2e00518fec2bb97bd5933859de592a00e2eb7774b6ecf
# ls /root/test
16dc1f96e3a1bb628be2e00518fec2bb97bd5933859de592a00e2eb7774b6ecf.tar manifest.json
```

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.

13.3.3. 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
INFO[0000] Downloading
myubi8/blobs/sha256:16dc1f96e3a1bb628be2e00518fec2bb97bd5933859de592a00e2eb7774b6ecf
# find .
./layers-myubi8-latest-698503105
./layers-myubi-latest-698503105/manifest.json
./layers-myubi8-latest-
698503105/16dc1f96e3a1bb628be2e00518fec2bb97bd5933859de592a00e2eb7774b6ecf.tar
```

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.

CHAPTER 14. USING THE CONTAINER-TOOLS API

The new REST based Podman 2.0 API replaces the old remote API for Podman that used the varlink library. The new API works in both a rootful and a rootless environment.

The Podman v2.0 RESTful API consists of the Libpod API providing support for Podman, and Docker-compatible API. With this new REST API, you can call Podman from platforms such as cURL, Postman, Google's Advanced REST client, and many others.

14.1. ENABLING THE PODMAN API USING SYSTEMD IN ROOT MODE

This procedure shows how to do the following:

1. Use systemd to activate the Podman API socket.
2. Use a Podman client to perform basic commands.

Prerequisities

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

```
# yum install podman-remote
```

Procedure

1. 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

1. Display system information of Podman:

```
# podman-remote info
```

2. Verify the link:

```
# ls -al /var/run/docker.sock
lrwxrwxrwx. 1 root root 23 Nov  4 10:19 /var/run/docker.sock -> /run/podman/podman.sock
```

Additional resources

- Podman v2.0 RESTful API - upstream documentation
- A First Look At Podman 2.0 API - article
- Sneak peek: Podman's new REST API - article

14.2. ENABLING THE PODMAN API USING SYSTEMD IN ROOTLESS MODE

This procedure shows how to use systemd to activate the Podman API socket and podman API service.

Prerequisites

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

```
# yum install podman-remote
```

Procedure

1. Create a rootless socket for Podman:

```
$ vim ~/.config/systemd/user/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. Create a rootless service for Podman:

```
$ vim ~/.config/systemd/user/podman.service
[Unit]
Description=Podman API Service
Requires=podman.socket
```

```
After=podman.socket
Documentation=man:podman-api(1)
StartLimitIntervalSec=0

[Service]
Type=oneshot
Environment=REGISTRIES_CONFIG_PATH=/etc/containers/registries.conf
ExecStart=/usr/bin/podman system service unix:///%t/podman.sock
TimeoutStopSec=30
KillMode=process

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

- 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

- Podman v2.0 RESTful API - upstream documentation
- A First Look At Podman 2.0 API - article
- Sneak peek: Podman's new REST API - article
- Exploring Podman RESTful API using Python and Bash - article

14.3. RUNNING THE PODMAN API MANUALLY

This procedure describes how to run the Podman API. This is useful for debugging API calls, especially when using the Docker compatibility layer.

Prerequisities

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

```
# yum install podman-remote
```

Procedure

1. Run the service for the REST API:

   ```
   # podman system service -t 0 --log-level=debug
   ```

   - The value of 0 means no timeout. The default endpoint for a rootful service is **unix:/run/podman/podman.sock**.
   - The **--log-level <level>** option sets the logging level. The standard logging levels are **debug**, **info**, **warn**, **error**, **fatal**, and **panic**.

2. In another terminal, display system information of Podman. The **podman-remote** command, unlike the regular **podman** command, communicates through the Podman socket:

   ```
   # podman-remote info
   ```

3. To troubleshoot the Podman API and display request and responses, use the **curl** comman. To get the information about the Podman installation on the Linux server in JSON format:

   ```
   # curl -s --unix-socket /run/podman/podman.sock http://d/v1.0.0/libpod/info | jq
       {
     "host": {
       "arch": "amd64",
       "buildahVersion": "1.15.0",
       "cgroupVersion": "v1",
       "conmon": {
         "package": "conmon-2.0.18-1.module+el8.3.0+7084+c16098dd.x86_64",
         "path": "/usr/bin/conmon",
         "version": "conmon version 2.0.18, commit:
   7fd3f71a218f8d3a7202e464252aeb1e942d17eb"
       },
       …
     "version": {
       "APIVersion": 1,
       "Version": "2.0.0",
       "GoVersion": "go1.14.2",
       "GitCommit": "",
       "BuiltTime": "Thu Jan  1 01:00:00 1970",
       "Built": 0,
       "OsArch": "linux/amd64"
     }
   }
   ```

   A **jq** utility is a command-line JSON processor.

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

   ```
   # curl -XPOST --unix-socket /run/podman/podman.sock -v 'http://d/v1.0.0/images/create?
   fromImage=registry.access.redhat.com%2Fubi8%2Fubi'
   ```

```
*   Trying /run/podman/podman.sock...
* Connected to d (/run/podman/podman.sock) port 80 (#0)
> POST /v1.0.0/images/create?fromImage=registry.access.redhat.com%2Fubi8%2Fubi
HTTP/1.1
> Host: d
> User-Agent: curl/7.61.1
> Accept: /
>
< HTTP/1.1 200 OK
< Content-Type: application/json
< Date: Tue, 20 Oct 2020 13:58:37 GMT
< Content-Length: 231
<
{"status":"pulling image () from registry.access.redhat.com/ubi8/ubi:latest,
registry.redhat.io/ubi8/ubi:latest","error":"","progress":"","progressDetail":
{},"id":"ecbc6f53bba0d1923ca9e92b3f747da8353a070fccbae93625bd8b47dbee772e"}
* Connection #0 to host d left intact
```

5. Display the pulled image:

```
# curl --unix-socket /run/podman/podman.sock -v 'http://d/v1.0.0/libpod/images/json' | jq
*   Trying /run/podman/podman.sock...
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0* Connected to d
(/run/podman/podman.sock) port 80 (0) > GET /v1.0.0/libpod/images/json HTTP/1.1 > Host: d
> User-Agent: curl/7.61.1 > Accept: / > < HTTP/1.1 200 OK < Content-Type: application/json
< Date: Tue, 20 Oct 2020 13:59:55 GMT < Transfer-Encoding: chunked < { [12498 bytes
data] 100 12485 0 12485 0 0 2032k 0 --:--:-- --:--:-- --:--:-- 2438k * Connection #0 to host d
left intact [ { "Id":
"ecbc6f53bba0d1923ca9e92b3f747da8353a070fccbae93625bd8b47dbee772e",
"RepoTags": [ "registry.access.redhat.com/ubi8/ubi:latest", "registry.redhat.io/ubi8/ubi:latest"
], "Created": "2020-09-01T19:44:12.470032Z", "Size": 210838671, "Labels": { "architecture":
"x86_64", "build-date": "2020-09-01T19:43:46.041620", "com.redhat.build-host": "cpt-
1008.osbs.prod.upshift.rdu2.redhat.com", ... "maintainer": "Red Hat, Inc.", "name": "ubi8", ...
"summary": "Provides the latest release of Red Hat Universal Base Image 8.", "url":
"https://access.redhat.com/containers//registry.access.redhat.com/ubi8/images/8.2-347",
      ...
    },
    "Names": [
      "registry.access.redhat.com/ubi8/ubi:latest",
      "registry.redhat.io/ubi8/ubi:latest"
    ],
    ...
    ]
  }
]
```

Additional resources

- Podman v2.0 RESTful API - upstream documentation
- Sneak peek: Podman's new REST API - article
- Exploring Podman RESTful API using Python and Bash - article

- **podman-system-service** man page

CHAPTER 15. 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