Red Hat Enterprise Linux 8

Building, running, and managing containers

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

Last Updated: 2019-05-07

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

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

Abstract

This guide describes how to work with Linux containers on RHEL 8 systems using command-line tools such as podman, buildah, skopeo and runc.

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 for multi-node systems, although single-node, all-in-one installations are supported as well. 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).

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, make sure you are viewing the documentation in the Multi-page HTML format. Highlight the part of text that you want to comment on. Then, click the **Add Feedback** pop-up that appears below the highlighted text, and follow the displayed instructions.

- For submitting more complex feedback, create a Bugzilla ticket:

  1. Go to the Bugzilla website.

  2. As the Component, use **Documentation**.

  3. Fill in the **Description** field with your suggestion for improvement. Include a link to the relevant part(s) of documentation.

  4. Click **Submit Bug**.

CHAPTER 1. STARTING WITH CONTAINERS

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

Red Hat Enterprise Linux implements Linux Containers using core technologies such as Control Groups (Cgroups) for Resource Management, Namespaces for Process Isolation, SELinux for Security, enabling secure multi-tenancy and reducing the potential for security exploits. All this is meant to provide you with an environment to producing and running enterprise-quality containers.

Red Hat OpenShift provides powerful command-line and Web UI tools for building, managing and running containers in units referred to as **pods**. However, there are times when you might want to build and manage individual containers and container images outside of OpenShift. Tools provided to perform those tasks that run directly on RHEL systems are described in this guide.

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

- **podman** - For directly managing pods and container images (run, stop, start, ps, attach, exec, and so on)
- **buildah** - For building, pushing and signing container images
- **skopeo** - For copying, inspecting, deleting, and signing images
- **runc** - For providing container run and build features to podman and buildah

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

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

1.1. RUNNING CONTAINERS WITHOUT DOCKER

Red Hat did not just remove the Docker container engine from OpenShift. It also removed the Docker container engine, along with the **docker** command, from Red Hat Enterprise Linux 8 entirely. For RHEL 8, Docker is not included and not supported.

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

- In the enterprise, the focus is not on running individual containers from the command line. The primary venue for running containers is a Kubernetes-based platform, such as OpenShift.
- By repositioning OpenShift as the project for running containers, container engines like Docker become just another component of OpenShift with no direct access by end users.
- Because the container engine in OpenShift is not meant to be used directly, it can be implemented with a limited feature set that focuses on doing everything that OpenShift needs, without having to implement lots of standalone features.

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

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

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

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

1.2. CHOOSING A RHEL ARCHITECTURE FOR CONTAINERS

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

- X86 64-bit (base and layered images) (no support for X86 32-bit)
- PowerPC 8 64-bit (base image and most layered images)
- PowerPC 9 64-bit (base image and most layered images)
- IBM s390x (base image and most layered images)
- ARM 64-bit (base image only)

The following table lists container images that are available for different architectures for RHEL 8.

Table 1.1. Red Hat container images and supported architectures

Image name	X86_64	PowerPC 8 & 9	s390x	ARM 64
ubi8/rhel	Yes	Yes	Yes	No
ubi8/rhel-minimal	Yes	Yes	Yes	No
ubi8/rhel-init	Yes	Yes	Yes	No
ubi8/rsyslog	Yes	Yes	Yes	No
ubi8/support-tools	Yes	Yes	Yes	Yes
ubi8/net-snmp	Yes	Yes	Yes	No
ubi8/ubi8-aarch64	No	No	No	Yes

1.3. GETTING CONTAINER TOOLS

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

- **podman** - Client tool for managing containers. Can replace most features of the **docker** command for working with individual containers and images.
- **buildah** - Client tool for building OCI-compliant container images.
- **skopeo** - Client tool for copying container images to and from container registries. Includes features for signing and authenticating images as well.
- **runc** - Container runtime client for running and working with Open Container Initiative (OCI) format containers.

Using the RHEL subscription model, if you want to create container images, you must properly register and entitle the host computer on which you build them. When you install packages, as part of the process of building a container, the build process automatically has access to entitlements available from the RHEL host. So it can get RPM packages from any repository enabled on that host.

1. **Install RHEL**: If you are ready to begin, you can start by installing a Red Hat Enterprise Linux system.
2. **Register RHEL**: Once RHEL is installed, register the system. You will be prompted to enter your user name and password. Note that the user name and password are the same as your login credentials for Red Hat Customer Portal.

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

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

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

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

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

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

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

1.4. ENABLING CONTAINER SETTINGS

No container engine (such as Docker or CRI-O) is required for you to run containers on your local system. However, configuration settings in the **/etc/containers/registries.conf** file let you define access to container registries when you work with container tools such as **podman** and **buildah**.

Here are example settings in the **/etc/containers/registries.conf** file:

```
[registries.search]
registries = ['registry.redhat.io', 'quay.io', 'docker.io']

[registries.insecure]
registries = []

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

By default, when you use **podman search** to search for images from a container registries, based on the **registries.conf** file, **podman** looks for the requested image in registry.redhat.io, quay.io, and docker.io, in that order.

To add access to a registry that doesn't require authentication (an insecure registry), you must add the name of that registry under the **[registries.insecure]** section. Any registries that you want to disallow from access from your local system need to be added under the **[registries.block]** section.

CHAPTER 2. WORKING WITH CONTAINER IMAGES

Using **podman**, you can run, investigate, start, stop, investigate, and remove container images.

2.1. PULLING IMAGES FROM REGISTRIES

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

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

The *<registry>* is a host that provides a container registry service on TCP *<port>*. Together, *<namespace>* and *<name>* identify a particular image controlled by *<namespace>* at that registry. 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 supports are registry.redhat.io (requiring authentication) and registry.access.redhat.com (requires no authentication, but is deprecated). For details on the transition to registry.redhat.io, see Red Hat Container Registry Authentication . Before you can pull containers from registry.redhat.io, you need to authenticate. For example:

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

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

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

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

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

```
REPOSITORY                      TAG    IMAGE ID      CREATED     SIZE
registry.redhat.io/ubi8/ubi       latest eb205f07ce7d  2 weeks ago 214MB
registry.redhat.io/rhel8/rsyslog  latest 85cfba5cd49c  2 weeks ago 234MB
```

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

2.2. INVESTIGATING IMAGES

Using **podman images** you can see which images have been pulled to your local system. To look at the metadata associated with an image, use **podman inspect**.

2.2.1. Listing images

To see which images have been pulled to your local system and are available to use, type:

```
# podman images
REPOSITORY                           TAG      IMAGE ID     CREATED       VIRTUAL SIZE
registry.redhat.io/rhel8/support-tools   latest   b3d6ce4e0043 2 days ago   234MB
registry.redhat.io/rhel8/rhel-init       latest   779a05997856 2 days ago   225MB
registry.redhat.io/ubi8/ubi              latest   a80dad1c1953 3 days ago   210MB
```

2.2.2. Inspecting local images

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

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

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

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

```
# podman inspect registry.redhat.io/ubi8/rhel
...
  "Cmd": [
     "/bin/bash"
  ],
  "Labels": {
     "architecture": "x86_64",
     "authoritative-source-url": "registry.redhat.io",
     "build-date": "2018-10-24T16:46:08.916139",
     "com.redhat.build-host": "cpt-0009.osbs.prod.upshift.rdu2.redhat.com",
     "com.redhat.component": "rhel-server-container",
     "description": "The Red Hat Enterprise Linux Base image is designed to be a fully
supported...
...
```

```
# podman inspect registry.redhat.io/rhel8/rsyslog
  "Cmd": [
    "/bin/rsyslog.sh"
  ],
  "Labels": {
    "License": "GPLv3",
    "architecture": "x86_64",
...
    "install": "docker run --rm --privileged -v /:/host -e HOST=/host \
       -e IMAGE=IMAGE -e NAME=NAME IMAGE /bin/install.sh",

...
    "run": "docker run -d --privileged --name NAME --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=IMAGE -e NAME=NAME --restart=always IMAGE /bin/rsyslog.sh",
    "summary": "A containerized version of the rsyslog utility
...
```

The rhel8/rhel container will execute the bash shell, if no other argument is given when you start it with **podman run**. If an Entrypoint were set, its value would be used instead of the Cmd value (and the value of Cmd would be used as an argument to the Entrypoint command).

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

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

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

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

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

```
# rpm -qa --root=/var/lib/containers/storage/overlay/65881e78.../merged
redhat-release-server-7.6-4.el7.x86_64
filesystem-3.2-25.el7.x86_64
basesystem-10.0-7.el7.noarch
ncurses-base-5.9-14.20130511.el7_4.noarch
```

```
glibc-common-2.17-260.el7.x86_64
nspr-4.19.0-1.el7_5.x86_64
libstdc++-4.8.5-36.el7.x86_64
```

2.2.3. Inspecting remote images

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

The following command inspects the **rhel-init** image from the Red Hat registry:

```
# skopeo inspect docker://registry.redhat.io/rhel8/rhel-init
{
   "Name": "registry.redhat.io/rhel8/rhel-init",
   "Digest": "sha256:53dfe24...",
   "RepoTags": [
      "8.0.0-9",
      "8.0.0",
      "latest"
   ],
   "Created": "2019-05-13T20:50:11.437931Z",
   "DockerVersion": "1.13.1",
   "Labels": {
      "architecture": "x86_64",
      "authoritative-source-url": "registry.access.redhat.com",
      "build-date": "2019-05-13T20:49:44.207967",
      "com.redhat.build-host": "cpt-0013.osbs.prod.upshift.rdu2.redhat.com",
      "com.redhat.component": "rhel-init-container",
      "description": "The Red Hat Enterprise Linux Init image is designed to be...
```

2.3. TAGGING IMAGES

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

registryhost/username/NAME:tag

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

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

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

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

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

```
# podman tag 474ff279782b jsmith/myrhel8
# podman images | grep 474ff279782b
rhel8           latest  474ff279782b  7 days ago  139.6 MB
myrhel8         latest  474ff279782b  7 months ago  139.6 MB
myrhel8         7.1     474ff279782b  7 months ago  139.6 MB
jsmith/myrhel8  latest  474ff279782b  7 months ago  139.6 MB
```

Above, you can see all the image names assigned to the single image ID.

2.4. SAVING AND IMPORTING IMAGES

If you want to save a container image you created, you can use **podman save** to save the image to a tarball. After that, you can store it or send it to someone else, then reload the image later to reuse it. Here is an example of saving an image as a tarball:

```
# podman save -o myrsyslog.tar registry.redhat.io/rhel8/rsyslog:latest
Getting image source signatures
Copying blob sha256:dd7d5adb457...
Writing manifest to image destination
Storing signatures
# ls
myrsyslog.tar
```

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

```
# cat myrsyslog.tar | podman import - rhel8/myrsyslog
# podman images
```

2.5. REMOVING IMAGES

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

```
# podman rmi rhel-init
7e85c34f126351ccb9d24e492488ba7e49820be08fe53bee02301226f2773293
```

You can remove multiple images on the same command line:

```
# podman rmi registry.redhat.io/rhel8/rsyslog support-tools
46da8e23fa1461b658f9276191b4f473f366759a6c840805ed0c9ff694aa7c2f
85cfba5cd49c84786c773a9f66b8d6fca04582d5d7b921a308f04bb8ec071205
```

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

```
# podman rmi $(podman images -a -q)
1ca061b47bd70141d11dcb2272dee0f9ea3f76e9afd71cd121a000f3f5423731
ed904b8f2d5c1b5502dea190977e066b4f76776b98f6d5aa1e389256d5212993
```

```
83508706ef1b603e511b1b19afcb5faab565053559942db5d00415fb1ee21e96
```

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

```
# podman rmi $(podman images -a -q)
unable to delete eb205f07ce7d0bb63bfe5603ef8964648536963e2eee51a3ebddf6cfe62985f7 (must
force) - image is referred to in multiple tags
unable to delete eb205f07ce7d0bb63bfe5603ef8964648536963e2eee51a3ebddf6cfe62985f7 (must
force) - image is referred to in multiple tags

# podman rmi -f eb205f07ce7d
eb205f07ce7d0bb63bfe5603ef8964648536963e2eee51a3ebddf6cfe62985f7
```

CHAPTER 3. WORKING WITH CONTAINERS

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

3.1. RUNNING CONTAINERS

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

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

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

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

```
# podman run --rm registry.redhat.io/ubi8/ubi cat /etc/os-release
NAME="Red Hat Enterprise Linux"
VERSION="8.0 (Ootpa)"
ID="rhel"
ID_LIKE="fedora"
VERSION_ID="8.0"
PLATFORM_ID="platform:el8"
PRETTY_NAME="Red Hat Enterprise Linux 8.0 (Ootpa)"
ANSI_COLOR="0;31"
CPE_NAME="cpe:/o:redhat:enterprise_linux:8.0:latest"
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=8.0
REDHAT_SUPPORT_PRODUCT="Red Hat Enterprise Linux"
REDHAT_SUPPORT_PRODUCT_VERSION="8.0"
...
```

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

```
# podman run --rm registry.access.redhat.com/rhel8/rsyslog  ls /root/buildinfo
Dockerfile-rhel8-rsyslog-8
```

Now you know what the Dockerfile is called, you can list its contents:

```
# podman run --rm registry.access.redhat.com/rhel8/rsyslog \
    cat /root/buildinfo/Dockerfile-rhel8-rsyslog-8
FROM sha256:eb205f07ce7d0bb63bfe560...
LABEL maintainer="Red Hat, Inc."

RUN INSTALL_PKGS="\
rsyslog \
rsyslog-gnutls \
rsyslog-gssapi \
rsyslog-mysql \
rsyslog-pgsql \
rsyslog-relp \
" && yum -y install $INSTALL_PKGS && rpm -V --nosize
    --nofiledigest --nomtime --nomode $INSTALL_PKGS && yum clean all
LABEL com.redhat.component="rsyslog-container"
LABEL name="rhel8/rsyslog"
LABEL version="8.0"
...
```

EXAMPLE #3 (Run a shell inside the container) Using a container to launch a bash shell lets you look inside the container and change the contents. This sets the name of the container to **mybash**. The **-i** creates an interactive session and **-t** opens a terminal session. Without **-i**, the shell would open and then exit. Without **-t**, the shell would stay open, but you wouldn't be able to type anything to the shell.

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

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

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

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

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

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

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

_

```
# podman run --name="log_test" -v /dev/log:/dev/log --rm \
     registry.redhat.io/ubi8/ubi logger "Testing logging to the host"
# journalctl -b | grep Testing
Nov 12 20:00:10 ubi8 root[17210]: Testing logging to the host
```

3.2. INVESTIGATING RUNNING AND STOPPED CONTAINERS

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

3.2.1. Listing containers

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

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

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

If there are containers that are not running, but were not removed (--rm option), the containers are still hanging around and can be restarted. The **podman ps -a** command shows 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
```

3.2.2. Inspecting containers

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

```
# podman inspect 74b1da000a11
   ...
  "ID": "74b1da000a114015886c557deec8bed9dfb80c888097aa83f30ca4074ff55fb2",
  "Created": "2018-11-13T10:30:31.884673073-05:00",
  "Path": "/bin/rsyslog.sh",
  "Args": [
      "/bin/rsyslog.sh"
  ],
  "State": {
      OciVersion": "1.0.1-dev",
      Status": "running",
      Running": true,
   ...
```

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

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

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

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

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

3.2.3. Investigating within a container

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

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

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

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

   ```
   # podman ps
   CONTAINER ID   IMAGE          COMMAND               CREATED        STATUS         PORTS
   NAMES
   74b1da000a11   rsyslogd:latest "/usr/sbin/rsyslogd -n 6 minutes ago Up 6 minutes
   0.0.0.0:514->514/tcp   rsyslogd
   ```

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

   ```
   # podman exec -it 74b1da000a11 /bin/bash
   [root@74b1da000a11 /]# cat /etc/redhat-release
   Red Hat Enterprise Linux release 8.0
   [root@74b1da000a11 /]# ps -ef
   ```

```
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 15:30 ?        00:00:00 /usr/sbin/rsyslogd -n
root         8     0  6 16:01 pts/0    00:00:00 /bin/bash
root        21     8  0 16:01 pts/0    00:00:00 ps -ef
[root@74b1da000a11 /]# df -h
Filesystem      Size  Used Avail Use% Mounted on
overlay          39G  2.5G   37G   7% /
tmpfs            64M     0   64M   0% /dev
tmpfs           1.5G  8.7M  1.5G   1% /etc/hosts
shm              63M     0   63M   0% /dev/shm
tmpfs           1.5G     0  1.5G   0% /sys/fs/cgroup
tmpfs           1.5G     0  1.5G   0% /proc/acpi
tmpfs           1.5G     0  1.5G   0% /proc/scsi
tmpfs           1.5G     0  1.5G   0% /sys/firmware
[root@74b1da000a11 /]# uname -r
4.18.0-27.el8.x86_64
[root@74b1da000a11 /]# rpm -qa | more
redhat-release-server-7.6-4.el7.x86_64
filesystem-3.2-25.el7.x86_64
basesystem-10.0-7.el7.noarch
ncurses-base-5.9-14.20130511.el7_4.noarch
...
bash-4.2# free -m
      total  used  free  shared  buff/cache   available
Mem:   2926   224  1303      8       1398        2526
Swap:     0     0     0
[root@74b1da000a11 /]# exit
```

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

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

3.3. STARTING AND STOPPING CONTAINERS

If you ran a container, but didn't remove it (**--rm**), that container is stored on your local system and ready to run again. To start a previously run container that wasn't removed, use the **start** option. To stop a running container, use the **stop** option.

3.3.1. Starting containers

A container that doesn't need to run interactively can sometimes be restarted after being stopped with only the **start** option and the container ID or name. For example:

```
# podman start myrhel_httpd
myrhel_httpd
```

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

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

3.3.2. Stopping containers

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

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

The **stop** option sends a SIGTERM signal to terminate a running container. If the container doesn't stop after a grace period (10 seconds by default), **podman** sends a SIGKILL signal. You could also use the **podman kill** command to kill a container (SIGKILL) or send a different signal to a container. Here's 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
```

3.4. REMOVING CONTAINERS

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

```
# podman rm goofy_wozniak
```

You can remove multiple containers on the same command line:

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

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

```
# podman rm $(podman ps -a -q)
```

CHAPTER 4. USING RED HAT UNIVERSAL BASE IMAGES (STANDARD, MINIMAL, AND RUNTIMES)

Red Hat Enterprise Linux (RHEL) base images are meant to form the foundation for the container images you build. For RHEL 8, all Red Hat base images are available as new Universal Base Images (UBI). These include versions of RHEL standard, minimal, init, and Red Hat Software Collections that are all now freely available and redistributable. Characteristics of RHEL base images include:

- **Supported**: Supported by Red Hat for use with your containerized applications. Contains the same secured, tested, and certified software packages you have in Red Hat Enterprise Linux.
- **Cataloged**: Listed in the Red Hat Container Catalog , where you can find descriptions, technical details, and a health index for each image.
- **Updated**: Offered with a well-defined update schedule, so you know you are getting the latest software (see Red Hat Container Image Updates).
- **Tracked**: Tracked by errata, to help you understand the changes that go into each update.
- **Reusable**: Only need to be downloaded and cached in your production environment once, where each base image can be reused by all containers that include it as their foundation.

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

4.1. WHAT ARE RED HAT BASE IMAGES?

Red Hat provides multiple base images that you can use as a starting point for your own images. These images are available through the Red Hat Registry (registry.access.redhat.com and registry.redhat.io) and described in the Red Hat Container Catalog .

For RHEL 8, there are standard, minimal and init base image available. Red Hat also provides a set of Red Hat Software Collections images that you can build on when you are creating containers for applications that require specific runtimes. These include python, php, nodejs, and others. All of these images RHEL 8 images are UBI images, which means that you can freely obtain and redistribute them.

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

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

4.1.1. Using standard Red Hat base images

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

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

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

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

- **python**: Python runtime libraries and modules (currently Python 2.7) are included in the standard base image.

4.1.2. Using minimal Red Hat base images

The **ubi8-minimal** images are stripped-down RHEL images to use when a bare-bones base image in desired. If you are looking for the smallest possible base image to use as part of the larger Red Hat ecosystem, you can start with these minimal images.

RHEL minimal images provide a base for your own container images that is less than half the size of the standard image, while still being able to draw on RHEL software repositories and maintain any compliance requirements your software has.

Here are some features of the minimal base images:

- **Small size**: Minimal images are about 75M on disk and 28M compressed. This makes it less than half the size of the standard images.

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

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

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

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

4.1.3. Using Init Red Hat base images

The UBI **ubi8-init** images contains 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.

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

4.2. HOW ARE UBI IMAGES DIFFERENT?

UBI images were created so you can build your container images on a foundation of official Red Hat software that can be freely shared and deployed. From a technical perspective, they are nearly identical to legacy Red Hat Enterprise Linux images, which means they have great security, performance, and life cycles, but they are released under a different End User License Agreement. Here are some attributes of Red Hat UBI images:

- **Built from a subset of RHEL content** Red Hat Universal Base images are built from a subset of normal Red Hat Enterprise Linux content. All of the content used to build selected UBI images is released in a publicly available set of yum repositories. This lets you install extra packages, as well as update any package in UBI base images.
- **Redistributable**: The intent of UBI images is to allow Red Hat customers, partners, ISVs, and others to standardize on one container base image, allowing users to focus on application needs instead of distribution rules. These images can be shared and run in any environment capable of running those images. As long as you follow some basic guidelines, you will be able to freely redistribute your UBI-based images.
- **Base and RHSCL images** Besides the three types of base images, UBI versions of some Red Hat Software Collections (RHSCL) runtime images are available as well. These RHSCL images provide a foundation for applications that can benefit from standard, supported runtimes such as python, php, nodejs, and ruby.
- **Enabled yum repositories**: The following yum repositories are enabled within each RHEL 8 UBI image:
  - The first (**rhubi-8.0-baseos**) holds the redistributable subset of RHEL packages you can include in your container.
  - The second (**rhubi-8.0-appstream**) holds Red Hat Software Collections packages that you can add to a UBI image to help you standardize the environments you use with applications that require particular runtimes.

Although the legacy RHEL 7 base images will continue to be supported, UBI images are recommended going forward. For that reason, examples in the rest of this chapter are done with RHEL 8 UBI images.

4.3. GET UBI IMAGES

The following table shows the UBI base images currently available. Notice that some of those images have multiple names you can use to refer to them.

Table 4.1. Available Red Hat Universal Base Images

Registry names	Namespace	Image	RHEL version
registry.access.redhat.com registry.redhat.io	/	ubi8	8.0.0
		ubi8-minimal	8.0.0
		ubi8-init	8.0.0
registry.access.redhat.com registry.redhat.io	ubi8/	ubi	8.0.0
		ubi-init	8.0.0
		ubi-minimal	8.0.0
		s2i-core	8.0.0
		s2i-base	8.0.0
		dotnet-21-runtime	8.0.0
		dotnet-21	8.0.0
		perl-526	8.0.0
		php-72	8.0.0
		nodejs-10	8.0.0
		ruby-25	8.0.0
		python-27	8.0.0
		python-36	8.0.0
registry.redhat.io	rhel8	s2i-core	8.0.0
		s2i-base	8.0.0
		dotnet-21-runtime	8.0.0
		dotnet-21	8.0.0
		nginx-114	8.0.0

Registry names	Namespace	Image	RHEL version
		perl-526	8.0.0
		php-72	8.0.0
		nodejs-10	8.0.0
		httpd-24	8.0.0
		redis-5	8.0.0
		varnish-6	8.0.0
		ruby-25	8.0.0
		python-27	8.0.0
		python-36	8.0.0
		memcached	8.0.0
		net-snmp	8.0.0
		sssd	8.0.0
		ipa-server	8.0.0
		support-tools	8.0.0
		rsyslog	8.0.0
		GCC	8.0.0
		rust-toolset	8.0.0
		go-toolset	8.0.0
		llvm-toolset	8.0.0
		postgresql-10	8.0.0
		postgresql-96	8.0.0
		mysql-80	8.0.0
		mariadb-103	8.0.0

Registry names	Namespace	Image	RHEL version
registry.access.redhat.com registry.redhat.io	/	ubi7	7.6.4
		ubi7-minimal	7.6.4
		ubi7-init	7.6.4
registry.access.redhat.com registry.redhat.io	ubi7/	ubi	7.6.4
		ubi-init	7.6.4
		ubi-minimal	7.6.4
		php-72	7.6.4
		nodejs-8	7.6.4
		ruby-25	7.6.4
		python-27	7.6.4
		python-36	7.6.4
		s2i-core	7.6.4
		s2i-base	7.6.4

4.4. PULL UBI IMAGES

To pull UBI images to your system so you can use them with tools such as podman, buildah or skopeo, type the following:

```
# podman pull registry.access.redhat.com/ubi8/ubi:latest
# podman pull registry.access.redhat.com/ubi/ubi-minimal:latest
```

To check that the images are available on your system, type:

```
# podman images
REPOSITORY                                   TAG    IMAGE ID       CREATED       SIZE
registry.access.redhat.com/ubi8/ubi-minimal  latest  c94a444803e3  8 hours ago  80.9 MB
registry.access.redhat.com/ubi8/ubi          latest  40b488f87628  17 hours ago 214 MB
```

When pulled in this way, images are available and usable by **podman**, **buildah**, **skopeo** and the CRI-O container image, but they are not available to the Docker service or **docker** command. To use these images with Docker, you can run **docker pull** instead.

4.5. REDISTRIBUTING UBI IMAGES

After you pull a UBI image, you are free to push it to your own registry and share it with others. You can upgrade or add to that image from UBI yum repositories as you like. Here is an example of how to push a UBI image to your own or another third-party repository:

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

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

4.6. RUN UBI IMAGES

To start a container from a UBI image and run the bash shell in that image (so you can look around inside), do the following (type exit when you are done):

```
# podman run --rm -it registry.access.redhat.com/ubi8/ubi-minimal:latest /bin/bash
[root@da9213157c51 /]#
# podman run --rm -it registry.access.redhat.com/ubi8/ubi:latest /bin/bash
bash-4.2#
```

While in the container:

- Run **rpm -qa** to see a list of package inside each container.
- Type **yum list available** to see packages available to add to the image from the UBI yum repos. (The yum command is not available in the **ubi-minimal** containers.)
- Get source code, as described in the "Getting UBI Container Image Source Code," later in this chapter.

On systems that include the Docker service, you can use **docker run** instead.

4.7. ADD SOFTWARE TO A RUNNING UBI CONTAINER

UBI images are built from 100% 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 repos. Refer to the “Build a UBI-based image” for more permanent ways of building UBI-based images.

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

4.7.1. Adding software to a UBI container (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 repos. So the full set of Red Hat packages is available. From the UBI minimal container, all UBI repos are enabled by default, but no repos are enabled from the host by default.

4.7.2. Adding software inside the standard UBI container

To ensure the containers you build can be redistributed, disable subscription management in the standard UBI image when you add software. If you disable the subscription-manager plugin, only packages from the freely available repos 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 --disableplugin=subscription-manager bzip2
```

To add software inside a standard UBI container that is in the RHEL server repo, but not in UBI repos, leave the subscription-manager plugin intact and just install the package:

```
# yum install zsh
```

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

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

WARNING

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

4.7.3. 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 repos available on a subscribed host that are not part of a UBI yum repo, you would have to explicitly enable those repos. For example:

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

WARNING

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

4.7.4. Adding software to a UBI container (unsubscribed host)

To add software packages to a running container that is either on an unsubscribed RHEL host or some other Linux system, you don't have to disable the subscription-manager plugin. For example:

```
# yum install bzip2
```

To install that package on a subscribed RHEL host from the UBI minimal container, type:

```
# microdnf install bzip2
```

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

4.7.5. Build a UBI-based image

You can build UBI-based container images in the same way you build other images, with one exception. You should disable Red Hat subscriptions 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's an example of creating a UBI-based Web server container from a Dockerfile with the **buildah** utility:

NOTE

For ubl8/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 --disableplugin=subscription-manager -y && rm -rf /var/cache/yum
RUN yum install --disableplugin=subscription-manager 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 --disableplugin=subscription-manager -y
. . .
No packages marked for update
STEP 5: RUN yum install --disableplugin=subscription-manager 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
```

4.7.6. Using Red Hat Software Collections runtime images

Red Hat Software Collections 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 some already updated as UBI images. Each of these images include additional software you might want to use for specific runtime environments.

So, 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 software collections image.

Here are a few examples of Red Hat Software Collections 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 container 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.

Red Hat Software Collections container images are updated every time RHEL base images are updated. 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 .

4.7.7. Getting UBI Container Image Source Code

You can download the source code for all UBI base images (excluding the minimal images) by starting up those images with a bash shell and running the following set of commands from inside that container:

```
# yum install yum-utils -y
# for i in `rpm -qa`
    do
      yumdownloader --source $i
    done
```

The source code RPM for each binary RPM package is downloaded to the current directory. Because the UBI minimal images include a subset of RPMs from the regular UBI images, running the **yumdownloader** loop just shown will get you the minimal image packages as well.

4.7.8. Tips and tricks for using UBI images

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

- Hundreds of RPM packages used in existing Red Hat Software Collections 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.

4.7.9. How to request new features in UBI?

Red Hat partners and customers can request new features, including package requests, by filing 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. See also: Red Hat Bugzilla Queue

4.7.10. How to file a support case for UBI?

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

See also: Open a Support Case

CHAPTER 5. 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.

5.1. 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 dnf) 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

5.1.1. Installing Buildah

The buildah package is available with the container-tools module in RHEL 8. 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.

5.2. GETTING IMAGES WITH BUILDAH

To get a container image to use with **buildah**, use the **buildah from** command. 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
rhel-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 rhel-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 (rhel-working-container). Here's an example of how to execute a command from that container:

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

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

5.3. 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, but doesn't create a working container, as demonstrated when you run **buildah containers** below:

```
# buildah images
IMAGE ID     IMAGE NAME          CREATED AT          SIZE
1456eedf8101 registry.redhat.io/ubi8/ubi:latest
                           May 15, 2019 10:15  214 MB
1d87ff386090 localhost/myecho:latest
                           may 15, 2019 13:20  214 MB
# buildah containers
```

Next, you can make the image into a container and run it, to make sure it is working.

5.3.1. Running a container with Buildah

To check that the image you built previously works, you need to create a working container from the image, then use **podman run** to run the working container.

```
# buildah from myecho
myecho-working-container
# buildah containers
CONTAINER ID  BUILDER  IMAGE ID     IMAGE NAME             CONTAINER NAME
43236f6fd1f8     *     1d87ff386090 localhost/myecho:latest myecho-working-container

# podman run myecho-working-container
This container works!
```

The steps just shown used the image (myecho) to create a container (myecho-working-container). After that, **buildah containers** showed the container exists and **podman run** ran the container, producing the output: This container works!

5.3.2. Inspecting a container with Buildah

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

```
# buildah inspect 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.

5.4. 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.

5.4.1. 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
            rhel-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 myecho2-working-container
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
```

5.4.2. 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:

```
# podman 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
```

5.5. 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:

- With a scratch container, you can simply copy executables that have no dependencies to 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 **rhel** or **rhel-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. In the example, instead of committing the image to Buildah (containers-storage which stores locally in /var/lib/containers), we illustrate how to commit the image so it can be managed by the local Docker service (docker-daemon which stores locally in /var/lib/docker). You could just have easily committed it to Buildah, which would let you then push it to a Docker service (docker), a local OSTree repository (ostree), or other OCI-compliant storage (oci). (Type **man buildah push** for details.)

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):

```
# rpm --root $scratchmnt --initdb
# yum install yum-utils       (if not already installed)
# yumdownloader --destdir=/tmp redhat-release-server
# rpm --root $scratchmnt -ihv /tmp/redhat-release-server*.rpm
```

Install the httpd service to the scratch directory:

```
# yum install -y --installroot=$scratchmnt 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 docker-daemon:myhttpd:latest
```

By default, the **buildah commit** command adds the docker.io repository name to the image name and copies the image to the storage area for your local Docker service (/var/lib/docker). For now, you can use the Image ID to run the new image as a container with the **docker** command:

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

5.6. 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
```

5.7. 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.

5.7.1. 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!
```

5.7.2. 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 6. RUNNING CONTAINERS AS SYSTEMD SERVICES WITH 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. It was not originally designed to bring up an entire Linux system or manage services for such things as start-up order, dependency checking, and failed service recovery. That is the job of a full-blown initialization system like systemd.

Red Hat has become a leader in integrating containers with systemd, so that OCI and Docker-formatted containers built by Podman can be managed in the same way that other services and features are managed in a Linux system. This chapter describes how you can use the systemd initialization service to work with containers in two different ways:

- **Starting Containers with systemd**: By setting up a systemd unit file on your host computer, you can have the host automatically start, stop, check the status, and otherwise manage a container as a systemd service.

- **Starting services within a container using systemd** Many Linux services (Web servers, file servers, database servers, and so on) are already packaged for Red Hat Enterprise Linux to run as systemd services. If you are using the latest RHEL container image, you can set the RHEL container image to start the systemd service, then automatically start selected services within the container when the container starts up.

The following two sections describe how to use systemd container in those ways.

6.1. STARTING CONTAINERS WITH SYSTEMD

When you set up a container to start as a systemd service, you can define the order in which the containerized service runs, check for dependencies (like making sure another service is running, a file is available or a resource is mounted), and even have a container start by using the runc command.

This section provides an example of a container that is configured to run directly on a RHEL or RHEL Atomic Host system as a systemd service.

1. Get the image you want to run on your system. For example, to use the redis service from docker.io, run the following command:

```
# podman pull docker.io/redis
```

2. Open Selinux permission. If SELinux is enabled on your system, you must turn on the **container_manage_cgroup** boolean to run containers with systemd as shown here (see the Containers running systemd solution for details):

```
# setsebool -P container_manage_cgroup on
```

3. Run the image as a container, giving it a name you want to use in the systemd service file. For example, to name the running redis container redis_server, type the following:

```
# podman run -d --name redis_server -p 6379:6379 redis
```

4. Configure the container as a systemd service by creating the unit configuration file in the **/etc/systemd/system/** directory. For example, the contents of the **/etc/systemd/system/redis-container.service** can look as follows (note that redis_server matches the name you set on the

podman run line):

```
[Unit]
Description=Redis container

[Service]
Restart=always
ExecStart=/usr/bin/podman start -a redis_server
ExecStop=/usr/bin/podman stop -t 2 redis_server

[Install]
WantedBy=local.target
```

5. After creating the unit file, to start the container automatically at boot time, type the following:

```
# systemctl enable redis-container.service
```

6. Once the service is enabled, it will start at boot time. To start it immediately and check the status of the service, type the following:

```
# systemctl start redis-container.service
# systemctl status redis-container.service
* redis-container.service - Redis container
   Loaded: loaded (/etc/systemd/system/redis-container.service; enabled; vendor preset:
disabled)
   Active: active (running) since Fri 2019-03-15 16:22:55 EDT; 6s ago
 Main PID: 1540 (podman)
    Tasks: 8 (limit: 2353)
   Memory: 7.7M
   CGroup: /system.slice/redis-container.service
           └─1540 /usr/bin/podman start -a redis_server

Mar 15 16:22:55 localhost.localdomain systemd[1]: Started Redis container.
```

To learn more about configuring services with systemd, refer to the System Administrator’s Guide chapter called Managing Services with systemd.

6.2. STARTING SERVICES WITHIN A CONTAINER USING SYSTEMD

A package with the systemd initialization system is included in the official Red Hat Enterprise Linux Init base image named registry.redhat.io/ubi8/ubi. This means that applications created to be managed with systemd can be started and managed inside a container. A container running systemd will:

- Start the /sbin/init process (the systemd service) to run as PID 1 within the container.
- Start all systemd services that are installed and enabled within the container, in order of dependencies.
- Allow systemd to restart services or kill zombie processes for services started within the container.

The general steps for building a container that is ready to be used as a systemd services is:

- Install the package containing the systemd-enabled service inside the container. This can include dozens of services that come with RHEL, such as Apache Web Server (httpd), FTP server (vsftpd), Proxy server (squid), and many others. For this example, we simply install an Apache (httpd) Web server.

- Use the systemctl command to enable the service inside the container.
- Add data for the service to use in the container (in this example, we add a Web server test page). For a real deployment, you would probably connect to outside storage.
- Expose any ports needed to access the service.
- Set /sbin/init as the default process to start when the container runs

In this example, we build a container by creating a Dockerfile that installs and configures a Web server (httpd) to start automatically by the systemd service (/sbin/init) when the container is run on a host system.

1. **Create Dockerfile**: In a separate directory, create a file named Dockerfile with the following contents:

   ```
   FROM ubi8/ubi
   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 Dockerfile installs the httpd package, enables the httpd service to start at boot time (i.e. when the container starts), 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**: From the directory containing the Dockerfile, type the following:

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

3. **Open Selinux permission**. If SELinux is enabled on your system, you must turn on the **container_manage_cgroup** boolean to run containers with systemd as shown here (see the Containers running systemd solution for details):

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

4. **Run the container**: Once the container is built and named mysysd, type the following to run the container:

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

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

5. **Check that the container is running** To make sure that the container is running and that the service is working, type the following commands:

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

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

At this point, you have a container that starts up a Web server as a systemd service inside the container. Install and run any services you like in this same way by modifying the Dockerfile and configuring data and opening ports as appropriate.

CHAPTER 7. CONTAINER COMMAND-LINE REFERENCE

7.1. PODMAN

The **podman** command 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 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 container, events, image, network, node, plugin (**podman** does not support plugins), port, rename (use rm and create to rename containers with **podman**), secret, service, stack, swarm (**podman** does not support Docker Swarm), system, and volume (for **podman**, create volumes on the host, then mount in a container). The container and image options are used to run subcommands that are used directly in **podman**.
- The following features are currently in development for **podman**:
  - To interact programmatically with **podman**, a remote API for Podman is being developed using a technology called varlink. This will let **podman** listen for API requests from remote tools (such as the RHEL 8 web console or the **atomic** command) and respond to them.
  - Support for user namespaces is just on the horizon. This feature will let you run a container as one user or group (for example, uid 0) inside the container and another user (for example, uid 1000000) outside the container. See User namespaces support in Podman for details.

- A feature in development will allow **podman** to run and manage a Pod (which may consist of multiple containers and some metadata) without Kubernetes or OpenShift being active. (However, **podman** is not expected to do some of Kubernetes' more advanced features, such as scheduling pods across clusters).

7.1.1. 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 7.1. 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		

7.1.2. Trying basic podman commands

Because the use of **podman** mirrors the features and syntax of the **docker** command, you can refer to Working with Docker Formatted Container Images for examples of how to use those options to work with containers. Simply replace **docker** with **podman** in most cases. Here are some examples of using **podman**.

7.1.3. Pull a container image to the local system

```
# podman pull registry.redhat.io/ubi8/ubi
Trying to pull registry.redhat...Getting image source signatures
Copying blob sha256:d1fe25896eb5cbcee...
```

```
Writing manifest to image destination
Storing signatures
fd1ba0b398a82d56900bb798c...
```

7.1.4. List local container images

```
# podman images
REPOSITORY                   TAG      IMAGE ID       CREATED        SIZE
registry.redhat.io/ubi8/ubi  latest   de9c26f23799   5 weeks ago    80.1MB
registry.redhat.io/ubi8/ubi  latest   fd1ba0b398a8   5 weeks ago    211MB
```

7.1.5. Inspect a container image

```
# podman inspect registry.redhat.io/ubi8/ubi | less
[
    {
        "Id": "4bbd153adf8487a8a5114af0d6...",
        "Digest": "sha256:9999e735605c73f...",
        "RepoTags": [
            "registry.access.redhat.com/ubi8/ubi:latest"
        ],
        "RepoDigests": [
            "registry.access.redhat.com/ubi8/ubi/rhel@sha256:9999e7356...
```

7.1.6. Run a container image

Run a container that opens a shell inside the container:

```
# podman run -it registry.redhat.io/ubi8/ubi /bin/bash
[root@8414218c04f9 /]# ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 13:48 pts/0    00:00:00 /bin/bash
root        21     1  0 13:49 pts/0    00:00:00 ps -ef
[root@8414218c04f9 /]# exit
#
```

7.1.7. List containers that are running or have exited

```
# podman ps -a
CONTAINER ID   IMAGE                                      COMMAND
   CREATED AT                    STATUS                    PORTS NAMES
440becd26893   registry.redhat.io/ubi8/ubi-minimal:latest  /bin/bash
   2018-05-10 09:02:52 -0400 EDT   Exited (0) About an hour ago  happy_hodgkin
8414218c04f9   registry.redhat.io/ubi8/ubi:latest          /bin/bash
   2018-05-10 09:48:07 -0400 EDT   Exited (0) 14 minutes ago     nostalgic_boyd
```

7.1.8. Remove a container or image

Remove a container by its container ID:

```
# podman rm 440becd26893
```

Remove a container image by its image ID or name (use -f to force):

```
# podman rmi registry.redhat.io/ubi8/ubi
# podman rmi de9c26f23799
# podman rmi -f registry.redhat.io/ubi8/ubi:latest
```

7.1.9. Build a container

```
# cat Dockerfile
FROM registry.redhat.io/ubi8/ubi
ENTRYPOINT "echo 'Podman built this container.'"

# podman build -t podbuilt .
STEP 1: FROM registry.access...
...
Writing manifest to image destination
Storing signatures
91e043c11617c08d4f8...

# podman run podbuilt
Podman build this container.
```

7.2. RUNC

"runC" is a lightweight, portable implementation of the Open Container Initiative (OCI) container runtime specification. 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.

7.2.1. 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".

7.3. 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.
- 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:5000/myubi8
# podman push localhost:5000/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.

7.3.1. 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:5000), 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:5000/myubi8** to a container registry running on your local system, the following command inspects that image:

```
# skopeo inspect docker://localhost:5000/myubi8
{
    "Name": "localhost:5000/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"
    ]
}
```

7.3.2. 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:5000/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 begin 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.

7.3.3. 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.jason file) in the current directory. For example

```
# skopeo layers docker://localhost:5000/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 8. 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