



# Red Hat Enterprise Linux Atomic Host 7

## Managing Containers

Managing Containers





## Legal Notice

Copyright © 2018 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

## Abstract

Manage regular and super-privileged containers with systemd, runc, skopeo, and buildah

## Table of Contents

<b>CHAPTER 1. MANAGING STORAGE WITH DOCKER-FORMATTED CONTAINERS</b> .....	<b>5</b>
1.1. OVERVIEW	5
1.2. USING CONTAINER-STORAGE-SETUP	5
1.2.1. LVM thin pool in the volume group containing the root volume	5
1.2.2. LVM thin pool in a user specified volume group	6
1.2.3. Setting up a volume group and LVM thin pool on user specified block device	6
1.3. MANAGING STORAGE IN RED HAT ENTERPRISE LINUX	6
1.3.1. How to Leave Space in the Volume Group Backing Root During Installation	6
1.3.1.1. GUI Installation	6
1.3.1.2. Kickstart Installation	7
1.4. MANAGING STORAGE IN RED HAT ENTERPRISE LINUX ATOMIC HOST	7
1.4.1. Contents of the Atomic Host filesystem	7
1.4.2. Changing the Default Size of the Root Partition During Installation	8
1.4.3. Changing the Size of the Root Partition After Installation	9
1.4.3.1. How to extend the Root Partition to use free space in volume group	9
1.4.3.2. How to Add Additional Storage to the Host and Extend the Root Partition	9
1.4.3.3. How to Extend the Root Partition Without Adding More Storage	10
1.5. CHANGING STORAGE CONFIGURATION	10
1.6. USING THE OVERLAY GRAPH DRIVER	10
1.7. INCREASING THE BASE DEVICE SIZE	13
1.8. RESETTING STORAGE FOR CONTAINERS	14
1.9. STORAGE BACKUP GUIDELINES	15
1.10. ADDITIONAL INFORMATION ABOUT STORAGE	15
<b>CHAPTER 2. SIGNING CONTAINER IMAGES</b> .....	<b>16</b>
2.1. GETTING CONTAINER SIGNING SOFTWARE	17
2.2. CREATING IMAGE SIGNATURES	17
2.2.1. Create GPG Keys	17
2.2.2. Creating an Image Signature	18
2.3. SET UP TO DO IMAGE SIGNING	18
2.4. CREATING A SIGNATURE FOR AN IMAGE IN A REPOSITORY	19
2.5. CREATING AN IMAGE SIGNATURE AT PUSH TIME	19
2.6. SHARING THE SIGNATURE STORE	20
2.7. VALIDATING AND TRUSTING SIGNED IMAGES	20
2.8. VALIDATING SIGNED IMAGES FROM RED HAT	21
2.9. UNDERSTANDING IMAGE SIGNING CONFIGURATION FILES	22
2.9.1. policy.json file	22
2.9.2. whatever.yaml	23
<b>CHAPTER 3. USING SYSTEMD WITH CONTAINERS</b> .....	<b>24</b>
3.1. STARTING CONTAINERS WITH SYSTEMD	24
3.2. STARTING SERVICES WITHIN A CONTAINER USING SYSTEMD	25
<b>CHAPTER 4. RUNNING SUPER-PRIVILEGED CONTAINERS</b> .....	<b>28</b>
4.1. OVERVIEW	28
4.2. RUNNING PRIVILEGED CONTAINERS	29
4.2.1. Understanding Name Spaces in Privileged Containers	30
4.3. USING THE ATOMIC TOOLS CONTAINER IMAGE	31
4.3.1. Overview	31
4.3.2. Getting and Running the RHEL Tools Container	32
4.3.3. Running Commands from the RHEL Tools Container	32
4.3.4. More Information About Running RHEL Tools Container	32

4.4. USING THE ATOMIC SUPPORT TOOLS CONTAINER IMAGE	33
4.4.1. How It Is Different from RHEL Atomic Tools	33
4.4.2. Getting and Running the Support Tools Container	33
4.4.3. Running Commands from the Support Tools Container	34
4.5. USING THE ATOMIC RSYSLOG CONTAINER IMAGE	34
4.5.1. Overview	35
4.5.2. Getting and Running the RHEL rsyslog Container	35
4.5.3. Tips for Running rsyslog Container	36
4.6. USING THE ATOMIC SYSTEM ACTIVITY DATA COLLECTOR (SADC) CONTAINER IMAGE	38
4.6.1. Overview	38
4.6.2. Getting and Running the RHEL sadc Container	38
4.6.3. Tips for Running the sadc Container	40
4.7. USING THE ATOMIC NET-SNMP CONTAINER IMAGE	40
4.7.1. Installing and Running the Net-SNMP Container	40
4.7.2. Running Commands in the Net-SNMP Container	41
4.7.3. Configuring the SNMP Agent	41
4.7.4. Monitoring an Atomic Host System Using Net-SNMP	41
4.7.5. Extending Net-SNMP to Provide Application Metrics	41
4.8. USING THE ATOMIC SSSD CONTAINER IMAGE	42
4.8.1. Overview	42
4.9. USING THE ATOMIC RHEVM-GUEST-AGENT CONTAINER IMAGE	42
4.9.1. Overview	42
4.9.1.1. Overview of the rhvm-guest-agent Container	42
4.9.2. Getting and Running the RHEL rhvm-guest-agent Container	43
4.9.3. Tips for Running the rhvm-guest-agent Container	44
4.10. USING THE ATOMIC RHEL7 INIT CONTAINER IMAGE	44
4.10.1. Overview	44
4.10.2. Getting the Atomic RHEL7 Init Container Image	45
4.10.3. Creating Container Images based on the Atomic RHEL7 Init Container Image	45
4.11. USING THE ATOMIC RHEL6 INIT CONTAINER IMAGE	46
4.11.1. Overview	46
4.11.2. Getting the Atomic RHEL6 Init Container Image	46
4.11.3. Creating Container Images based on the Atomic RHEL6 Init Container Image	46
<b>CHAPTER 5. FINDING, RUNNING, AND BUILDING CONTAINERS WITHOUT DOCKER</b> .....	<b>48</b>
5.1. OVERVIEW	48
5.2. USING PODMAN TO WORK WITH CONTAINERS	48
5.2.1. Installing podman	49
5.2.2. Running containers with podman	50
5.2.3. Trying basic podman commands	51
5.2.3.1. Pull a container image to the local system	51
5.2.3.2. List local container images	52
5.2.3.3. Run a container image	52
5.2.3.4. List containers that are running or have exited	52
5.2.3.5. Remove a container or image	52
5.2.3.6. Remove a container image by its image ID or name (use -f to force):	52
5.2.3.7. Build a container	53
5.3. RUNNING CONTAINERS WITH RUNC	53
5.3.1. Installing and running containers	53
5.4. USING SKOPEO TO WORK WITH CONTAINER REGISTRIES	54
5.4.1. Inspecting container images with skopeo	55
5.4.2. Copying container images with skopeo	56
5.4.3. Getting image layers with skopeo	56

---

5.5. BUILDING CONTAINER IMAGES WITH BUILDDAH	57
5.5.1. Understanding Buildah	57
5.5.2. Installing Buildah	58
5.5.3. Getting Images with buildah	58
5.5.4. Building an Image from a Dockerfile with Buildah	59
5.5.5. Running a Container with Buildah	60
5.5.6. Inspecting a Container with buildah	60
5.5.7. Modifying a Container to Create a new Image with Buildah	61
5.5.7.1. Using buildah mount to Modify a Container	61
5.5.7.2. Using buildah copy and buildah config to Modify a Container	62
5.5.8. Creating images from scratch with Buildah	63
5.5.9. Removing Images or Containers with Buildah	64
5.5.10. Using container registries with Buildah	65
5.5.10.1. Pushing containers to a private registry	65
5.5.10.2. Pushing containers to the Docker Hub	66
<b>CHAPTER 6. RUNNING SYSTEM CONTAINERS</b> .....	<b>68</b>
6.1. USING THE ETCD SYSTEM CONTAINER IMAGE	69
6.1.1. Overview	69
6.1.2. Getting and Running the etcd System Container	70
6.1.3. Configuring etcd	71
6.1.3.1. Configuring etcd during "atomic install"	71
6.1.3.2. Configuring etcd security settings	72
6.1.3.3. Configuring etcd with "runc"	73
6.1.4. Tips for Running etcd Container	73
6.2. USING THE FLANNEL SYSTEM CONTAINER IMAGE	73
6.2.1. Overview	73
6.2.2. Getting and Running the RHEL flannel System Container	74
6.2.3. Configuring flannel	75
6.2.3.1. Configuring etcd during "atomic install"	75
6.2.3.2. Configuring flannel with "runc"	76
6.2.4. Tips for Running flannel Container	76
6.3. USING THE OVRT-GUEST-AGENT SYSTEM CONTAINER IMAGE FOR RED HAT VIRTUALIZATION	76
6.3.1. Overview	76
6.3.2. Getting and Running the ovirt-guest-agent System Container	77
6.3.3. Removing the ovirt-guest-agent Container and Image	78
6.4. USING THE OPEN-VM-TOOLS SYSTEM CONTAINER IMAGE FOR VMWARE	78
6.4.1. Overview	78
6.4.2. Getting and Running the open-vm-tools System Container	79
6.4.3. Removing the open-vm-tools Container and Image	80





# CHAPTER 1. MANAGING STORAGE WITH DOCKER-FORMATTED CONTAINERS

## 1.1. OVERVIEW

Running a large number of containers in production requires a lot of storage space. Additionally, creating and running containers requires the underlying storage drivers to be configured to use the most performant options. The default storage options for Docker-formatted containers vary between the different systems and in some cases they need to be changed. A default installation of RHEL uses loopback devices, whereas RHEL Atomic Host has LVM thin pools created during installation. Note that using the loopback option is not recommended for production systems.

During the planning phase, ensure that:

- 1) You are running in the direct-lvm mode and have LVM thin pools set up. This can be done using the **docker-storage-setup** utility.
- 2) You allocate enough free space during installation or plan for an external storage to be attached to the system.

This document also includes procedures on how to extend the storage when you run out of space. Some of these procedures are destructive, this is why it is recommended to plan in advance. Use the described procedures relevant to your system to help you set up the environment.



### NOTE

Prior to RHEL 7.4, the *container-storage-setup* utility was called *docker-storage-setup*. As with previous RHEL Server and RHEL Atomic releases, you can add storage settings to */etc/sysconfig/docker-storage-setup* as input, which results in those settings being placed in */etc/sysconfig/docker-storage* as output.

## 1.2. USING CONTAINER-STORAGE-SETUP

The *container-storage-setup* utility is installed with the *container-storage-setup* package, while leveraging components from the *docker* package. This utility can assist you in setting up the direct LVM storage.

When docker starts, it automatically starts the *docker-storage-setup* daemon. By default, *docker-storage-setup* tries to find free space in the Volume Group containing the root Logical Volume and tries to set up an LVM thin pool. If there is no free space in the Volume Group, *docker-storage-setup* will fail to set up an LVM thin pool and will fall back to using loopback devices.

The default behavior of *docker-storage-setup* is controlled by the */etc/sysconfig/docker-storage* configuration file. You can override these options by creating a file */etc/sysconfig/docker-storage-setup* using new values.

The *container-storage-setup* utility needs to know where the free space is to set up a thin pool. The following sections describe some of the ways you can configure the system to make sure *container-storage-setup* can setup an LVM thin pool.

See *man 1 container-storage-setup* for more information. (Note that manual pages are not available on RHEL Atomic, however they are available on RHEL Server systems.)

### 1.2.1. LVM thin pool in the volume group containing the root volume

By default, `container-storage-setup` looks for free space in the root volume group and creates an LVM thin pool. Hence you can leave free space during system installation in the root volume group and starting `docker` will automatically set up a thin pool and use it.

### 1.2.2. LVM thin pool in a user specified volume group

`container-storage-setup` can be configured to use a specific volume group for creating a thin pool.

```
# echo VG=docker-vg >> /etc/sysconfig/docker-storage-setup
# systemctl start docker
```

### 1.2.3. Setting up a volume group and LVM thin pool on user specified block device

You can specify one or multiple block devices in the `/etc/sysconfig/docker-storage-setup` file and `container-storage-setup` will create a volume group and an LVM thin pool for the `docker` service to use.

```
# echo DEVS=/dev/vdb >> /etc/sysconfig/docker-storage-setup
# systemctl start docker
```

## 1.3. MANAGING STORAGE IN RED HAT ENTERPRISE LINUX

In Red Hat Enterprise Linux, there is no free space in the root volume group by default. Therefore, some action to ensure `container-storage-setup` can find free space is required.

An easy way is to leave some free space in the volume group containing root during installation. The following section explains how to leave free space.

### 1.3.1. How to Leave Space in the Volume Group Backing Root During Installation

There are two methods to leave free space in the root volume group during installation. Using the interactive graphical installation utility `Anaconda` or by preparing a Kickstart file to control the installation.

#### 1.3.1.1. GUI Installation

1. Start the graphical installation; when you arrive at the "Installation Destination" screen, select "I will configure partitioning" from "Other Storage Options" and click "Done".
2. On the "Manual Partitioning" screen, where you are prompted to create mount points. Choose "Click here to create them automatically". This will create the default partitioning scheme.
3. Choose the root partition (`/`), this displays the "Desired Capacity" input field.
4. Reduce that capacity to leave some free space in the root volume group.
5. By default, the volume group which has the root LV is big enough to accommodate user-created volumes. Any free space on disk is left free and is not part of that volume group. Change that by clicking on "Modify", selecting "Size policy" and setting that to "As large as possible". Click "Save". This makes sure that any unused space on disk is left free in the volume group.
6. Click "Done" to accept the proposed partitioning.
7. Click "Begin Installation".

### 1.3.1.2. Kickstart Installation

In a Kickstart file, use the "volgroup" Kickstart option with the "--reserved-percent" and "--reserved-space" options where you can specify how much space to leave free in the volume group. Here is an example section of a Kickstart file which leaves 20% free space in the root LV:

```
# Disk partitioning information
part /boot --size=500
part pv.1 --size=500 --grow
volgroup rhel --pesize=4096 pv.1 --reserved-percent=20
logvol / --size=500 --grow --name=root --vgname=rhel
logvol swap --size=2048 --name=swap --vgname=rhel
```

## 1.4. MANAGING STORAGE IN RED HAT ENTERPRISE LINUX ATOMIC HOST

On RHEL Atomic Host, the root volume size is 3GB. There is free space in the root volume group and 60% of that is used by *container-storage-setup* for setting up an LVM thin pool. The rest of the space is free and can be used for extending the root volume or for creating a thin pool.

On RHEL Atomic Host with default partitioning setup, the *docker-storage-setup* service creates an LVM thin pool to be used by the container images. During installation, the installation program creates the **root** Logical Volume that is 3GB by default. Next, during boot, the *docker-storage-setup* service automatically sets up an LVM thin pool called **docker-pool** which takes 60% of the remaining space. The rest can be used for extending **root** or **docker-pool**. During boot, *docker-storage-setup* reads the */etc/sysconfig/docker-storage* file to determine the type of storage used and it modifies it so that docker makes use of the LVM thin pool. You can override the defaults by creating a file called */etc/sysconfig/docker-storage-setup* which will modify the behavior of the service during boot. If you do not create such file, then an LVM thin pool will be created by default.

Red Hat Enterprise Linux Atomic Host installed from a cloud image with default partitioning has a Volume Group called **atomicos** and two Logical Volumes as part of that group. The name of the Volume Group varies between different images of Red Hat Enterprise Linux Atomic Host. For bare-metal and virtual installations the Volume Group name is derived from the host name. If the host is unnamed, the Volume Group will be called **rah**. The properties of the Volume Group and the Logical Volumes in them are the same across all images.

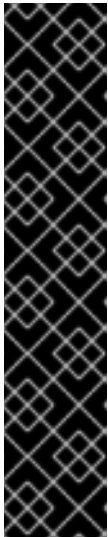
### 1.4.1. Contents of the Atomic Host filesystem

You can run the **lvs** command to list the Logical Volumes on the system and see the Volume Group name:

```
# lvs
LV          VG          Attr          LSize Pool Origin Data%
Meta%  Move Log Cpy%Sync Convert
docker-pool  atomicos    twi-aotz--    7.69g          14.36  2.56
root        atomicos    -wi-ao----    2.94g
```

1. The **Root partition** is called root and is 3GB by default. root is a Logical Volume that contains the following:
  - The */var* and */etc* directories.
  - The */ostree/repo* which contains the OSTree versions.

- The `/var/lib/docker/` directory which contains container images data, such as temporary data or the **docker volumes**. A **docker volume** is a unit of storage that a running container can request from the host system. The unit of storage can be provided by another container but also by the host directly. In the case of Red Hat Enterprise Linux Atomic Host, these volumes are automatically allocated to the **Root Partition**, in `/var/lib/docker/vfs/`.
2. A **Container Image Partition** called **docker-pool** which takes 60% of the remaining space. It is formatted as an LVM thin pool by the `docker-storage-setup` service. It is used to store the container images. The space used by **docker-pool** is managed by the `docker-storage-setup` service. When you pull a container image from a registry, for example, the image takes up space on this partition. Container images are read-only. Once an image is launched as a container, all writes (except to mounted volumes or docker volumes) are stored in this Logical Volume.



### IMPORTANT

It is important to monitor the free space in `docker-pool` and not to allow it to run out of space. If the LVM thin pool runs out of space it will lead to a failure because the XFS file system underlying the LVM thin pool will be retrying indefinitely in response to any I/O errors. The LVM2 tools provide a facility to monitor a thin pool and extend it based on user settings. See the *Automatically extend thin pool LV and Data space exhaustion* sections of the `lvmthin(7)` manual page for more information. By default, `docker-storage-setup` configures the thin pool for auto extension. This means as the pool fills up, it will automatically grow and consume free space available in that volume group. If the volume group gets full and there is no space left for auto extension, then you can preemptively destroy old containers that are no longer needed in order to reclaim space. Or you can stop creating or modifying container images until additional storage is added to the system.

The `docker-pool` partition contains the following directories:

- `/etc/sysconfig/docker` - configured by the user
- `/etc/sysconfig/docker-storage` - configured by programs, but can be edited by the user (you have to disable `docker-storage-setup`)
- `/etc/sysconfig/docker-storage-setup` - configured by the user; only available in RHEL Atomic Host

## 1.4.2. Changing the Default Size of the Root Partition During Installation

To change the default **Root Partition** size, use the method below for your installation.

- **Anaconda:** When you arrive at the "Installation Destination" screen, select "I will configure partitioning" from "Other Storage Options" and click "Done". This will lead you to the "Manual Partitioning" screen, where you are prompted to create mount points. Choose "Click here to create them automatically", which will give you the boot, root, and swap partitions. (At this point, you only have these partitions, **docker-pool** is created later by the `docker-storage-setup` service). Choose the root partition (`/`) and enter the new value in the "Desired Capacity" input field. When you finish the installation, the system boots with your custom configuration.
- **Kickstart:** In the `%post` section of the Kickstart file, give the path to the `/etc/sysconfig/docker-storage-setup` file (which will be created automatically) and specify the necessary options after the command. The syntax is as follows:

```
%post
cat > /etc/sysconfig/docker-storage-setup << EOF
ROOT_SIZE=6G
EOF
%end
```

- **cloud-init:** The `write_files` directive in the user-data file is used to setup the `/etc/sysconfig/docker-storage-setup` file similarly to the Kickstart example above. This example user-data file sets the password for **cloud-user** to "atomic" and configures the root partition to be 6GB instead of the default 3GB.

```
#cloud-config
password: atomic
write_files:
- path: /etc/sysconfig/docker-storage-setup
  permissions: 0644
  owner: root
  content: |
    ROOT_SIZE=6G
```

### 1.4.3. Changing the Size of the Root Partition After Installation

When you add container images to the **Container Image Partition** which require space in `/var/lib/docker/`, the image can request more space than is currently available on the **Root Partition**. A container image can request a docker volume when it has data that should not be stored in the container, for example the data from a database server. If you run out of space on **root**, you have three options:

- Extend the Root Partition to use the free space in the volume group.
- Add new storage to the host and extend the Root Partition.
- Extend the Root Partition and shrink the Container Image Partition.

#### 1.4.3.1. How to extend the Root Partition to use free space in volume group

If there is free space in volume group, then you can extend the root volume to use some or all of that free space and grow the root partition.

```
# lvextend -r -L +3GB /dev/atomicos/root
```

#### 1.4.3.2. How to Add Additional Storage to the Host and Extend the Root Partition

This option is non-destructive and will enable you to add more storage to the **Root Partition** and use it. This requires creating a new Physical Volume using a new disk device (in this example `/dev/sdb`), add it to **atomicos** Volume Group and then extend the **Root Partition** Logical Volume. You must stop the docker daemon and the `docker-storage-setup` service for this task. Use the following commands:

```
# systemctl stop docker docker-storage-setup
# pvcreate /dev/sdb
# vgextend atomicos /dev/sdb
# lvextend -r -L +3GB /dev/atomicos/root
# systemctl start docker docker-storage-setup
```

### 1.4.3.3. How to Extend the Root Partition Without Adding More Storage

This option is destructive because the **Container Image Partition** will be destroyed. When it is not possible to add more storage to the **Root Partition**, you can extend it. Extending the **Root Partition** means that you will have to shrink the **Container Image Partition**. However, since LVM does not support shrinking Thinly-Provisioned Logical Volumes,

Therefore, you must stop all running containers, destroy the **Container Image Partition**, and extend the **Root Partition**. *docker-storage-setup* will reallocate the remaining space to the **Container Image Partition** when it is restarted. Use the following commands:

```
# systemctl stop docker docker-storage-setup
# rm -rf /var/lib/docker/*
# lvremove atomicos/docker-pool
# lvextend -L +3GB /dev/atomicos/root
# systemctl start docker-storage-setup
# systemctl start docker
```

At this point you will need to download all container images again.

## 1.5. CHANGING STORAGE CONFIGURATION

If you change the storage configuration for Docker-formatted containers, you must also remember to remove the */var/lib/docker* directory. This directory contains the metadata for old images, containers, and volumes which are not valid for the new configuration. Examples of instances in which changing the storage configuration might be required include when switching from using loop devices to LVM thin pool, or switching from one thin pool to another. In the latter case, the old thin pool should be removed.

```
# systemctl stop docker docker-storage-setup
# rm /etc/sysconfig/docker-storage-setup
# lvremove docker/docker-pool
# rm -rf /var/lib/docker/
# systemctl start docker
```

## 1.6. USING THE OVERLAY GRAPH DRIVER

The **overlay** graph driver uses OverlayFS, a copy-on-write union file system that features page-cache sharing between snapshot volumes. Similarly to LVM thin pool, OverlayFS supports efficient storage of image layers. However, compared to LVM thin pool, container creation and destruction with OverlayFS uses less memory and is more performant.



### WARNING

OverlayFS is not POSIX-compliant, because some of the file system semantics are different from standard file systems like **ext4** and **XFS**. Hence, make sure your applications work with OverlayFS before changing the storage driver to **overlay**.

In Atomic Host 7.5 and later, the **overlay** driver is the default storage driver. If you have a system installed before Atomic Host 7.5, upgrading it to 7.5 will not change the configured storage driver to **overlay**. You can switch to **overlay** manually by executing these steps:

## Enabling Overlay

Execute the following steps to enable the **overlay** Graph Driver for the docker service:



### IMPORTANT

Changing the storage backend is a destructive operation. Before starting, be sure to back up your images. This can be done in two ways:

1. Use **docker save** to back up your images and then **docker load** to restore them.
2. Use **atomic storage export** to save all data and **atomic storage import** to restore it into the new storage backend.

1. Stop **docker** and remove the current storage:

```
#systemctl stop docker
# container-storage-setup --reset
# rm -rf /var/lib/docker/
```

2. In the `/etc/sysconfig/docker-storage-setup` file, specify the following directives:

- **STORAGE\_DRIVER** - **overlay2** or **overlay**



### NOTE

Red Hat recommends to use **overlay2** and not **overlay**, because **overlay2** is more performant.

- **DEVS** - block storage device
- **CONTAINER\_ROOT\_LV\_NAME** - the LVM logical volume name
- **CONTAINER\_ROOT\_LV\_SIZE** - the absolute size or percentage of the volume group to use for the logical volume
- **CONTAINER\_ROOT\_LV\_MOUNT\_PATH** - the root location for the container storage
- **VG** - the LVM volume group

For example:

```
STORAGE_DRIVER=overlay2
DEVS=/dev/sdb
CONTAINER_ROOT_LV_NAME=dockerlv
CONTAINER_ROOT_LV_SIZE=100%FREE
CONTAINER_ROOT_LV_MOUNT_PATH=/var/lib/docker
VG=dockervg
```

If the atomic rootfs and the overlay volume use different volume groups, it is often optimal to let overlay use all available space on its volume group. To do this, set **CONTAINER\_ROOT\_LV\_SIZE** to **100%FREE**. If not specified, the volume defaults to using 40% of space on the volume group.

Variables **CONTAINER\_ROOT\_LV\_NAME** and **CONTAINER\_ROOT\_LV\_MOUNT\_PATH** behave this way:

- If you specify these variables, a new logical volume for storing container images and container rootfs is created.
  - If you do not specify them, then no new logical volume will be created, and overlay2 will be setup on rootfs.
3. If in the previous step you set the root location for the container storage (**CONTAINER\_ROOT\_LV\_MOUNT\_PATH**) to **/var/lib/docker**, which is the standard docker storage directory, then the SELinux context on that directory and its contents are properly set to **container\_var\_lib\_t**.

However, if you use a different directory, you must relabel it. Otherwise SELinux will not allow its use. For example, to use **/var/lib/mystorage**, you would type:

```
# mkdir -p /var/lib/mystorage
# chmod 600 /var/lib/mystorage
# semanage fcontext -a -t container_var_lib_t
'/var/lib/mystorage(/.*)?'
# restorecon -vvFR /var/lib/mystorage/
# ls -ldZ /var/lib/mystorage/
drwx----- . root root system_u:object_r:container_var_lib_t:s0
/var/lib/mystorage/
```

4. Restart docker:

```
# systemctl start docker
```

## Enabling Overlay in Kickstart

For a Kickstart installation you need to put the Overlay configuration in the **%post** section. For example:

```
%post
echo "STORAGE_DRIVER=overlay2
DEVS=/dev/sdb
CONTAINER_ROOT_LV_NAME=dockerlv
CONTAINER_ROOT_LV_SIZE=100%FREE
CONTAINER_ROOT_LV_MOUNT_PATH=/var/lib/docker
VG=dockervg" >> /etc/sysconfig/docker-storage-setup
%end
```

For explanation and guidance on the exact values of the directives, see the [Enabling Overlay](#) procedure.

## Enabling Overlay in cloud-init

For a cloud-init installation you need to include the Overlay configuration as a snippet in the *user-data* file. For example:

```
runcmd:
- echo "STORAGE_DRIVER=overlay2" >> /etc/sysconfig/docker-storage-setup
```



```

- echo "DEVS=/dev/sdb" >> /etc/sysconfig/docker-storage-setup
- echo "CONTAINER_ROOT_LV_NAME=dockerlv" >> /etc/sysconfig/docker-
storage-setup
- echo "CONTAINER_ROOT_LV_SIZE=100%FREE" >> /etc/sysconfig/docker-
storage-setup
- echo "CONTAINER_ROOT_LV_MOUNT_PATH=/var/lib/docker" >>
/etc/sysconfig/docker-storage-setup
- echo "VG=dockervg" >> /etc/sysconfig/docker-storage-setup

```

For explanation and guidance on the exact values of the directives, see the [Enabling Overlay](#) procedure.

## 1.7. INCREASING THE BASE DEVICE SIZE

The "base device size" is the maximum size an image or container can grow to. You can check the default base size for your version of docker by running **docker info**:

```

# docker info
Containers: 0
Images: 0
Server Version: 1.9.1
Storage Driver: devicemapper
Pool Name: docker-253:1-1313713-pool
Pool Blocksize: 65.54 kB
Base Device Size: 107.4 GB

```

The base device size has been changed since docker 1.9 from 100GB to 10GB. The following is a list of the default sizes for the different versions of docker:

- docker 1.9 Base Device Size: 107.4 GB
- docker 1.10 Base Device Size: 10.74 GB
- docker 1.11 Base Device Size: 10.74 GB

This default limit is defined by docker and will apply to all future images and containers. You can increase this per container limit using the **--dm.basesize** option, and the docker-storage service will update it on next reboot.

```
# docker daemon --storage-opt --dm.basesize=20GB
```

Limitations:

- This option only applies to the *devicemapper* storage backend.
- You can only expand the base size, but you cannot set a limit smaller than the default for your version of docker.
- All new containers would not have the increased roots size. Even after restarting the daemon with the new base device size using **--storage-opt dm.basesize=20G**, you still need to update all the existing images in order for new containers to reap benefits of this new size.
- With this approach, the heaviest application (container) dictates the size for the rest of the containers, for example, if you want to have 100 containers on your infrastructure and one of them is a data intensive application requiring 100 GB of space, you would have to set the base

device size to 100 GB. Even though there are 99 other containers that only need 200 MB of space each.

## 1.8. RESETTING STORAGE FOR CONTAINERS

Since this is a destructive command, and requires some preparations, following is a procedure explaining in detail how to use the command:

1. Make sure that you have a version of Atomic Host that is 7.2.5 or later:

```
# atomic host upgrade
```

2. Confirm that you have a version of Atomic Host that is 7.2.5 or later by checking the Version field when you run **atomic host status**:

```
# atomic host status
State: idle
Deployments:
* rhel-atomic-host-ostree:rhel-atomic-host/7/x86_64/standard
  Version: 7.2.6 (2016-07-29 19:54:25)
  Commit:
b672bf8a457cb28e003dee20c53749636ef5fce3e4743afe4aaad269d3aaa62a
  OSName: rhel-atomic-host

rhel-atomic-host-ostree:rhel-atomic-host/7/x86_64/standard
  Version: 7.2.5 (2016-06-18 15:21:12)
  Commit:
9bfe1fb65094d43e420490196de0e9aea26b3923f1c18ead557460b83356f058
  OSName: rhel-atomic-host
```

3. List the current contents of the storage to make sure everything is safe to delete.

```
# atomic images

REPOSITORY                                TAG      IMAGE ID      CREATED
VIRTUAL SIZE
registry.access.redhat.com/rhel6          latest   sha256:b335a
2016-07-07 13:31    195.66 MB
registry.access.redhat.com/rhel7/rhel-tools latest   sha256:38819
2016-06-22 06:54    1.3 GB
registry.access.redhat.com/rhel7/openscap latest   sha256:da0d5
2016-06-20 14:24    363.37 MB
registry.access.redhat.com/rhel7/rsyslog  latest   sha256:878a5
2016-06-16 17:18    216.0 MB
registry.access.redhat.com/rhel7          latest   sha256:5fbb7
2016-06-16 13:27    203.5 MB
```

4. Stop the docker daemon:

```
# systemctl stop docker
```

5. Run the **atomic storage reset** command:

```
# atomic storage reset
```

6. Start the docker daemon again:

```
# systemctl start docker
```

7. Run the **atomic images list** command to show that all images and containers have been removed and that storage on the Atomic Host has been reset:

```
# atomic images
```

```
REPOSITORY TAG IMAGE ID          CREATED          VIRTUAL SIZE
```

## 1.9. STORAGE BACKUP GUIDELINES

Red Hat currently does not endorse any single backup technology for containers. We do, however, suggest the following general guidelines:

- Ensure that the Dockerfiles that you use to create containers are backed up.
- Ensure that any data required by or generated by your containers is housed on an external source. Back up the external source in a responsible manner and on a reasonable schedule.
- Create a local docker repository and use it for saving and retrieving custom containers.
- Use **docker save/load** or **atomic storage export/import** to create portable images of containers and back up those images.

## 1.10. ADDITIONAL INFORMATION ABOUT STORAGE

- The [Thinly-Provisioned Logical Volumes](#) section from the LVM Administrator Guide explains LVM Thin Provisioning in detail.
- The [Red Hat Enterprise Linux 7 Storage Administration Guide](#) provides information on adding storage to Red Hat Enterprise Linux 7.

## CHAPTER 2. SIGNING CONTAINER IMAGES

Signing container images on RHEL systems provides a means of validating where a container image came from, checking that the image has not been tampered with, and setting policies to determine which validated images you will allow to be pulled to your systems. Before you begin, there are a few things you should know about Red Hat container image signing:

- **Docker version:** The features described here require at least Docker 1.12.3. So you can use the `docker` package for any RHEL and RHEL Atomic release after 7.3.2.
- **Red Hat Signed Images:** As of RHEL and RHEL Atomic 7.4, image signing is fully supported (no longer tech preview). With RHEL 7.4, Red Hat has also begun signing its own container images. So you can use the instructions provided in this chapter to determine the authenticity of those images using Red Hat GPG keys.

This chapter describes tools and procedures you can use on Red Hat systems to not only sign images, but also consume images that have been signed in these ways:

- **Creating Image Signatures:** By signing images with a private key and sharing a public key generated from it, others can use the public key to authenticate the images you share. The signatures needed to validate the images can be made available from an accessible location (like a Web server) in what is referred to as a "signature store" or made available from a directory on the local filesystem. The actual signature can be created from an image stored in a registry or at the time the image is pushed to a container registry.
- **Verifying Signed Images:** You can check a signed image when it is pulled from a registry. This includes verifying Red Hat's signed images.
- **Trusting Images:** Besides determining that an image is valid, you can also set policies that say which valid images you trust to use on your system, as well as which registries you trust to use without validation.

For the current release Red Hat Enterprise Linux and RHEL Atomic Host, there are a limited number of tools and container registries that support image signing. Over time, however, you can expect most features on RHEL systems that pull or store images to support signing. To get you started in the mean time, however, you can use the following features in RHEL:

- **Registries:** Currently, you can use a local container registry (`docker-distribution` package) and the Docker Hub (`docker.io`) from RHEL systems to push and pull signed images. For the time being, image signing features are only supported in v2 (not v1) Docker Registries.
- **Image signing tools:** To create image signatures, you can use **`atomic sign`** (to create a signature from a stored image) or **`atomic push`** (to create an image signature as you push it to a registry).
- **Image verifying tools:** To verify a signed image, you can use the **`atomic trust`** command to identify which image registries you trust without verification and which registries require verification. Later, when you pull an image, the **`atomic pull`** or **`docker pull`** command will validate the image as it is pulled to the local system.
- **Operating systems:** The image signing and verification features described here are supported in Red Hat Enterprise Linux Server and RHEL Atomic Host systems, version 7.3.1 and later.

For a more complete description of Red Hat container image signing, see:

[Container Image Signing Integration Guide](#)

## 2.1. GETTING CONTAINER SIGNING SOFTWARE

If you are using a RHEL Atomic Host 7.3.2 system (or later), the tools you need to sign, trust and verify images are already included.

Most container-related software for RHEL server is in the `rhel-7-server-extras-rpms` yum repository. So, on your RHEL 7.3 server system, you should enable that repository, install packages, and start the docker service as follows:

```
# subscription-manager repos --enable=rhel-7-server-extras-rpms
# yum install skopeo docker atomic
# systemctl enable docker; systemctl start docker
```

The docker service should now be running and ready for you to use.

## 2.2. CREATING IMAGE SIGNATURES

Image signing in this section is broken down into the following steps:

- **GPG keys:** Create GPG keys for signing images.
- **Sign images:** Choose from either creating a signature from an image already in a container registry or creating a signature as you push it to a container registry.

### 2.2.1. Create GPG Keys

To sign container images on Red Hat systems, you need to have a private GPG key and a public key you create from it. If you don't already have GPG keys you want to use, you can create them with the `gpg2 --gen-key` command. This procedure was done from a terminal window on a GNOME desktop as the user who will sign the images:

1. **Create private key:** Use the following command to interactively add information needed to create the private key. You can use defaults for most prompts, although you should properly identify your user name, email address, and add a comment. You also must add a passphrase when prompted.

```
$ gpg2 --gen-key
Please select what kind of key you want:
Your selection? 1
What keysize do you want? (2048) 2048
Please specify how long the key should be valid.
    0 = key does not expire
Key is valid for? (0) 0
Key does not expire at all
Is this correct? (y/N) y
Real name: Joe Smith
Email address: jjsmith@example.com
Comment: Image Signing Key
You selected this USER-ID:
    "Joe Smith (Image Signing Key) <jjsmith@example.com>"
Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit? 0
You need a Passphrase to protect your secret key.
gpg: /home/jsmith/.gnupg/trustdb.gpg: trustdb created
gpg: key D3F46FFF marked as ultimately trusted
public and secret key created and signed.
```

You will need the passphrase later, when you attempt to sign an image. Anyone with access to your private key and passphrase will be able to identify content as belonging to you.

2. **Create entropy:** You need to generate activity on your system to help produce random data that will be used to produce your key. This can be done by moving windows around or opening another Terminal window and running a variety of commands, such as ones that write to disk or read lots of data. Once enough entropy is generated, the `gpg2` command will exit and show information about the key just created.

3. **Create public key:** Here's an example of creating a public key from that private key:

```
$ gpg2 --armor --export --output mysignkey.gpg jjsmith@example.com
```

4. **List keys:** Use the following command to list your keys.

```
$ gpg2 --list-keys
/home/jsmith/.gnupg/pubring.gpg
-----
pub    2048R/D3F46FFF 2016-10-20
uid                               Joe Smith (Image Signing Key)
<jjsmith@example.com>
sub    2048R/775A4344 2016-10-20
```

At this point, the private key you created is available to use from this user account to sign images and the public key (**mysignkey.gpg**) can be shared with others for them to use to validate your images. Keep the private key secure. If someone else has that key, they could use it to sign files as though those files belonged to you.

### 2.2.2. Creating an Image Signature

With your private key in place, you can begin preparing to sign your images. The steps below show two different ways to sign images:

- **From a repository:** You can create a signature for an image that is already in a repository using **atomic sign**.
- **Image at push time:** You can tag a local image and create an image signature as you push it to a registry using **atomic push**.

Image signing requires super user privileges to run the **atomic** and **docker** commands. However, when you sign, you probably want to use your own keys. To take that into account, when you run the **atomic** command with **sudo**, it will read keys from your regular user account's home directory (not the root user's directory) to do the signing.

## 2.3. SET UP TO DO IMAGE SIGNING

If you are going to sign a lot of images on a personal system, you can identify signing information in your `/etc/atomic.conf` file. Once you add that information to **atomic.conf**, the **atomic** command assumes that you want to use that information to sign any image you push or sign. For example, for a user account `jjsmith` with a default signer of `jjsmith@example.com`, you could add the following lines to the `/etc/atomic.conf` file so that all images you push or sign would be signed with that information by default:

```
default_signer: jjsmith@example.com
gnupg_homedir: /home/jjsmith/.gnupg
```

If you want to use a different signer or signing home directory, to override those default values, you can do that later on the **atomic** command line using the **--sign-by** and **--gnupghome** options, respectively. For example, to have `jjsmith@example.com` and `/home/jjsmith/.gnupg` used as the signer and default gnupg directory, type the following on the **atomic** command line:

```
$ sudo atomic push --sign-by jjsmith@example.com \
    --gnupghome /home/jjsmith/.gnupg \
    docker.io/wherever/whatever
```

## 2.4. CREATING A SIGNATURE FOR AN IMAGE IN A REPOSITORY

You can create an image signature for an image that is already pushed to a registry using the **atomic sign** command. Use **docker search** to find the image, then **atomic sign** to create a signature for that image.

**IMPORTANT:** The image signature is created using the exact name you enter on the **atomic sign** command line. When someone verifies that image against the signature later, they must use the exact same name or the image will not be verified.

1. **Find image:** Find the image for which you want to create the signature using the **docker search** command:

```
$ sudo docker search docker.io/jjsmith/mybusybox
INDEX          NAME                                DESCRIPTION  STARS
OFFICIAL      AUTOMATED
docker.io     docker.io/jjsmith/mybusybox        0....
```

2. **Create the image signature:** Choose the image you want to sign (`jjsmith/mybusybox` in this example). To sign it with the default signer and home directory entered in `/etc/atomic.conf`, type the following:

```
$ sudo atomic sign docker.io/jjsmith/mybusybox
Created:
/var/lib/atomic/sigstore/docker.io/jjsmith/mybusybox@sha256:9393222c
6789842b16bcf7306b6eb4b486d81a48d3b8b8f206589b5d1d5a6101/signature-1
```

When you are prompted for a passphrase, enter the passphrase you entered when you created your private key. As noted in the output, you can see that the signature was created and stored in the `/var/lib/atomic/sigstore` directory on your local system under the registry name, user name, and image name (`docker.io/jjsmith/mybusybox*sha256:...`).

## 2.5. CREATING AN IMAGE SIGNATURE AT PUSH TIME

To create an image signature for an image at the time you push it, you can tag it with the identity of the registry and possibly the username you want to be associated with the image. This shows an example of creating an image signature at the point that you push it to the Docker Hub (`docker.io`). In this case, the procedure relies on the default signer and GPG home directory assigned earlier in the `/etc/atomic.conf` file.

1. **Tag image:** Using the image ID of the image, tag it with the identity of the registry to which you want to push it.

```
$ sudo docker tag hangman docker.io/jjsmith/hangman:latest
```

- 2. **Push and sign the image:** The following command creates the signature as the image is pushed to docker.io:

```
$ sudo atomic push -t docker docker.io/jjsmith/hangman:latest
Registry Username: jjsmith
Registry Password: *****
Copying blob sha256:5f70bf18...
Signing manifest
Writing manifest to image destination
Storing signatures
```

When prompted, enter the passphrase you assigned when you created your private key. At this point, the image should be available from the repository and ready to pull.

## 2.6. SHARING THE SIGNATURE STORE

The signatures you just created are stored in the `/var/lib/atomic/sigstore` directory. For the purposes of trying out signatures, you can just use that signature store from the local system. However, when you begin sharing signed images with others and have them validate those images, you would typically share that signature store directory structure from a Web server or other centrally available location. You would also need to share the public key associated with the private key you used to create the signatures.

For this procedure, you could just copy your public key to the `/etc/pki/containers` directory and use the signature store from the local `/var/lib/atomic/sigstore` directory. For example:

```
$ sudo mkdir /etc/pki/containers
$ sudo cp mysignkey.gpg /etc/pki/containers/
```

As for the location of the signature store, you can assign that location when you run an `atomic trust add` command (shown later). Or you can edit the `/etc/containers/registries.d/default.yaml` file directly and identify a value for the `sigstore` setting (such as, `sigstore: file:///var/lib/atomic/sigstore`).

## 2.7. VALIDATING AND TRUSTING SIGNED IMAGES

Using the `atomic trust` command, you can identify policies that determine which registries you trust to allow container images to be pulled to your system. To further refine the images you accept, you can set a trust value to accept all images from a registry or accept only signed images from a registry. As part of accepting signed images, you can also identify the location of the keys to use to validate the images.

The following procedure describes how to show and change trust policies related to pulling images to your system with the `atomic` command.

1. **Check trust values:** Run this command to see the current trust value for pulling container images with the `atomic` command:

```
$ sudo atomic trust show
* (default)                accept
```

When you start out, the trust default allows any request to pull an image to be accepted.

2. **Set default to reject:** Having the default be reject might be harsh if you are just trying out containers, but could be considered when you are ready to lock down which registries to allow.



So you could leave the default as `accept`, if you like. To limit pulled images to only accept images from certain registries, you can start by changing the default to `reject` as follows:

```
$ sudo atomic trust default reject
$ sudo atomic trust show
* (default)                reject
$ sudo atomic pull docker.io/centos
Pulling docker.io/library/centos:latest ...
FATA[0000] Source image rejected: Running image
docker://centos:latest is rejected by policy.
```

You can see that the default is now to reject all requests to pull images, so an attempt to pull a container image fails.

3. **Add trusted registry without signatures:** This step shows how to allow to your system to pull images from the `docker.io` registry without requiring signature checking. You could repeat this step for every registry you want to allow (including your own local registries) for which you don't want to require signatures. Type the following to allow `docker.io` images to be pulled, without signature verification:

```
$ sudo atomic trust add docker.io --type insecureAcceptAnything
$ sudo atomic pull docker.io/centos
Pulling docker.io/library/centos:latest ...
Copying blob ...
$
```

Notice that you were able to pull the `centos` image from `docker.io` after adding that trust line.

4. **Add trusted registry with signatures:** This example identifies the Red Hat Registry (`registry.access.redhat.com`) as being a registry from which the local system will be able to pull signed images:

```
sudo atomic trust add registry.access.redhat.com \
  --pubkeys /etc/pki/rpm-gpg/RPM-GPG-KEY-redhat-release \
  --sigstore
https://access.redhat.com/webassets/docker/content/sigstore \
  --type signedBy
```

5. **Pull and verify image:** At this point, you can verify the trust settings. Run the **`atomic trust show`** command, which shows that only signed images from `registry.access.redhat.com` and any image from `docker.io` will be accepted. All other images will be rejected.

```
$ sudo atomic trust show
* (default)                reject
docker.io                  accept
registry.access.redhat.com signed
security@redhat.com,security@redhat.com
```

The trust policies you just set are stored in the `/etc/containers/policy.json` file. See the [Reasonable Locked-Down System](#) example `policy.json` file for an good, working example of this file. You can add your own policy files to the `/etc/containers/registries.d` directory. You can name those files anything you like, as long as `.yaml` is at the end.

## 2.8. VALIDATING SIGNED IMAGES FROM RED HAT

Red Hat signs its container images using its own GPG keys. Using the same public keys Red Hat uses to validate RPM packages, you can validate signed Red Hat container images. The following procedure describes the process of validating signed images from Red Hat.

Refer to the following articles related to validating Red Hat container images:

- [Verifying image signing for Red Hat Container Registry](#): Describes how to use the **atomic** command to indicate that you trust images from the Red Hat Registry and check the signatures of images you pull from that registry using the docker service.
- [Image Signatures](#): Describes how to use image signatures with OpenShift commands and the OpenShift Registry.

Follow these steps to identify how to trust the Red Hat Registry and validate the signatures of images you get from that registry:

1. Add the Red Hat container registry ([registry.access.redhat.com](https://registry.access.redhat.com)) as a trusted registry. Identify the location of the signature store (`--sigstore`) that contains the signature for each signed image in the registry and the public keys (`--pubkeys`) used to validate those signatures against the images.

```
$ sudo atomic trust add \
  --pubkeys /etc/pki/rpm-gpg/RPM-GPG-KEY-redhat-release \
  --sigstore
https://access.redhat.com/webassets/docker/content/sigstore \
  registry.access.redhat.com
```

2. Pull a container from the trusted registry. If the signature is valid, the image should be pulled with the output looking similar to the following:

```
$ sudo atomic pull rhel7/etcd
Pulling registry.access.redhat.com/rhel7/etcd:latest ...
Copying blob ...
...
Writing manifest to image destination
Storing signatures
```

If you had identified the wrong signature directory or no signature was found there for the image you wanted to pull, you would see output that looks as follows:

```
FATA[0004] Source image rejected: A signature was required, but no
signature exists
```

## 2.9. UNDERSTANDING IMAGE SIGNING CONFIGURATION FILES

The image signing process illustrated in this chapter resulted in several configuration files being created or modified. Instead using the **atomic** command to create and modify those files, you could edit those files directly. Here are examples of the files created when you run commands to trust registries.

### 2.9.1. policy.json file

The **atomic trust** command modifies settings in the `/etc/containers/policy.json` file. Here is the content of that file that resulted from changing the default trust policy to reject, accepting all requests to pull images from the **registry.access.redhat.com** registry without verifying signatures, and accepting

requests to pull images from the **jjones** account in the **docker.io** registry that are signed by GPGKeys in the **/etc/pki/containers/mysignkey.gpg** file:

```

    "default": [
      {
        "type": "reject"
      }
    ],
    "transports": {
      "docker": {
        "registry.access.redhat.com": [
          {
            "type": "insecureAcceptAnything"
          }
        ],
        "docker.io/jjones": [
          {
            "keyType": "GPGKeys",
            "type": "signedBy",
            "keyPath": "/etc/pki/containers/mysignkey.gpg"
          }
        ]
      }
    }
  }
}

```

### 2.9.2. whatever.yaml

Settings added from the **atomic trust add** command line when adding trust settings for a registry are stored in a new file that is created in the **/etc/containers/registries.d/** directory. The file name includes the registry's name and ends in the **.yaml** suffix. For example, if you were adding trust settings for the user **jjones** at **docker.io** (**docker.io/jjones**), the file that stores the settings is **/etc/containers/registries.d/docker.io-jjones.yaml**. The command line could include the location of the signature store:

```

docker:
  docker.io/jjones:
    sigstore: file:///var/lib/atomic/sigstore/

```

The settings in the **docker.io-jjones.yaml** file override default setting on your system. Default signing settings are stored in the **/etc/containers/registries.d/default.yaml** file. For more information of the format of the **registries.d** files, see [Registries Configuration Directory](#).

## CHAPTER 3. USING SYSTEMD WITH CONTAINERS

The docker daemon was designed to provide a simple means of starting, stopping and managing containers. 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 Docker-formatted containers 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.

### 3.1. 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 before the docker service is up (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:

```
# docker 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:

```
# docker 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 **docker run** line):

```
[Unit]
Description=Redis container
After=docker.service

[Service]
Restart=always
ExecStart=/usr/bin/docker start -a redis_server
ExecStop=/usr/bin/docker 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 Mon 2016-08-22 16:12:30 EDT; 27min
   ago
   Main PID: 20814 (docker-current)
   Memory: 6.9M
   CGroup: /system.slice/redis-container.service
           └─20814 /usr/bin/docker-current start -a redis_server

Aug 22 16:12:30 rhel7u2-c-nfs-s.utau.local systemd[1]: Started Redis
container.
Aug 22 16:12:30 rhel7u2-c-nfs-s.utau.local systemd[1]: Starting
Redis container...
```

To learn more about configuring services with systemd, refer to the System Administrator's Guide chapter called [Managing Services with systemd](#).

## 3.2. STARTING SERVICES WITHIN A CONTAINER USING SYSTEMD

A package with the systemd initialization system is included in the official Red Hat Enterprise Linux base images. This means that applications created to be managed with systemd can be started and managed inside a container. A container running systemd will:



### NOTE

Previously, a modified version of the systemd initialization system called **systemd-container** was included in the Red Hat Enterprise Linux versions 7.2 base images. Now, the systemd package is the same across systems.

- 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 rhel7
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:

```
# docker build -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:

```
# docker 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:

```
# docker ps | grep mysysd_run
de7bb15fc4d1  mysysd  "/sbin/init"  3 minutes ago  Up 2 minutes
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 4. RUNNING SUPER-PRIVILEGED CONTAINERS

### 4.1. OVERVIEW

Containers are designed to keep their own, contained views of namespaces and have limited access to the hosts they run on. By default, containers have a process table, network interfaces, file systems, and IPC facilities that are separate from the host. Many security features like capabilities and SELinux are wrapped around containers to control access to the host system and other containers. Although containers can use resources from the host, commands run from a container have a very limited ability to interface directly with the host.

Some containers, however, are intended to access, monitor, and possibly change features on the host system directly. These are referred to as *super privileged containers*. Because of the nature of Red Hat Enterprise Linux Atomic hosts (RHEL Atomic), SPCs offer some important uses for RHEL Atomic hosts. For example:

- The RHEL Atomic Host is meant to be lean. Many tools that you might want to use to manage or troubleshoot RHEL Atomic host are not included by default.
- Because Atomic Host does not allow for packages to be installed using **yum** or **rpm** commands, the best way to add tools from RHEL or third parties on to a RHEL Atomic Host is to include those tools in a container.
- You can bring an SPC into a RHEL Atomic Host, troubleshoot a problem, then remove it when it is no longer needed, to free up resources.

Red Hat produces several SPCs that are tailored specifically to run on RHEL Atomic hosts, and more are in the pipeline for later. These include:

- **RHEL Atomic Tools Container Image:** This container can be thought of as the administrator's shell. Many of the debugging tools (such as `strace`, `traceroute`, and `sosreport`) and man pages that an administrator might use to diagnose problems on the host are in this container.
- **RHEL Atomic rsyslog Container Image:** This container runs the `rsyslogd` service, allowing you to offload log messages to a centralized server or to manage log files in RHEL Atomic. Note that the `systemd-journald` service is collecting all logging data on the RHEL Atomic Host, even if you do not install the `rsyslog` container.
- **RHEL Atomic System Activity Data Collector (sadc) Container Image:** This container runs the `sadc` service from the `sysstat` package and causes the RHEL Atomic system to gather data continuously that can be read later by the `sar` command.
- **RHEL Atomic System Security Services Daemon (SSSD) Container Image:** This container allows the Red Hat Enterprise Linux Atomic Host authentication subsystem to be connected to central identity providers like Red Hat Identity Management and Microsoft Active Directory.

Super privileged containers can be found in other Red Hat products as well. For example, here are reasons some containers need extra privileges and examples of containers requiring those features that are part of the Red Hat OpenShift Container Platform:

- **Access to docker socket:** Communicating with the `docker` service via the `docker` socket requires special privilege. Containers that need to do that include the `openshift3/node` and `openshift3/openshift-docker-builder` containers.



- **Need to load kernel modules:** Any container needing to load kernel modules must be privileged. These include openshift3/openswitch and openshift3/ose-keepalived-ipfailover containers.
- **Need to bypass SELinux:** If a container won't work with SELinux in enforcing mode, it needs special privileges to escape SELinux confinement. The openshift3/logging-fluentd container is an example of such a container. (Note that it is better to create an SELinux policy for the container and label the files it needs to access appropriately so it can work with SELinux.)

The OpenShift containers just described are all available from the Red Hat Registry ([registry.access.redhat.com](https://registry.access.redhat.com)). Search the [Red Hat Container Catalog](#) for descriptions of those containers.

Using the RHEL Atomic Tools Container Image as an example, the next section illustrates how super privileged containers are run and how host features are accessed from an SPC.

## 4.2. RUNNING PRIVILEGED CONTAINERS

Running a **docker** command to include every option you need to run as a super privileged container would require a long and complicated command line. For that reason, we have made it simpler by introducing the **atomic** command to run containers. If you run an **atomic** command like the following:

```
# atomic run rhel7/rhel-tools
[root@localhost /]#
```

It creates and starts up the rhel-tools container using the docker command with multiple options. This makes it simpler to use and execute containers in the RHEL Atomic Host. The resulting docker command is as follows:

```
docker run -it --name rhel-tools --privileged \
  --ipc=host --net=host --pid=host -e HOST=/host \
  -e NAME=rhel-tools -e IMAGE=rhel7/rhel-tools \
  -v /run:/run -v /var/log:/var/log \
  -v /etc/localtime:/etc/localtime -v /:/host rhel7/rhel-tools
```

By understanding what options are run for a super privileged container you can better understand how you might want to use those options when running your own containers that need to access resources on the host. Here are descriptions of those options:

- **-i -t:** Open a terminal device (**-t**) and run interactively (**-i**).
- The **--name** option sets the name of the container (rhel-tools, in this case).
- The **--privileged** option turns off the Security separation, meaning a process running as root inside of the container has the same access to the RHEL Atomic host that it would have if it were run outside the container.
- The **--ipc=host**, **--net=host**, and **--pid=host** flags turn off the ipc, net, and pid namespaces inside the container. This means that the processes within the container see the same network and process table, as well as share any IPCs with the host processes.

There several options to set environment variables inside the container (**-e**). You can refer to any of these options from the shell that opens when you start the container (for example, **echo \$HOST**). These include:

- **-e HOST=/host**: Sets the location of the host filesystem within the container (in other words, where / from the host is mounted). You can append \$HOST to any file name so a command you run accesses that file on the host instead of within the container. For example, from within the container, **\$HOST/etc/passwd** accesses the **/etc/passwd** file on the host.
- **-e NAME=rhel-tools**: Sets the name of the container (what you see when you type **docker ps**).
- **-e IMAGE=rhel7/rhel-tools**: Identifies the name of the image (what you see when you type **docker images**).

Several files and directories from the host (in addition to what is normally mounted from the host to a container) are mounted from the host file system to the container. These include:

- **-v /run:/run**: The **-v /run:/run** option mounts the **/run** directory from the host on the **/run** directory inside the container. This allows processes within the container to talk to the host's dbus service and talk directly to the systemd service. Processes within the container can even communicate with the docker daemon.
- **-v /var/log:/var/log**: Allows commands run within the container to read and write log files from the host's **/var/log** directory.
- **-v /etc/localtime:/etc/localtime**: Causes the host system's timezone to be used with the container.
- **-v /:/host**: Mounting / from the host on **/host** allows a process within the container to easily modify content on the host. Running **touch /host/etc/passwd** would actually act on the **/etc/passwd** file on the host.

The last argument identifies **rhel7/rhel-tools** as the image to run.

In the case of the RHEL Tools privileged container, when you run it, a shell opens and you can start using the commands from inside that container. As an alternative, you could add an option to the end of the **atomic** or **docker** command line to run a particular command (such as **sosreport** or **traceroute**). The next section describes how to begin investigating that container.

### 4.2.1. Understanding Name Spaces in Privileged Containers

Many of the basic Red Hat Enterprise administrative commands have been modified to be aware they are running in a container. For example, when you run **sosreport** inside the RHEL Atomic Tools Container, it knows to use **/host** as the root of the file system and not **/**. When you run other commands from within the RHEL Atomic Tools Container (or any privileged container), however, it might help to understand how they may behave differently when run in a privileged container, based on the following topics:

- **Privileges**

A privileged container runs applications as root user on the host by default. The container has this ability because it runs with an **unconfined\_t** SELinux security context.

- **Mount Tables**

When you use tools such as **df** and **mount** to see what file systems are mounted, you see different information from inside the privileged container than you would see if you ran the same command directly on the host. That's because the two environments maintain their own mount table.

- **Process Tables**

Unlike a regular container, that only sees the processes running inside the container, running a **ps -e** command within a privileged container (with **--pid=host** set) lets you see every process running on the host. So, you can pass a process ID from the host to commands that run in the privileged container (for example, **kill PID**). With some commands, however, permissions issues could occur when they try to access processes from the container.

- **Inter-process communications**

The IPC facility on the host is accessible from within the privileged container. So, you can run commands such as **ipcs** to see information about active message queues, shared memory segments, and **semaphore** sets on the host.

## 4.3. USING THE ATOMIC TOOLS CONTAINER IMAGE

The Red Hat Enterprise Linux Atomic Tools Container (RHEL Tools Container) is a docker-formatted image that includes a set of software tools for troubleshooting and investigating a Red Hat Enterprise Linux Atomic (RHEL Atomic) Host. Designed to run as a privileged container, the RHEL Tools Container allows you to interact directly with the RHEL Atomic Host system to uncover and solve problems. Inside the RHEL Tools Container are popular system administration tools such as **dig**, **iotop**, **ip**, **ss**, **less**, **ncat**, **lspci**, **perf**, **screen**, **strace**, **sar**, **tcpdump**, and **vim**. Most of these tools are not included with RHEL Atomic.

This section covers:

- How to get and run the RHEL Atomic Tools Container
- How the RHEL Atomic Tools Container works
- What commands are in the RHEL Atomic Tools Container and how to use them

### 4.3.1. Overview

RHEL Atomic is designed to be a light-weight, streamlined version of Red Hat Enterprise Linux that is configured and tuned specifically for running Linux containers. It is kept light so it can consume minimal amounts of resources during deployment and run efficiently once deployed. This makes RHEL Atomic particularly suited for hosting containers in cloud environments.

One of the problems of the reduced size of Atomic is that many of the standard RHEL tools are not installed in Atomic. To make the problem worse, the support for installing additional software packages is limited.

The RHEL Tools Container solves this problem. It provides a chosen set of popular system administration tools. You can install this container either during the deployment of the Atomic system or later, when you need to troubleshoot a problem.

Consider these facts about the RHEL Tools Container:

- **It opens privileges.** Containers, by default, cannot see most of the Atomic Host's file system or namespaces (networking, IPC, process table, and so on). However, because the RHEL Tools Container runs as a privileged host and opens access to host namespaces and features, most commands you run from within that container will be able to view and act on the host as if run directly on the host.
- **The tools might behave differently.** Some commands, when run from within the container, even with privileges open, will behave differently than if run directly on the host system. See [Running Commands from the RHEL Tools Container](#) for examples of this.

### 4.3.2. Getting and Running the RHEL Tools Container

To get and run the RHEL Tools Container:

1. Install RHEL Atomic Host using the [Installation and Configuration Guide](#). The RHEL Tools Container is designed to run on RHEL Atomic Host systems.
2. Pull the RHEL Tools Container image to the Atomic Host:

```
# docker pull rhel7/rhel-tools
```

3. Run the RHEL Tools Container:

```
# atomic run rhel7/rhel-tools
[root@localhost /]#
```

You now have a shell open inside the container, with all the tools in the container ready to run. When you are finished using the tools, run **exit**.

### 4.3.3. Running Commands from the RHEL Tools Container

The following examples show some of the commands from the RHEL Tools Container and the differences between running them in the container and on the host system:

- **blktrace**: To use **blktrace** from within a container, you need to first mount the **debugfs** file system. In this example the administrator mounts **debugfs** and runs **blktrace**:

```
# mount -t debugfs debugfs /sys/kernel/debug/
# blktrace /dev/vda
^C
=== vda ===
  CPU 0:          38086 events,      1786 KiB data
  Total:          38086 events (dropped 0),      1786 KiB
data
```

- **useradd**: To add a user to do non-root activities within the container, you can use the **useradd** command:

```
# useradd jjones
# su - jjones
[jjones@example ~]$
```

### 4.3.4. More Information About Running RHEL Tools Container

Note these details about the RHEL Tools Container:

- The container image that you pull is named **rhel7/rhel-tools**. However, once you run it, a container is created using that image. That container is called **rhel-tools**.
- Changes you make to a container (for example, **yum install package**) continue to exist even after you stop and start the container again. This means that **atomic run rhel7/rhel-tools** will not pull down any files or do any additional setup on the host the second time you run it.
- To see the name of the container, even after it is stopped, type **docker ps -a**.

- Unless you explicitly remove the container (**docker rm rhel-tools**), the container continues to exist on your system.
- Even if you remove the **rhel7/rhel-tools** container *image*, the **rhel-tools** container, if there is one, is retained. While the container is retained, you cannot upgrade the container image. To upgrade the **rhel7/rhel-tools** image, preserve any files from the existing container that you want to keep by copying them somewhere on **/host**. Then remove the container by running **docker rm rhel-tools**. Finally, get the updated container image by running **docker pull rhel7/rhel-tools**.
- Commands that should run directly on the Atomic host include those related to systemd (**systemctl** and **journalctl**), LVM (**lvm**, **lvdisplay**, **vgdisplay** and so on), the **atomic** command, and any commands that modify block devices.
- If you have any issues with the RHEL Tools Container, you can file bugs and RFEs at **bugzilla.redhat.com** under the "Red Hat Enterprise Linux" product and the "rhel-tools-docker" component.

## 4.4. USING THE ATOMIC SUPPORT TOOLS CONTAINER IMAGE

The Red Hat Enterprise Linux Atomic Support Tools container (Support Tools Container) is a docker-formatted image that includes the **sos**, **strace**, **tcpdump**, and **redhat-support-tool** tools.

The four tools comprising the Support Tools Container were previously part of the RHEL Tools Container (**strace** and **tcpdump** still are). The two containers are essentially the same but with a different set of included tools. Thus, the information in the [Using the Atomic Tools Container Image](#) section applies to Support Tools Container, except for two things: set of available tools and container size.

This section covers:

- The relationship between the Support Tools Container and the RHEL Tools Container
- How to get and run the RHEL Atomic Tools Container
- What commands are in the RHEL Atomic Tools Container and how to use them

### 4.4.1. How It Is Different from RHEL Atomic Tools

The Support Tools Container has been created as part of the effort to reduce the size of the huge RHEL Tools Container.

A frequent use case for the RHEL Tools Container has been to gather **sosreports** and run **tcpdump**, **strace**, and **redhat-support-tool** tools. That is why the dedicated Support Tools Container, which consists of these four tools, has been created.

The Support Tools Container is much smaller than the RHEL Tools Container, 218MB instead of 370MB, and can be installed and used more liberally.

The **sos** and **redhat-support-tool** tools have been removed from the RHEL Tools Container, while the **tcpdump** and **strace** remain there.

For more information on the splitting of the RHEL Tools Container, see [the RHEL Atomic Host 7.4.3 Release Note](#).

### 4.4.2. Getting and Running the Support Tools Container

To get and run the Support Tools Container:

1. Install RHEL Atomic Host using the [Installation and Configuration Guide](#). The Support Tools Container is designed to run on RHEL Atomic Host systems.
2. Pull the Support Tools Container image to the Atomic Host:

```
# docker pull rhel7/support-tools
```

3. Run the RHEL Tools Container:

```
# atomic run rhel7/support-tools
[root@localhost /]#
```

You now have a shell open inside the container, with all the tools in the container ready to run. When you are finished using the tools, run **exit**.

#### 4.4.3. Running Commands from the Support Tools Container

The following examples two of the commands from the Support Tools Container and the differences between running them in the container and on the host system:

- **sosreport**: The **sosreport** command includes an atomic plugin that makes it container-aware. This means that you can run **sosreport** to generate a report with almost exactly the same results as if run directly on the host. For example:

```
# sosreport
Please enter your first initial and last name
[localhost.localdomain]: jjones
Please enter the case id that you are generating this report for:
12345678
...
# ls /host/var/tmp
sosreport-jjones.12345678-20150203102944.tar.xz
sosreport-jjones.12345678-20150203102944.tar.xz.md5
```

Note that the report is copied to the **/var/tmp** directory on the host, which makes it available after you close the container. Additionally, the host's root file system is mounted on **/host** within the container, which allows you to access the report in the **/host/var/tmp** directory within the container.

- **strace**: Since the host's process table is visible from within the RHEL Tools Container, the commands that take a process ID as an argument work from within the container. This enables using the **strace** command:

```
# ps -ef | grep ssh
root          998          1  0 Jan29 ?           00:00:00 /usr/sbin/sshd -D
# strace -p 998
Process 998 attached
select(7, [3 4], NULL, NULL, NULL ...
```

## 4.5. USING THE ATOMIC RSYSLOG CONTAINER IMAGE

### 4.5.1. Overview

The Red Hat Enterprise Linux rsyslog Atomic Container Image is a Docker formatted image that is designed to run on a Red Hat Enterprise Linux Atomic (RHEL Atomic) host.

With this container, you can start an rsyslogd daemon that:

- Uses configuration files and log files that are stored on the Atomic host's file system
- Can be configured to provide any standard rsyslog features, including directing log message to a remote log host

This topic describes how to get and run the RHEL rsyslog container.

Because the rsyslog service is not installed on a Red Hat Enterprise Linux Atomic Host, the rsyslog container offers a way of adding that service to an Atomic host.

Here are some of the features of the rsyslog container:

- **Installs from atomic command:** When you use the **atomic install** command to get and run the rsyslog container, several things happen. The container itself is pulled from the registry, files and directories needed by the rsyslog service are added to the host, and the container is started with **docker run**.
- **Configure from the host:** Because the files needed by the rsyslog service are stored on the Atomic host, there is no need to go inside the container itself. All configuration can be done from the host.
- **Restarting the service:** If you make any changes to the configuration, to pick up the changes you just have to stop, remove and restart the container again (**docker stop rsyslog; docker rm rsyslog; atomic run rhel7/rsyslog**).
- **Super privileged container:** Keep in mind that running the rsyslog container opens privileges from that container to the host system. The container has root access to RHEL Atomic host and opens access to privileged configuration and log files.

### 4.5.2. Getting and Running the RHEL rsyslog Container

To use the rsyslog Atomic Container Image on a RHEL Atomic host, you need to install it, load it and run it, as described in the following procedure:

1. **Install RHEL Atomic Host:** To install and configure a RHEL Atomic host, refer to the appropriate installation guide listed on the [Red Hat Enterprise Linux Atomic Host Documentation](#) page.
2. **Install the RHEL rsyslog Container:** While logged into the RHEL Atomic host, get and start the RHEL rsyslog Container by running the following command:

```
# docker pull rhel7/rsyslog
# atomic install rhel7/rsyslog
...
docker run --rm --privileged -v /:/host -e HOST=/host -e
IMAGE=rhel7/rsyslog -e NAME=rsyslog rhel7/rsyslog /bin/install.sh
Creating directory at /host//etc/pki/rsyslog
Installing file at /host//etc/rsyslog.conf
Installing file at /host//etc/sysconfig/rsyslog
```

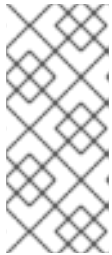
3. **Start the rsyslog container:** To run the RHEL rsyslog container, use the atomic command. The following command starts the container using the docker command with appropriate options:

```
# atomic run rhel7/rsyslog
docker run -d --privileged --name rsyslog --net=host -v
/etc/pki/rsyslog:/etc/pki/rsyslog -v
/etc/rsyslog.conf:/etc/rsyslog.conf -v /etc/rsyslog.d:/etc/rsyslog.d
-v /var/log:/var/log -v /var/lib/rsyslog:/var/lib/rsyslog -v
/run/log:/run/log -v /etc/machine-id:/etc/machine-id -v
/etc/localtime:/etc/localtime -e IMAGE=rhel7/rsyslog -e NAME=rsyslog
--restart=always rhel7/rsyslog /bin/rsyslog.sh
5803dbade82274158f0694a19fdcd7aac044a2656b2ce96d1aebdb0e30ad5ffd
```

After the atomic command starts, you can see the exact 'docker' command that is run to start the rsyslog container. The rsyslogd container runs as a super privileged container.

4. **Check that the container is running:** Type the following to check that the rsyslog container is running:

```
# docker ps
CONTAINER ID IMAGE
COMMAND CREATED STATUS PORTS NAMES
5803dbade822 registry.access.stage.redhat.com/rhel7/rsyslog:7.1-3
"/bin/rsyslog.sh" 9 minutes ago Up 9 minutes rsyslog
```



#### NOTE

The full name of the image is "registry.access.redhat.com/rhel7/rsyslog:7.1-3", which include both the name of the registry from which it was downloaded and the version of the image obtained. The actual container that is run locally, however, is simply called rsyslog. The difference between the image and container is central to the way docker works.

5. **Check that the rsyslog service is working:** From a shell, type the following to watch as messages come into the `/var/log/messages` file:

```
# tail -f /var/log/messages
```

6. **Generate a log message:** Type the following to generate a log message:

```
# logger "Test that rsyslog is doing great"
```

If the rsyslog service is working, the message should appear from the shell running the tail command. You can start using the rsyslog service on the Atomic host now.

### 4.5.3. Tips for Running rsyslog Container

Here are some tips to help you understand a few other issues related to running the RHEL rsyslog container:

- **Understanding persistent logging:** By default, the Red Hat Enterprise Linux Atomic Host system is configured to log to persistent logs on the local root filesystem with journald by setting the following value in `/etc/systemd/journald.conf`:

■



```
Storage=persistent
```

To configure persistent logging to either local rsyslog logs or to a remote rsyslog server, you may want to disable the local journald persistent logging by changing that line to:

```
Storage=volatile
```

and rebooting the RHEL Atomic Host system. journald will still maintain local logs in a ramdisk if you do this, but will not write them to disk. This can save on local disk IO if the data is already being captured securely in another location. The rsyslog container will still be able to capture and process journald logs.

- **Changing rsyslog configuration:** Every time you change the rsyslog container configuration, you must stop and remove the running rsyslog container, then start a new one. To do that, run the following commands:

```
# docker stop rsyslog
# docker rm rsyslog
# atomic run rhel7/rsyslog
```

- **Log rotation:** In the initial version of the rsyslog container image, there is no support for local rotation of rsyslog log files. This will be added in a future update. But rsyslog can still be used with local log files if space permits. There is no requirement for local log rotation if rsyslog is configured to send logs only to a remote log collection host. Refer to the [Red Hat Enterprise Linux System Administrator's Guide](#) for information on configuring both local and remote logging with rsyslog.

- **Ensure there is enough space for logs.**

- The ideal configuration for many cloud environments is to configure rsyslog to log to a remote rsyslog server.
- If you are logging to local storage, be aware that log rotation within a container currently does not occur. In upcoming releases, we will support the configuring log file size limits for the rsyslog configuration by editing logrotate configuration file (such as those in `/etc/logrotate.d/` directory and the `/etc/logrotate.conf` file). This feature is not yet supported.
- Note especially that the amount of space available on the root file system of Atomic host qcow2 images is limited. A larger space can be provisioned by installing via the Red Hat Enterprise Linux Atomic host anaconda installer ISO image.

- **Image and Container Lifecycle**

If you want to upgrade to a newer version of the Red Hat Enterprise Linux rsyslog Atomic container image, it is not enough to merely download the new image with **docker pull rhel7/rsyslog**. You must also explicitly remove the existing rsyslog container with the following commands, before re-running it, in order to create a fresh container from the new image:

```
# docker pull rhel7/rsyslog
```

If a new image downloads, run the following:

```
# docker stop rsyslog
# docker rm rsyslog
# atomic install rhel7/rsyslog
```

```
# atomic run rhel7/rsyslog
```

## 4.6. USING THE ATOMIC SYSTEM ACTIVITY DATA COLLECTOR (SADC) CONTAINER IMAGE

The Red Hat Enterprise Linux sadc Atomic Container Image is a Docker-formatted containerized version of the system monitoring and data collection utilities contained in the sysstat package. This container is designed to run on a Red Hat Enterprise Linux Atomic host. With this container installed and running, the following occurs on your Atomic system:

- System activity data are gathered on an on-going basis
- Commands such as **cifsiostat**, **iostat**, **mpstat**, **nfsiostat**, **pidstat**, **sadf**, and **sar** are available to display that data. You use the **docker exec sadc** command to run the commands.

This topic describes how to get and run the sadc container.

### 4.6.1. Overview

Because sysstat package (which includes sar, iostat, sadc and other tools) is not installed on a Red Hat Enterprise Linux Atomic host, the sadc container offers a way of adding those utilities to an Atomic host. Here are some of the features of the sadc container:

- **Installs from atomic command:** When you use the "atomic install" command to get and run the sadc container, several things happen. The container itself is pulled from the registry, files and directories needed by the sadc service are added to the host, and the container is started with **docker run**.
- **Configure from the host:** Because the files needed by the sadc data collection service are stored on the Atomic host, there is no need to go inside the container itself. All configuration can be done from the host.
- **Super privileged container:** Keep in mind that running the sadc container opens privileges from that container to the host system. The container has root access to RHEL Atomic host and opens access to privileged configuration and log files. For more information on privileged containers, see Running Privileged Docker Containers in RHEL Atomic.

### 4.6.2. Getting and Running the RHEL sadc Container

To use the sadc container on a Red Hat Enterprise Linux Atomic host, you need to install it, load it and run it, as described in the following procedure:

1. **Install RHEL Atomic Host:** To install and configure a RHEL Atomic host, refer to the appropriate installation guide listed on the Red Hat Enterprise Linux Atomic Host Documentation page.
2. **Install the RHEL sadc Container:** While logged into the RHEL Atomic host, get and start the sadc container by running the following command::

```
# docker pull rhel7/sadc
# atomic install rhel7/sadc
docker run --rm --privileged --name sadc -v /:/host -e HOST=/host -e
IMAGE=rhel7/sadc -e NAME=name rhel7/sadc /usr/local/bin/sysstat-
install.sh
Installing file at /host//etc/cron.d/sysstat
```

```
Installing file at /host//etc/sysconfig/sysstat
Installing file at /host//etc/sysconfig/sysstat.ioconf
Installing file at /host//usr/local/bin/sysstat.sh
```

3. **Start the sadc container:** To run the RHEL sadc container, use the **atomic** command. The following command starts the container using the docker command with appropriate options:

```
# atomic run rhel7/sadc
docker run -d --privileged --name sadc -v
/etc/sysconfig/sysstat:/etc/sysconfig/sysstat -v
/etc/sysconfig/sysstat.ioconf:/etc/sysconfig/sysstat.ioconf -v
/var/log/sa:/var/log/sa -v /:/host -e HOST=/host -e IMAGE=rhel7/sadc
-e NAME=sadc --net=host --restart=always rhel7/sadc
/usr/local/bin/sysstat.sh
11c566e20ec995a164f815d9bb76b4b876c555f507c9f56c41f5009c9b1bebf4
```

After the **atomic** command starts, you can see the exact docker command that is run to start the sadc container. The sadc container runs as a super privileged container. For more information on super privileged containers, refer to *Running Super Privileged Docker Containers on a Red Hat Enterprise Linux Atomic Host*.

4. **Check that the container is running:** Type the following to check that the sadc container is running:

```
# docker ps
CONTAINER ID IMAGE
COMMAND CREATED STATUS PORTS NAMES
11c566e20ec9 registry.access.stage.redhat.com/rhel7/sadc:7.1-3
"/usr/local/bin/syss 3 minutes ago Up 2 minutes sadc
```



#### NOTE

While "registry.access.redhat.com/rhel7/sadc:7.1-3" is the full name of the image, including both the name of the registry from which it was downloaded and the version of the image obtained. The actual container that is run locally, however, is simply called "sadc". The difference between the image and container is central to the way docker works.

5. **Generate sadc data:** From a shell, type the following generate some system activity data and test that sadc is working properly:

```
# docker exec sadc /usr/lib64/sa/sa1 1 1
```

6. **Check that sadc worked properly:** If sadc generated some system activity data, you should be able to see it using the sar command as follows:

```
# docker exec sadc sar
Linux 3.10.0-229.el7.x86_64 (minion1.example.com) 02/27/15 _x86_64_
(1 CPU)

09:31:25 LINUX RESTART
09:32:00 CPU %user %nice %system %iowait %steal %idle
09:32:18 all 0.86 0.00 0.92 0.00 0.00 98.22
```

If `sadc` is working, you should be able to see the data generated by the `sadc` command you just ran. New data should be generated every 10 minutes. So you can run the `sar` command again to make sure that data is being collected in an on-going basis.

### 4.6.3. Tips for Running the `sadc` Container

Here are some tips to help you understand a few other issues related to running the `sadc` container:

- **Running `sysstat` commands:** You can run any of the commands in the `sysstat` package to view data gathered by the `sadc` container. These include `cifsiostat`, `iostat`, `mpstat`, `nfsiostat`, `pidstat`, `sadf`, and `sar`. Because these commands are not on the Atomic host, you must run them using `docker exec`. For example:

```
# docker exec sadc iostat
```

- **Image and Container Lifecycle**

If you want to upgrade to a newer version of the Red Hat Enterprise Linux `sadc` Atomic container image, it is not enough to merely download the new image with `docker pull rhel7/sadc`. You must also explicitly remove the existing `sadc` container with the following commands, before re-running it, in order to create a fresh container from the new image:

```
# docker stop sadc
# docker rm sadc
```

## 4.7. USING THE ATOMIC NET-SNMP CONTAINER IMAGE

The Red Hat Enterprise Linux Atomic Net-SNMP Container (Net-SNMP Container) is a docker-formatted image that provides the Net-SNMP software suite, including an SNMP agent. Net-SNMP allows you to set up performance monitoring of an Atomic Host system using the SNMP protocol.

Non-containerized Net-SNMP is documented in the [RHEL7 System Administrator's Guide](#).

### 4.7.1. Installing and Running the Net-SNMP Container

To install and run the Net-SNMP Container:

1. Pull the Net-SNMP Container image:

```
# atomic pull rhel7/net-snmp
```

2. Install the image:

```
# atomic install --system --system-package=no --name=net-snmp
rhel7/net-snmp
```

Currently, Red Hat recommends using the `--system-package=no` option to prevent `rpmbuild` from creating an RPM file during installation and to fall back to copying files to the host instead.

3. Run the Net-SNMP service:

```
# systemctl start net-snmp
```

### 4.7.2. Running Commands in the Net-SNMP Container

When you need to run the commands, for example commands in the [non-containerized Net-SNMP documentation](#), they must be run inside the container:

```
# atomic run --storage ostree net-snmp COMMAND ARGUMENTS
```

For example, to run `snmpwalk -v3 localhost system`, use:

```
# atomic run --storage ostree net-snmp snmpwalk -v3 localhost system
```

### 4.7.3. Configuring the SNMP Agent

To configure the SNMP Agent, follow the instructions in [Configuring Net-SNMP](#). That document shows how to set the system information and configure authentication.



#### NOTE

If you want increased security, configure Net-SNMP to use version 3 of SNMP.

During configuring, you will change configuration files such as `/etc/snmp/snmpd.conf` and run commands such as `net -snmp -create -v3 -user`.

When you change configuration files:

1. Stop the `net -snmp` service:

```
# systemctl stop net-snmp
```

2. Make changes in the configuration files.

3. Start the `net -snmp` service:

```
# systemctl start net-snmp
```

When you run commands:

1. Make sure the `net -snmp` service is running:

```
# systemctl start net-snmp
```

2. Run the commands.

### 4.7.4. Monitoring an Atomic Host System Using Net-SNMP

See [Retrieving Performance Data over SNMP](#) to learn how to access performance data about the managed Atomic Host over SNMP.

### 4.7.5. Extending Net-SNMP to Provide Application Metrics

See [Extending Net-SNMP](#) for information on extending Net-SNMP to provide application metrics in addition to the system metrics. The document shows how to extend Net-SNMP using shell scripts and Perl scripts.

Note that RHEL Atomic Host is a containerized system, and might not support some extensions.

## 4.8. USING THE ATOMIC SSSD CONTAINER IMAGE

### 4.8.1. Overview

The Red Hat Enterprise Linux Atomic SSSD Container Image provides the **ipa-client-install** and **realmd** tools for enrolling the host to an Identity Management (IdM) server or for connecting it to an Active Directory domain. It makes it possible to run the System Security Services Daemon (SSSD) in a container to provide identity, authentication, and authorization services to the Atomic Host itself and to applications running in other containers on the same Atomic Host system.

Usage of the SSSD container is described in the [Using Containerized Identity Management Services](#) guide:

- For conceptual information on the SSSD container and other IdM containers, see [Overview of the Containerized Identity Management Services](#).
- For information on using the SSSD container, see [Using the sssd Container](#).

IdM and non-containerized SSSD are described in other Identity Management Guides:

- For introduction to IdM and documentation for the **ipa-client-install** utility, see [Linux Domain Identity, Authentication, and Policy Guide](#).
- For documentation for **realmd** and using SSSD with Active Directory, see [Windows Integration Guide](#). In particular, see [Using Active Directory as an Identity Provider for SSSD](#).
- For documentation for non-containerized SSSD, see [System-Level Authentication Guide](#).

## 4.9. USING THE ATOMIC RHEVM-GUEST-AGENT CONTAINER IMAGE

### 4.9.1. Overview

The **rhev-guest-agent** container image is a Docker-formatted container that is used to run an agent inside of virtual machines on Red Hat Virtualization hosts. Communications between that agent and the Red Hat Virtualization Manager allows that manager to both monitor and change the state of the agent's virtual machine.

This topic describes how to get and run the **rhev-guest-agent** container.

#### 4.9.1.1. Overview of the **rhev-guest-agent** Container

A **rhev-guest-agent** running inside a virtual machine on a Red Hat Virtualization host allows a Red Hat Virtualization Manager (RHV-M) to access data and control the state of that virtual machine (VM). That agent provides information to the RHV-M that include heart-beat data, CPU usage, IP addresses, installed applications, and other content related to the health and processing of the virtual machine. Likewise, through the agent, the RHEV-M can shutdown, restart or check the status of the virtual machine.

## 4.9.2. Getting and Running the RHEL rhvm-guest-agent Container

To use the rhvm-guest-agent container on a RHEL or RHEL Atomic host system, you need to install it and run it as described in the following procedure:

1. **Install RHEL or RHEL Atomic Host:** Install and configure a RHEL or RHEL Atomic host system as a virtual machine in a Red Hat Virtualization environment (from the RHV Manager). The rhvm-guest-agent is made to run on a RHEL system on a RHEV Host so it can be managed by a RHV Manager.
2. **Pull the rhvm-guest-agent container:** While logged into the virtual machine, pull the rhvm-guest-agent container as follows:

```
# docker pull registry.access.redhat.com/rhev4/rhvm-guest-agent
Using default tag: latest
Trying to pull repository registry.access.redhat.com/rhev4/rhvm-
guest-agent ...
latest: Pulling from registry.access.redhat.com/rhev4/rhvm-guest-
agent
16dc1f96e3a1: Pull complete
83abca08dea6: Pull complete
Digest:
sha256:0ea0bf8729957454e1f134747d7539e37ea128f39e9757271eea4cbba8737
655
Status: Downloaded newer image for
registry.access.redhat.com/rhev4/rhvm-guest-agent:latest
```

3. **Install the rhvm-guest-agent container:** Use the **atomic** command to install the rhvm-guest-agent container as follows:

```
# atomic install rhv4/rhvm-guest-agent
docker run --rm --privileged --pid=host -v /:/host -e HOST=/host -e
IMAGE=rhev4/rhvm-guest-agent -e NAME=rhvm-guest-agent rhv4/rhvm-
guest-agent /usr/local/bin/ovirt-guest-agent-install.sh
Host group is
Creating ovirtagent group on host system
Host user is
Creating ovirtagent user on host system
```

Notice that the process of installing the container on the hosts system includes opening privileges to the host system and creating ovirtagent user and group accounts on the host.

4. **Start the rhvm-guest-agent container:** To run the RHEL rhvm-guest-agent container, use the **atomic** command as follows:

```
# atomic run rhv4/rhvm-guest-agent
docker run --privileged --pid=host --net=host -v /:/host -e
HOST=/host -v /proc:/hostproc -v /dev/virtio-
ports/com.redhat.rhvm.vdsm:/dev/virtio-ports/com.redhat.rhvm.vdsm
--env container=docker --restart=always -e IMAGE=rhev4/rhvm-guest-
agent -e NAME=rhvm-guest-agent rhv4/rhvm-guest-agent
```

After the **atomic** command starts, you can see the exact **docker** command that is run to start the rhvm-guest-agent container. In this case, the container is set to always restart if the service ever goes down (**--restart=always**). See the "Tips" section for information about privileges that

are open to the host system.

### 4.9.3. Tips for Running the `rhev-guest-agent` Container

Here are some tips to help you understand a few other issues related to running the `rhev-guest-agent` container:

- **Privileges opened:** The `rhev-guest-agent` is a super privileged container, opening up various features on the host. Privileges `rhev-guest-agent` opens include: `--privileged` (turns off security separation, allowing root access to the host), `--pid=host` (allows access to host process table), `--net=host` (allows access to host network interfaces), and `-v /:/host` (mount host's root file system in the container). Several other host assets are mounted inside the container as well. For more information on the implications of opening these privileges, see [Running Super-privileged Containers](#).
- **Viewing `rhev-guest-agent`:** The data made accessible by the `rhev-guest-agent` can be displayed from the Red Hat Virtualization Manager. From the RHV-M web-based interface, select the virtual machine on which the agent is running to be able to view information collected from the agent about the health and activity of that virtual machine.
- **Information, Notifications, and Actions:** The `rhev-guest-agent` provides information, notifications, and actions to the RHV-M. To see details about what the `rhev-guest-agent` provides, you can view the upstream [oVirt-guest-agent](#) page.
- **Image and Container Lifecycle**  
If you want to upgrade to a newer version of the `rhev-guest-agent` container image, it is not enough to merely download the new image with **`docker pull`**. You must also explicitly remove the existing `rhev-guest-agent` container with the following commands, before re-running it, in order to create a fresh container from the new image:

```
# docker stop rhev-guest-agent
# docker rm rhev-guest-agent
```

## 4.10. USING THE ATOMIC RHEL7 INIT CONTAINER IMAGE

### 4.10.1. Overview

The Red Hat Enterprise Linux 7 Init Image allows creating containerized services based on the **`systemd`** init system. This container image configures `systemd` in an OCI container and enables running one or more services in a RHEL7 user space using unit files, init scripts, or both.

This image is maintained by Red Hat and is updated regularly. Use this image like a regular Red Hat Enterprise Linux 7 system. Tools such as **`yum`**, **`gzip`**, and **`bash`** are provided by default.

You never run this container. It is only used when creating your container images.

Note that behavior of containers created using this image might be different from behavior of the corresponding application on a non-Atomic RHEL7 system.

**IMPORTANT:** To run the `rhel7-init` container on a RHEL or RHEL Atomic host system with SELinux enabled, the **`container_manage_cgroup`** boolean must be turned on. The boolean is off by default. You can turn that boolean on permanently by typing the following on the host system: **`setsebool -P container_manage_cgroup 1`**



### 4.10.2. Getting the Atomic RHEL7 Init Container Image

To use the RHEL7 Init Atomic Container Image on a RHEL Atomic host, you need to install an Atomic host system:

1. **Install RHEL Atomic Host:** To install and configure a RHEL Atomic host, refer to the appropriate installation guide listed on the [Red Hat Enterprise Linux Atomic Host Installation and Configuration Guide](#).
2. **Pull the RHEL7 Init Container:** While logged into the RHEL Atomic host, get the RHEL7 Init Container by running the following command:

```
# docker pull rhel7-init
```

3. **Allow init container to manage cgroups:** Turn on the `container_manage_cgroup` SELinux boolean on the host system to allow the init image to run:

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

### 4.10.3. Creating Container Images based on the Atomic RHEL7 Init Container Image

To assemble a new image from the RHEL7 Init Image:

1. Create a Dockerfile that includes these entries:

```
FROM rhel7-init
```

and

```
RUN yum -y install <daemon> && yum clean all && systemctl enable <daemon>
```

Substitute `<daemon>` with the name of the package containing the daemon for which you are creating the image.

For example, to create an Apache container image:

```
RUN yum -y install httpd && yum clean all && systemctl enable httpd
```

2. Assemble the image:

```
# docker build --tag <new_container_name> <directory_containing_dockerfile>
```

For example, to assemble the Apache container image:

```
# docker build -t myhttpd .
```

3. *(optional)* Run the new image:

```
# docker run <options> <new_container_name>
```

For example, to run the Apache container:

```
# docker run -d -p 80:80 --name my-httpd -v /var/log/httpd myhttpd
```

## 4.11. USING THE ATOMIC RHEL6 INIT CONTAINER IMAGE

### 4.11.1. Overview

The Red Hat Enterprise Linux 6 Init Image allows creating containerized services in a RHEL6-like environment. This container image enables running one or more services in a RHEL6 user space using init scripts. While the init system for Red Hat Enterprise Linux 6 is upstart, this image uses another init system, so no upstart-specific feature is supported in this image.

This image is maintained by Red Hat and is updated regularly. Use this image like a regular Red Hat Enterprise Linux 6 system. Tools such as **yum**, **gzip**, and **bash** are provided by default.

You never run this container. It is only used when creating your container images.

Note that behavior of containers created using this image might be different from behavior of the corresponding application on a non-Atomic RHEL6 system.

### 4.11.2. Getting the Atomic RHEL6 Init Container Image

To use the RHEL6 Init Atomic Container Image on a RHEL Atomic host, you need to install it:

1. **Install RHEL Atomic Host:** To install and configure a RHEL Atomic host, refer to the appropriate installation guide listed on the [Red Hat Enterprise Linux Atomic Host Installation and Configuration Guide](#).
2. **Pull the RHEL6 Init Container:** While logged into the RHEL Atomic host, get the RHEL6 Init Container by running the following command:

```
# docker pull rhel6/rhel6-init
```

### 4.11.3. Creating Container Images based on the Atomic RHEL6 Init Container Image

To assemble a new image from the RHEL6 Init Image:

1. Create a Dockerfile that includes these entries:

```
FROM rhel6-init
```

and

```
RUN yum -y install <daemon> && yum clean all && chkconfig <daemon>  
on
```

Substitute **<daemon>** with the name of the package containing the daemon for which you are creating the image.

For example, to create an Apache container image:

■

```
┃ RUN yum -y install httpd && yum clean all && chkconfig httpd on
```

2. Assemble the image:

```
┃ # docker build --tag <new_container_name>  
┃ <directory_containing_dockerfile>
```

For example, to assemble the Apache container image:

```
┃ # docker build -t myhttpd .
```

3. *(optional)* Run the new image:

```
┃ # docker run <options> <new_container_name>
```

For example, to run the Apache container:

```
┃ # docker run -d -p 80:80 --name my-httpd -v /var/log/httpd myhttpd
```

## CHAPTER 5. FINDING, RUNNING, AND BUILDING CONTAINERS WITHOUT DOCKER

### 5.1. OVERVIEW

The **docker** daemon and **docker** command represent a bundled container runtime environment for managing containers. With the advent of standards from the [Open Container Initiative](#) (OCI), other runtimes are being developed. One such runtime is [CRI-O](#), which was created along with a set of tools to provide alternate ways to work directly with containers.

The following text describes some of the tools being developed to use with the CRI-O runtime or as alternative container stand-alone tools. These tools can be used with docker-formatted containers or OCI-conformant containers. While some of these commands were made to use with CRI-O, others can also interact with the docker daemon (as a replacement for features of the **docker** command) or used to manage containers with no active runtime environment.

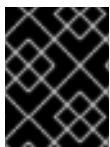


#### NOTE

CRI-O was created primarily as a container runtime for Kubernetes and OpenShift, and is not currently available as a stand-alone service for Red Hat Enterprise Linux. For OpenShift 3.9, CRI-O is available in tech preview. The [OpenShift 3.9 Release Notes](#) tells how to set up CRI-O for OpenShift.

This chapter describes the following tools that you can use instead of the **docker** command and daemon to work with OCI and Docker containers:

- **podman**: The **podman** command can run and manage containers and container images. It supports most of the same features and command options you find in the **docker** command, with the main differences being that **podman** doesn't require the docker service or any other active container runtime for the command to work. Also, **podman** stores its data in the same directory structure used by CRI-O, which will allow **podman** to eventually work with containers being actively managed by CRI-O in OpenShift.
- **runc**: The **runc** command can be used to start up docker-formatted or OCI containers.
- **skopeo**: The **skopeo** command lets you inspect images from container image registries, get images and image layers, and use signatures to create and verify images.



#### IMPORTANT

The **skopeo** version currently shipped with RHEL and Atomic Host only works with v2 registries and not the v1 registries Red Hat is currently using.

- **buildah**: The **buildah** command can be used in place of **docker build** to build container images from Dockerfiles and, ultimately, files in other formats.

The following sections describe **podman**, **runc**, **skopeo**, and **buildah**.

### 5.2. USING PODMAN TO WORK WITH CONTAINERS

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)
- **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 uses the CRI-O back-end store directory, `/var/lib/containers`, instead of using the Docker storage location (`/var/lib/docker`), by default.
- Although **podman** and CRI-O share the same storage directory, they cannot interact with each other's containers. (Eventually the two 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 **docker** 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 container 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 Cockpit or the **atomic** command) and respond to them.
  - 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).



## NOTE

The **podman** command is considered to be technology preview for RHEL and RHEL Atomic 7.5.1.

### 5.2.1. Installing podman

To start using **podman** to work with containers, you can simply install it on a Red Hat Enterprise Linux server system or try it on a RHEL Atomic Host (**podman** is preinstalled on RHEL Atomic Host 7.5.1 or later). No container runtime is needed to use **podman**.

To install **podman** on a RHEL server system, do the following:

```
# subscription-manager repos --disable='*'
# subscription-manager repos --enable=rhel-7-server-rpms
# subscription-manager repos --enable=rhel-7-server-extras-rpms
# subscription-manager repos --enable=rhel-7-server-optional-rpms
# yum install podman -y
```

## 5.2.2. Running containers with podman

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 5.1. Commands supported by podman**

podman command	Description	podman command	Description
<b>attach</b>	Attach to a running container	<b>commit</b>	Create new image from changed container
<b>build</b>	Build an image using Dockerfile instructions	<b>create</b>	Create, but do not start, a container
<b>diff</b>	Inspect changes on container's filesystems	<b>exec</b>	Run a process in a running container
<b>export</b>	Export container's filesystem contents as a tar archive	<b>help, h</b>	Shows a list of commands or help for one command
<b>history</b>	Show history of a specified image	<b>images</b>	List images in local storage
<b>import</b>	Import a tarball to create a filesystem image	<b>info</b>	Display system information
<b>inspect</b>	Display the configuration of a container or image	<b>kill</b>	Send a specific signal to one or more running containers
<b>load</b>	Load an image from an archive	<b>login</b>	Login to a container registry
<b>logout</b>	Logout of a container registry	<b>logs</b>	Fetch the logs of a container

<b>mount</b>	Mount a working container's root filesystem	<b>pause</b>	Pauses all the processes in one or more containers
<b>ps</b>	List containers	<b>port</b>	List port mappings or a specific mapping for the container
<b>pull</b>	Pull an image from a registry	<b>push</b>	Push an image to a specified destination
<b>restart</b>	Restart one or more containers	<b>rm</b>	Remove one or more containers from host. Add <b>-f</b> if running.
<b>rmi</b>	removes one or more images from local storage	<b>run</b>	run a command in a new container
<b>save</b>	Save image to an archive	<b>search</b>	search registry for image
<b>start</b>	Start one or more containers	<b>stats</b>	Display percentage of CPU, memory, network I/O, block I/O and PIDs for one or more containers
<b>stop</b>	Stop one or more containers	<b>tag</b>	Add an additional name to a local image
<b>top</b>	Display the running processes of a container	<b>umount, unmount</b>	Unmount a working container's root filesystem
<b>unpause</b>	Unpause the processes in one or more containers	<b>version</b>	Display podman version information

### 5.2.3. 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**.

#### 5.2.3.1. Pull a container image to the local system

```
# podman pull registry.access.redhat.com/rhel7/rhel
```

```
Trying to pull registry.access.redhat...Getting image source signatures
Copying blob sha256:d1fe25896eb5cbcee...
Writing manifest to image destination
Storing signatures
fd1ba0b398a82d56900bb798c...
```

### 5.2.3.2. List local container images

```
# podman images
REPOSITORY          TAG         IMAGE ID
CREATED            SIZE
registry.access.redhat.com/rhel7/rhel-minimal  latest     de9c26f23799  5
weeks ago          80.1MB
registry.access.redhat.com/rhel7/rhel         latest     fd1ba0b398a8  5
weeks ago          211MB
```

### 5.2.3.3. Run a container image

This runs a container image and opens a shell inside the container:

```
# podman run -it registry.access.redhat.com/rhel7/rhel /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
#
```

### 5.2.3.4. List containers that are running or have exited

```
# podman ps -a
CONTAINER ID   IMAGE
COMMAND
CREATED AT           STATUS           PORTS NAMES
440becd26893   registry.access.redhat.com/rhel7/rhel-minimal:latest
/bin/bash
2018-05-10 09:02:52 -0400 EDT   Exited (0) About an hour ago
happy_hodgkin
8414218c04f9   registry.access.redhat.com/rhel7/rhel:latest
/bin/bash
2018-05-10 09:48:07 -0400 EDT   Exited (0) 14 minutes ago
nostalgic_boyd
```

### 5.2.3.5. Remove a container or image

Remove a container by its container ID:

```
# podman rm 440becd26893
```

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

```
# podman rmi registry.access.redhat.com/rhel7/rhel-minimal
```



```
# podman rmi de9c26f23799
# podman rmi -f registry.access.redhat.com/rhel7/rhel:latest
```

### 5.2.3.7. Build a container

```
# cat Dockerfile
FROM registry.access.redhat.com/rhel7/rhel-minimal
ENTRYPOINT "echo "Podman build 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.
```

## 5.3. RUNNING CONTAINERS WITH 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. It 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 docker-formatted images with runc.

### 5.3.1. Installing and running containers

The runc package is available for Red Hat Enterprise Linux in the Extras channel. You need to have the Extras channel enabled to install it with yum. If you are using Red Hat Enterprise Linux Atomic Host, the runc package is already included. For a regular RHEL system, to enable the extras repository and install the package, run:

```
$ sudo subscription-manager repos --enable=rhel-7-server-extras-rpms
$ sudo yum install 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:

```
$ 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 docker-formatted Red Hat Enterprise Linux base image (rhel/rhel7) using docker, 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:

```
$ sudo docker pull registry.access.redhat.com/rhel7/rhel
$ sudo docker export $(docker create
registry.access.redhat.com/rhel7/rhel) > 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 the value of terminal from *false*
to *true*
$ sudo runc create -b rhel-runc/ rhel-container
$ sudo 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".

## 5.4. USING SKOPEO TO WORK WITH CONTAINER REGISTRIES

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, or Atomic 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 as described in [Working with Docker Registries](#).
- Pull the latest RHEL 7 image to your local system (**docker pull rhel7/rhel**).
- Retag the RHEL 7 image and push it to your local registry as follows:

```
$ sudo docker tag rhel7/rhel localhost:5000/myrhel7
$ sudo docker push localhost:5000/myrhel7
```

The rest of this section describes how to inspect, copy and get layers from the RHEL 7 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.

### 5.4.1. 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 rhel7/rhel).

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

```
$ sudo 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": "2016-06-10T01:53:48.812217692Z",
  "DockerVersion": "1.10.3",
  "Labels": {},
  "Architecture": "amd64",
  "Os": "linux",
  "Layers": [
  ],
  ...
}
```

Assuming you pushed a container image tagged **localhost:5000/myrhel7** to a docker registry running on your local system, the following command inspects that image:

```
$ sudo skopeo inspect docker://localhost:5000/myrhel7
{
  "Name": "localhost:5000/myrhel7",
  "Tag": "latest",
}
```

```

    "Digest":
    "sha256:4e09c308a9ddf56c0ff6e321d135136eb04152456f73786a16166ce7cba7c904",
    "RepoTags": [
      "latest"
    ],
    "Created": "2016-06-16T17:27:13Z",
    "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": "rhel7/rhel",
      "Release": "75",
      "Vendor": "Red Hat, Inc.",
      "Version": "7.2"
    },
    "Architecture": "amd64",
    "Os": "linux",
    "Layers": [
      "sha256:16dc1f96e3a1bb628be2e00518fec2bb97bd5933859de592a00e2eb7774b6ecf"
    ]
  }
}

```

### 5.4.2. Copying container images with skopeo

This command copies the `myrhel7` container image from a local registry into a directory on the local system:

```

# skopeo copy docker://localhost:5000/myrhel7 dir:/root/test/
INFO[0000] Downloading
myrhel7/blobs/sha256:16dc1f96e3a1bb628be2e00518fec2bb97bd5933859de592a00e2
eb7774b6ecf
# ls /root/test
16dc1f96e3a1bb628be2e00518fec2bb97bd5933859de592a00e2eb7774b6ecf.tar
manifest.json

```

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

### 5.4.3. Getting image layers with skopeo

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

```

# skopeo layers docker://localhost:5000/myrhel7
INFO[0000] Downloading
myrhel7/blobs/sha256:16dc1f96e3a1bb628be2e00518fec2bb97bd5933859de592a00e2
eb7774b6ecf
# find .

```

```
./layers-myrhel7-latest-698503105
./layers-myrhel7-latest-698503105/manifest.json
./layers-myrhel7-latest-
698503105/16dc1f96e3a1bb628be2e00518fec2bb97bd5933859de592a00e2eb7774b6ecf
.tar
```

As you can see from this example, a new directory is created (layers-myrhel7-latest-698503105) and, in this case, a single layer tarball and a manifest.json file are copied to that directory.

## 5.5. 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 runtimes that meet 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.

### 5.5.1. Understanding Buildah

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

- **No Daemon!:** Buildah 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). When you go to commit a container to an image, you can export that container as a local Docker image by indicating **docker-daemon** (stored in /var/lib/docker).



#### NOTE

The containers-storage location that the **buildah** command uses by default is the same place that the CRI-O container runtime 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, should be visible to both.

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**).

The buildah package is technology preview for Red Hat Enterprise Linux version 7.4.4. 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](#)

## 5.5.2. Installing Buildah

The buildah package is available from the Red Hat Enterprise Linux Server Extras repository. From a RHEL Server system with a valid subscription, install the buildah package as follows:

```
# subscription-manager repos --enable=rhel-7-server-rpms
# subscription-manager repos --enable=rhel-7-server-extras-rpms
# 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, type:

```
# rpm -qd buildah
```

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

## 5.5.3. 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 7 image from the Red Hat Registry as a working container to use with the **buildah** command:

```
# buildah from docker://registry.access.redhat.com/rhel7/rhel-minimal
Getting image source signatures
Copying blob...
Writing manifest to image destination
Storing signatures
rhel-minimal-working-container
# buildah images
IMAGE ID      IMAGE NAME      CREATED AT      SIZE
1456eedf8101 registry.access.redhat.com/rhel7/rhel-atomic:latest
                Oct 12, 2017 15:15      74.77 MB

# buildah containers
CONTAINER ID  BUILDER  IMAGE ID      IMAGE NAME      CONTAINER NAME
dc8f21Ag4a47 *        1456eedf8101 registry.access.redhat.com/rhel7/rhel-atomic:latest
                rhel-atomic-working-container
1456eedf8101 registry.access.redhat.com/rhel7/rhel-minimal:latest
                Oct 12, 2017 15:15      74.77 MB

# buildah containers
CONTAINER ID  BUILDER  IMAGE ID      IMAGE NAME      CONTAINER NAME
dc8f21Ag4a47 *        1456eedf8101 registry.access.redhat.com/rhel7/rhel-minimal:latest
                rhel-minimal-working-container
```

Notice that the result of the **buildah from** command is an image (registry.access.redhat.com/rhel7/rhel-minimal:latest) and a working container that is ready to run from that image (rhel-minimal-working-container). Here's an example of how to execute a command from that container:

```
# buildah run rhel-minimal-working-container cat /etc/redhat-release
Red Hat Enterprise Linux Server release 7.4 (Maipo)
```

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

#### 5.5.4. 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.access.redhat.com/rhel7/rhel-minimal
ADD myecho /usr/local/bin
ENTRYPOINT "/usr/local/bin/myecho"
# cat myecho
echo "This container works!"
# chmod 755 myecho
```

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

```
# buildah bud -t myecho .
STEP 1: FROM registry.access.redhat.com/rhel7/rhel-minimal
STEP 2: ADD myecho /usr/local/bin
STEP 3: ENTRYPOINT "/usr/local/bin/myecho"
STEP 4: COMMIT containers-storage:
[devicemapper@/var/lib/containers/storage+/var/run/containers/storage]docker.io/library/myecho:latest
```

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.access.redhat.com/rhel7/rhel-minimal:latest
                                         Oct 12, 2017 15:15  74.77 MB
ab230ac5aba3 docker.io/library/myecho:latest
                                         Oct 12, 2017 15:15   2.854 KB
# buildah containers
```

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

### 5.5.5. 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 **buildah run** to run the working container.

```
# buildah from myecho
myecho-working-container
# buildah containers
CONTAINER ID  BUILDER  IMAGE ID      IMAGE NAME                                CONTAINER NAME
dc8f21af4a47  *        1456eedf8101 registry.access.redhat.com/rhel7/rhel-minimal:latest
                                         rhel-minimal-working-container
6d1ffccb557d  *        ab230ac5aba3 docker.io/library/myecho:latest
                                         myecho-working-container
# buildah 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 **buildah run** ran the container, producing the output: `This container works!`

### 5.5.6. 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",
  "image": "docker.io/library/myecho:latest",
  "image-id":
  "e2b190ac8a37737ec03cfa4c9bfd989845b9bec3aa81ff48d8350d7418d748f6",
  "config": "eyJjcmVh..."
```



```

"ociv1": {
  "created": "2017-10-12T15:15:00.207103Z",
  "author": "Red Hat, Inc.",
  "architecture": "amd64",
  "os": "linux",
  "config": {
    "Entrypoint": [
      "/bin/sh",
      "-c",
      "\\usr/local/bin/myecho\\"
    ],
    "WorkingDir": "/",
    "Labels": {
      "architecture": "x86_64",
      "authoritative-source-url": "registry.access.redhat.com",

```

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

```

# buildah inspect myecho-working-container | less
{
  "type": "buildah 0.0.1",
  "image": "docker.io/library/myecho:latest",
  "image-id":
  "e2b190ac8a37737ec03cfa4c9bfd989845b9bec3aa81ff48d8350d7418d748f6",
  "config": "eyJjcmV...
  "container-name": "myecho-working-container",
  "container-id":
  "70f22e886310bba26bb57ca7afa39fd19af2791c4c66067cb6206b7c3ebdcd20",
  "process-label": "system_u:system_r:svirt_lxc_net_t:s0:c225,c716",
  "mount-label": "system_u:object_r:svirt_sandbox_file_t:s0:c225,c716",
  "ociv1": {
    "created": "2017-10-12T15:15:00.207103Z",
    "author": "Red Hat, Inc.",
    "architecture": "amd64",

```

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.

## 5.5.7. 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.

### 5.5.7.1. 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.access.redhat.com/rhel7/rhel-
minimal:latest
                rhel-minimal-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/176c273fe28c23e5319805a2c4855
9305a57a706cc7ae7bec7da4cd79edd3c02/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
# buildah 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
```

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

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

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

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

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

### 5.5.8. 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/cc92011e9a2b077d03a97c0809f1f
3e7fef0f29bdc6ab5e86b85430ec77b2bf6/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.
```

### 5.5.9. 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
```

### 5.5.10. 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 associated you 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.

#### 5.5.10.1. 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 simple registry with the `docker-distribution` package, as shown below).
- Create or acquire the container image you want to push.
- Use **buildah push** to push the image to the registry.

To install a registry on your local system, start it up, and enable it to start on boot, type:

```
# yum install -y docker-distribution
# systemctl start docker-distribution
# systemctl enable docker-distribution
```

By default, the `docker-distribution` service listens on TCP port 5000 on your localhost.

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 (`localhost:5000`, in this case):

```
# buildah images
IMAGE ID      IMAGE NAME                                CREATED AT      SIZE
cb702d492ee9 docker.io/library/myecho2:latest Nov 21, 2017 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": "sha256:8999fff6050...",
  "RepoTags": [
    "latest"
  ],
  "Created": "2017-11-21T16:50:25.830343Z",
  "Dockerversion": "",
  "Labels": {
    "architecture": "x86_64",
    "authoritative-source-url": "registry.access.redhat.com",
```

At this point, any tool that can pull container images from a container registry can get a copy of your pushed image. For example, 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!
```

### 5.5.10.2. 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 either the **buildah** or **docker** command:

```
# docker run docker.io/textaccountXX/myecho2:latest  
This container works!  
# buildah from docker.io/textaccountXX/myecho2:latest  
myecho2-working-container-2  
# buildah run myecho2-working-container-2  
This container works!
```

## CHAPTER 6. RUNNING SYSTEM CONTAINERS

System containers provide a way to containerize services that need to run before the docker daemon is running. They use different technologies than the Docker-formatted containers, **ostree** for storage, **runc** for runtime, **skopeo** for searching and **systemd** for service management. Previously, such services were provided in the system as packages, or as part of the ostree in Atomic Host. Excluding applications from the Atomic Host system and containerizing them makes the system itself smaller. Red Hat provides the **etcd** and **flannel** services as system containers.



### NOTE

To use the system containers on Atomic Host, you need to have the **atomic** command-line tool version 1.12 or later, along with **ostree** and **runc** utilities (all of which are included on the latest version of Atomic Host). To use system containers on RHEL Server systems, you must be running at least RHEL 7.3.3 (because the ostree package was not available on RHEL server until that release).

Because they are not Docker-formatted containers, you do not use the **docker** command for container management. The **atomic** command-line tool and **systemd** are used to pull, install and manage system containers. Here is a brief comparison between how you pull, install and run docker containers and system containers.

- **docker**

1. **docker pull rhel7/rsyslog**
2. **atomic install rhel7/syslog**
3. **atomic run rhel7/rsyslog**

- **system containers**

1. **atomic pull --storage=ostree rhel7/etcd**
2. **atomic install --system [--set=VARIABLE] rhel7/etcd** (you will notice this command also runs **systemctl start etcd**)

The **atomic install** command supports several options to configure the settings for system containers. The **--set** option is used to pass variables which you would normally set for this service. These variables are stored in the **manifest.json** file.

To uninstall a system image, use:

```
# atomic containers delete rhel7/etcd
# atomic uninstall rhel7/etcd
```

System containers use **runc** as runtime, and docker and runc images are stored in different places on the system: **/var/lib/containers/atomic/\$NAME** and **/etc/systemd/system/\$NAME.service** respectively.

Therefore, when you use **docker images** and **docker ps** you will only see the Docker-formatted containers. The **atomic** tool will show all containers on the system:

```
# atomic containers list -a
```



CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	RUNTIME		
etcd	rhel7/etcd	/usr/bin/etcd-env.sh	2016-10-13 14:21
running	runc		
flannel	rhel7/flannel	/usr/bin/flanneld-ru	2016-10-13 15:12
failed	runc		
1cf730472572	rhel7/cockpit-ws	/container/atomic-ru	2016-10-13 17:55
exited	Docker		
9a2bb24e5978	rhel7/rsyslog	/bin/rsyslog.sh	2016-10-13 17:49
created	Docker		
34f95af8f8f9	rhel7/cockpit-ws	/container/atomic-ru	2016-09-27 19:10
exited	Docker		

Note that unlike docker containers, where the services are managed by the docker daemon, with system containers you have to manage the dependencies between the services yourself. For example, **flannel** is a dependency for **etcd** and when you run flannel, it checks whether etcd is set up (if it is not, flannel will wait).

System containers require root privileges. Because **runc** requires root, containers also run as the root user.

## 6.1. USING THE ETCD SYSTEM CONTAINER IMAGE

### 6.1.1. Overview

The **etcd** service provides a highly-available key value store that can be used by applications that need to access and share configuration and service discovery information. Applications that use etcd include [Kubernetes](#), [flannel](#), [OpenShift](#), [fleet](#), [vulcand](#), and [locksmith](#).

The etcd container described here is what is referred to as a system container. A system container is designed to come up before the docker service or in a situation where no docker service is available. In this case, the etcd container can be used to bring up a keystore for the flannel system container, both of which can then be in place to provide networking services before the docker service comes up.

Prior to RHEL Atomic 7.3.2, there were two containerized versions of the etcd services maintained by Red Hat: etcd 2 (etcd container) and etcd 3 (etcd3 container). With 7.3.2, etcd 2 has been deprecated and etcd 3 is the only supported version of etcd. So the only available etcd container is:

- **etcd**: This is based on etcd version 3.



#### SUPPORT FOR ETCD

Along with the etcd 3 container, the etcd3 rpm package is also deprecated. Going forward, Red Hat expects to maintain only one version of etcd at a time. For RHEL Atomic 7.3.2, system containers in general and the etcd container specifically in supported as Tech Preview only.

Besides bypassing the docker service, this etcd container can also bypass the docker command and the storage area used to hold docker containers by default. To use the container, you need a combination of commands that include **atomic** (to pull, list, install, delete and unstage the image), **skopeo** (to inspect the image), **runc** (to ultimately run the image) and **systemctl** to manage the image among your other systemd services.

Here are some of the features of the etcd container:

- **Supports atomic pull:** Use the **atomic pull** command to pull the container to your system.
- **Supports atomic install:** Use the **atomic install --system** command to set up the etcd service to run as a systemd service.
- **Configures the etcd service:** When the etcd service starts, a set of ETCD environment variables are exported. Those variables identify the location of the etcd data directory and set the IP addresses and ports the etcd service listens on.
- **System container:** After you have used the **atomic** command to install the etcd container, you can use the systemd **systemctl** command to manage the service.

### 6.1.2. Getting and Running the etcd System Container

To use an etcd system container image on a RHEL Atomic system, you need to pull it, install it and enable it. The identity of the currently supported etcd container is:

```
registry.access.redhat.com/rhel7/etcd
```

The procedure below illustrates how to pull, install, and run the etcd container.

1. **Pull the etcd container:** While logged into the RHEL Atomic system, get the etcd container by running the following command:

```
# atomic pull --storage=ostree registry.access.redhat.com/rhel7/etcd
Image rhel7/etcd is being pulled to ostree ...
Pulling layer
2bf01635e2a0f7ed3800c8cb3effc5ff46adc6b9b86f0e80743c956371efe553
Pulling layer
38bd6ce6e1f2271d48ecb41a70a86122060ea91871a154b37d54ec66f593706f
Pulling layer
852368668be3e36086ae7a47c8b9e40b5ca87819b3200bc83d7a2f95b73f0f12
Pulling layer
e5d06327f2054d371f725243b619d66982c8d4589c1caa19bfcc23a93cf6b4d2
Pulling layer
82e7326c732857423e13163ff1e41ad63b3e2bddef8809175f89dec25f58b6ee
Pulling layer
b65a93c9f67115dc4c9da8dfeee63b58ec52c6ea58ff7f727b00d932d1f4e8f5
```

This pulls the etcd system container from the Red Hat Registry to the ostree storage area on the local system. By setting ostree storage, the docker storage area is not used and the docker daemon and docker command won't see the pulled etcd container image.

2. **Install the etcd container:** Type the following to do a default installation of the etcd container so it is set up as a systemd service.



#### NOTE

Before running **atomic install**, refer to "Configuring etcd" to see options you could add to the **atomic install** command to change it from the default install shown here.

```
# atomic install --system rhel7/etcd
Extracting to /var/lib/containers/atomic/etcd.0
```

```
systemctl daemon-reload
systemd-tmpfiles --create /etc/tmpfiles.d/etcd.conf
systemctl enable etcd
```

3. **Start the etcd service:** Use the **systemctl** command to start the installed etcd service as you would any other systemd service.

```
# systemctl start etcd
```

4. **Check etcd with runc:** To make sure the etcd container is running, you can use the **runc list** command as you would use **docker ps** to see containers running under docker:

```
# runc list
ID          PID      STATUS  BUNDLE          CREATED
etcd       4521    running /sysroot/ostree/deploy... 2016-10-
25T22:58:13.756410403Z
```

5. **Test that the etcd service is working:** You can use the **curl** command to set and retrieve keys from your etcd service. This example assigns a value to a key called **testkey**, then retrieves that value:

```
# curl -L http://127.0.0.1:2379/v2/keys/testkey -XPUT -d
value="testing my etcd"
{"action":"set","node":{"key":"/testkey","value":"testing my
etcd","modifiedIndex":6,"createdIndex":6}}
# curl -L http://127.0.0.1:2379/v2/keys/testkey
{"action":"get","node":{"key":"/testkey","value":"testing my
etcd","modifiedIndex":6,"createdIndex":6}}
```

Note that the first action does a **set** to set the key and the second does a **get** to return the value of the key.

The "Configuring etcd" section shows ways of setting up the etcd service in different ways.

### 6.1.3. Configuring etcd

You can change how the etcd service is configured on the **atomic install** command line or after it is running using the **runc** command.

#### 6.1.3.1. Configuring etcd during "atomic install"

The correct way to configure the etcd container image is when you first run **atomic install**. Setting that are defined initially in the **/etc/etcd/etcd.conf** file inside of the container can be overridden on the **atomic install** command line using the **--set** option. For example, this example shows how to reset the value of **ETCD\_ADVERTISE\_CLIENT\_URLS** value:

```
# atomic install --system --set
ETCD_ADVERTISE_CLIENT_URLS="http://192.168.122.55:2379" rhel/etcd
```

Here is the list of other values and setting in the **etcd.conf** file that you can change on the **atomic install** command line. See the [etcd.conf.yaml.sample](#) page for descriptions of these settings.

```
# [member]
ETCD_NAME=default
```

```

ETCD_DATA_DIR="/var/lib/etcd/default.etcd"
#ETCD_WAL_DIR=""
#ETCD_SNAPSHOT_COUNT="10000"
#ETCD_HEARTBEAT_INTERVAL="100"
#ETCD_ELECTION_TIMEOUT="1000"
#ETCD_LISTEN_PEER_URLS="http://localhost:2380"
ETCD_LISTEN_CLIENT_URLS="http://localhost:2379"
#ETCD_MAX_SNAPSHOTS="5"
#ETCD_MAX_WALS="5"
#ETCD_CORS=""
#[cluster]
#ETCD_INITIAL_ADVERTISE_PEER_URLS="http://localhost:2380"
# if you use different ETCD_NAME (e.g. test), set ETCD_INITIAL_CLUSTER
value for this name, i.e. "test=http://..."
#ETCD_INITIAL_CLUSTER="default=http://localhost:2380"
#ETCD_INITIAL_CLUSTER_STATE="new"
#ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster"
ETCD_ADVERTISE_CLIENT_URLS="http://localhost:2379"
#ETCD_DISCOVERY=""
#ETCD_DISCOVERY_SRV=""
#ETCD_DISCOVERY_FALLBACK="proxy"
#ETCD_DISCOVERY_PROXY=""
#ETCD_STRICT_RECONFIG_CHECK="false"
#[proxy]
#ETCD_PROXY="off"
#ETCD_PROXY_FAILURE_WAIT="5000"
#ETCD_PROXY_REFRESH_INTERVAL="30000"
#ETCD_PROXY_DIAL_TIMEOUT="1000"
#ETCD_PROXY_WRITE_TIMEOUT="5000"
#ETCD_PROXY_READ_TIMEOUT="0"
#[security]
#ETCD_CERT_FILE=""
#ETCD_KEY_FILE=""
#ETCD_CLIENT_CERT_AUTH="false"
#ETCD_TRUSTED_CA_FILE=""
#ETCD_PEER_CERT_FILE=""
#ETCD_PEER_KEY_FILE=""
#ETCD_PEER_CLIENT_CERT_AUTH="false"
#ETCD_PEER_TRUSTED_CA_FILE=""
#[logging]
#ETCD_DEBUG="false"
# examples for -log-package-levels etcdserver=WARNING,security=DEBUG
#ETCD_LOG_PACKAGE_LEVELS=""
#[profiling]
#ETCD_ENABLE_PPROF="false"

```

### 6.1.3.2. Configuring etcd security settings

The etcd service is configured with authentication and encryption disabled by default. Because etcd is initially configured to listen to localhost only, the lack of security becomes much more of an issue when the etcd service is exposed to nodes that are outside of the local host. Remote attackers will have access to passwords and secret keys.

In general, here is what you need to do to configure a secure, multi-node etcd cluster service:

1. Create TLS certificates and a signed key pair for every member in a cluster, as described in [The etcd Security Model](#).
2. Identify the certificates and keys in the `/etc/etcd/etcd.conf` file.
3. Open the firewall to allow access to TCP ports 7379 (client communication) and 7380 (server-to-server communication).
4. Install and run the etcd service (see `atomic install --system rhel7/etcd` as described earlier)

### 6.1.3.3. Configuring etcd with "runc"

With the etcd container running, you can configure settings in the etcd container using the `runc exec` command. For example, you could run the `etcdctl` command inside the etcd container to change the network range set by the `Network` value in the etcd keystore (used later by the flannel service) with the following command:

```
# runc exec etcd etcdctl set /atomic.io/network/config
'{"Network":"10.40.0.0/16"}'
# runc exec etcd etcdctl get /atomic.io/network/config
{"Network":"10.40.0.0/16"}
```

The example just shown illustrates the `runc exec` command running `etcdctl set` at first to set the `Network` value. After that, `runc` executes the `etcdctl get` command to get configuration information.

### 6.1.4. Tips for Running etcd Container

If you are done with the etcd container image, you can remove it with the `atomic uninstall` command:

```
# atomic uninstall etcd
```

For more information on system containers, see [Introduction to System Containers](#).

## 6.2. USING THE FLANNEL SYSTEM CONTAINER IMAGE

### 6.2.1. Overview

The `flannel` service was designed to provide virtual subnets for use among container hosts. Using flannel, Kubernetes (or other container platforms) can ensure that each container pod has a unique address that is routable within a Kubernetes cluster. As a result, the job of finding ports and services between containers is simpler.

The flannel container described here is what is referred to as a system container. A system container is designed to come up before the docker service or in a situation where no docker service is available. In this case, the flannel container is meant to be brought up after the etcd service (also available as a system container) and before docker and kubernetes services to provide virtual subnets that the later services can leverage.

Besides bypassing the docker service, the flannel container can also bypass the docker command and the storage area used to hold docker containers by default. To use the container, you need a combination of commands that include `atomic` (to pull, list, install, delete and unstage the image), `skopeo` (to inspect the image), `runc` (to ultimately run the image) and `systemctl` to manage the image among your other systemd services.



## NOTE

For RHEL 7.3, system containers in general and the flannel container specifically are supported as Tech Preview only.

Here are some of the features of the flannel container:

- **Supports atomic pull:** Use the **atomic pull --storage=ostree** command to pull the container to the ostree storage area, instead of default docker storage, on your system.
- **Supports atomic install:** Use the **atomic install --system** command to set up the flannel service to run as a systemd service.
- **Configures the flannel service:** When the flannel service starts, configuration data are stored for flannel in the etcd keystore. To configure flannel, you can use the **runc** command to run an etcdctl command to configure flannel settings inside the etcd container.
- **System container:** After you have used the **atomic** command to install the flannel container, you can use the systemd **systemctl** command to manage the service.

### 6.2.2. Getting and Running the RHEL flannel System Container

To use the flannel system container image on a RHEL system, you need to pull it, install it and enable it, as described in the following procedure:

1. **Pull and run the etcd container:** The flannel container is dependent on there being an available etcd keystore. See [Using the etcd System Container Image](#) for information on pulling, installing, and running the etcd system container before setting up the flannel system container.
2. **Pull the flannel container.** While logged into the RHEL system, get the RHEL etcd container by running the following command:

```
# atomic pull --storage=ostree rhel7/flannel
Image rhel7/flannel is being pulled to ostree ...
Pulling layer
2bf01635e2a0f7ed3800c8cb3effc5ff46adc6b9b86f0e80743c956371efe553
Pulling layer
38bd6ce6e1f2271d48ecb41a70a86122060ea91871a154b37d54ec66f593706f
...
```

This pulls the flannel system container from the Red Hat registry to the ostree storage area on the local system. By setting ostree storage, the docker storage area is not used and the docker daemon and docker command won't see the pulled flannel container image.

3. **Install the flannel container.** Type the following to do a default installation of the flannel container so it is set up as a systemd service. See "Configuring flannel" to see options you could add to the **atomic install** command to change it from the default install shown here.

```
# atomic install --system rhel7/flannel
Extracting to /var/lib/containers/atomic/flannel.0
systemctl daemon-reload
systemd-tmpfiles --create /etc/tmpfiles.d/flannel.conf
systemctl enable flannel
```

4. **Start the flannel service:** Use the **systemctl** command to start the installed etcd service as you would any other systemd service.

```
# systemctl start flannel
```

5. **Check etcd and flannel with runc:** To make sure the flannel and etcd containers are running, you can use the **runc list** command as you would use **docker ps** to see containers running under docker:

```
# runc list
ID          PID      STATUS  BUNDLE          CREATED
etcd       4521    running /sysroot/ostree/deplo... 2016-10-25T22:58:13.756410403Z
flannel    6562    running /sysroot/ostree/deplo... 2016-10-26T13:50:49.041148994Z
```

6. **Test that the flannel service is working:** If the flannel service is working properly, the next time you start up the docker0 network interface, the docker network interface should pick up an address range from those assigned by flannel. After starting flannel and before restarting docker, run these commands:

```
# ip a | grep docker | grep inet
    inet 172.17.0.1/16 scope global docker0
# systemctl reboot
# ip a | grep docker | grep inet
    inet 10.40.4.1/24 scope global docker0
```

Note that the docker0 interface picks up an address in the address range assigned by flannel and will, going forward, assign containers to addresses in the 10.40.4.0/24 address range.

The "Configuring flannel" section shows ways of setting up the etcd service in different ways.

### 6.2.3. Configuring flannel

You can change how the flannel service is configured on the **atomic install** command line or after it is running using the **runc** command.

#### 6.2.3.1. Configuring etcd during "atomic install"

Environment variables that are defined initially when the flannel container starts up can be overridden on the **atomic install** command line using the **--set** option. For example, this example shows how to reset the value of **FLANNELD\_ETCD\_ENDPOINTS**:

```
# atomic install --system --set
FLANNELD_ETCD_ENDPOINTS="http://192.168.122.55:2379" rhel7/flannel
```

This is how two of these variables are set by default:

- **FLANNELD\_ETCD\_ENDPOINTS=http://127.0.0.1:2379:** Identifies the location of the etcd service IP address and port number.
- **FLANNELD\_ETCD\_PREFIX=/atomic.io/network:** Identifies the location of flannel values in the etcd keystore.

Here is the list of other values that you can change on the **atomic install** command line. See the Key Command Line Options and Environment Variables sections of the [Flannel Github](#) page for descriptions of these settings.

```
* *FLANNELD_PUBLIC_IP*
* *FLANNELD_ETCD_ENDPOINTS*
* *FLANNELD_ETCD_PREFIX*
* *FLANNELD_ETCD_KEYFILE*
* *FLANNELD_ETCD_CERTFILE*
* *FLANNELD_ETCD_CAFILE*
* *FLANNELD_IFACE*
* *FLANNELD_SUBNET_FILE*
* *FLANNELD_IP_MASQ*
* *FLANNELD_LISTEN*
* *FLANNELD_REMOTE*
* *FLANNELD_REMOTE_KEYFILE*
* *FLANNELD_REMOTE_CERTFILE*
* *FLANNELD_REMOTE_CAFILE*
* *FLANNELD_NETWORKS*
```

### 6.2.3.2. Configuring flannel with "runc"

Flannel settings that are stored in the etcd keystore can be changed by executing **etcdctl** commands in the etcd container. Here's an example of how to change the Network value in the etcd keystore so that flannel uses a different set of IP address ranges.

```
# runc exec etcd etcdctl set /atomic.io/network/config
'{"Network":"10.40.0.0/16"}'
# runc exec etcd etcdctl get /atomic.io/network/config
{"Network":"10.40.0.0/16"}
```

The example just shown illustrates the **runc exec** command running **etcdctl set** at first to set the Network value. After that, runc executes the **etcdctl get** command to get configuration information.

### 6.2.4. Tips for Running flannel Container

If you are done with the flannel container image, you can remove it with the **atomic uninstall** command:

```
# atomic uninstall flannel
```

For more information on system containers, see [Introduction to System Containers](#).

## 6.3. USING THE OVIRT-GUEST-AGENT SYSTEM CONTAINER IMAGE FOR RED HAT VIRTUALIZATION

### 6.3.1. Overview

The ovirt-guest-agent container launches the Red Hat Virtualization (RHV) management agent. This container is made to be deployed on Red Hat Enterprise Linux virtual machines that are running in a RHV environment. The agent provides an interface to the RHV manager that supplies heart-beat and other run-time data from inside the guest VM. The RHV manager can send control commands to shutdown, restart and otherwise change the state of the virtual machine through the agent.



The `ovirt-guest-agent` is added automatically to the Red Hat Atomic Image for RHV, which is an OVA-formatted image made for RHEV environments. You can download the image from the [Red Hat Enterprise Linux Atomic Host download page](#). Or, you can get and run the container image manually on a RHEL Server or RHEL Atomic Host virtual machine you install yourself.

The `ovirt-guest-agent` container is a system container. System containers are designed to come up before the `docker` service or in a situation where no `docker` service is available. In this case, the `ovirt-guest-agent` allows the RHV manager to change the state of the virtual machine on which it is running whether the `docker` service is running or not.

Here are some of the features of the `ovirt-guest-agent` container:

- **Supports atomic pull:** Use the **atomic pull** command to pull the `ovirt-guest-agent` container to your system.
- **Supports atomic install:** Use the **atomic install --system** command to set up the `ovirt-guest-agent` service to run as a `systemd` service.
- **System container:** After you have used the **atomic** command to install the `ovirt-guest-agent` container, you can use the `systemd` **systemctl** command to manage the service.

Note that the `ovirt-guest-agent` container image is not made to run in environments other than a RHEL or RHEL Atomic virtual machine in a RHV environment.

### 6.3.2. Getting and Running the `ovirt-guest-agent` System Container

To use an `ovirt-guest-agent` system container image on a RHEL Server or RHEL Atomic system, you need to pull it, install it and enable it. The identity of the currently supported `ovirt-guest-agent` container is:

```
registry.access.redhat.com/rhev4/ovirt-guest-agent
```

The procedure below illustrates how to pull, install, and run the `ovirt-guest-agent` container.

1. **Pull the `ovirt-guest-agent` container.** While logged into the RHEL or RHEL Atomic system, get the `ovirt-guest-agent` container by running the following command:

```
# atomic pull --storage=ostree
registry.access.redhat.com/rhev4/ovirt-guest-agent
```

This pulls the `ovirt-guest-agent` system container from the Red Hat Registry to the `ostree` storage area on the local system. By setting `ostree` storage, the `docker` storage area is not used and the `docker` daemon and `docker` command won't see the pulled `ovirt-guest-agent` container image.

2. **Install the `ovirt-guest-agent` container.** Type the following to do a default installation of the `ovirt-guest-agent` container so it is set up as a `systemd` service.

```
# atomic install --system rhel7/ovirt-guest-agent
Extracting to /var/lib/containers/atomic/ovirt-guest-agent.0
systemctl daemon-reload
systemd-tmpfiles --create /etc/tmpfiles.d/ovirt-guest-agent.conf
systemctl enable ovirt-guest-agent
```

3. **Start the `ovirt-guest-agent` service:** Use the **systemctl** command to start and enable the installed `ovirt-guest-agent` service as you would any other `systemd` service.

```
# systemctl start ovirt-guest-agent
# systemctl enable ovirt-guest-agent
```

4. **Check ovirt-guest-agent with runc:** To make sure the ovirt-guest-agent container is running, you can use the **runc list** command as you would use **docker ps** to see containers running under docker:

```
# runc list
ID                PID      STATUS  BUNDLE
CREATED
ovirt-guest-agent 4521    running /sysroot/ostree/de... 2017-04-
07T21:01:07.279104535Z
```

### 6.3.3. Removing the ovirt-guest-agent Container and Image

If you are done with the ovirt-guest-agent container image, you can stop and remove the container, then uninstall the image:

```
# atomic containers delete ovirt-guest-agent
Do you wish to delete the following images?

ID                NAME                IMAGE_NAME                STORAGE
ovirt-guest-      ovirt-guest-agent   registry.access.redhat.com ostree

Confirm (y/N) y
systemctl stop ovirt-guest-agent
systemctl disable ovirt-guest-agent
systemd-tmpfiles --remove /etc/tmpfiles.d/ovirt-guest-agent.conf
# atomic uninstall registry.access.redhat.com/rhev4/ovirt-guest-agent
Do you wish to delete the following images?

IMAGE                STORAGE
registry.access.redhat.com/rhev4/ovirt-guest-agent  ostree
Confirm (y/N) y
```

For more information on system containers, see [Introduction to System Containers](#).

## 6.4. USING THE OPEN-VM-TOOLS SYSTEM CONTAINER IMAGE FOR VMWARE

### 6.4.1. Overview

The open-vm-tools container provides services and modules that allow VMware technology to manage and otherwise work with Red Hat Enterprise Linux and RHEL Atomic Host virtual machines running in VMware environments. Kernel modules included in this container are made to improve performance of RHEL systems running as VMware guests. Services provided by this container include:

- Graceful power operations
- Script execution on guests during power operations
- Enhanced guest automation via custom programs or file system operations

- Guest authentication
- Guest network, memory, and disk usage information collection
- Guest heartbeat generation, used to determine if guests are available
- Guest, host, and client desktop clock synchronization
- Host access to obtain file-system-consistent guest file system snapshots
- Guest script execution associated with quiescing guest file systems (pre-freeze and post-thaw)
- Guest customization opportunities after guests power up
- File folder sharing between VMware (Workstation or Fusion) and guest system
- Text, graphics, and file pasting between guests, hosts and client desktops

The open-vm-tools container is a system container, designed to come up before the docker service or in a situation where no docker service is available. In this case, the open-vm-tools container allows VMware technologies to manage the RHEL or RHEL Atomic virtual machines on which it is running whether the docker service is running or not.

Here are some of the features of the open-vm-tools container on the RHEL guest system:

- **Supports atomic pull:** Use the **atomic pull** command to pull the open-vm-tools container to your system.
- **Supports atomic install:** Use the **atomic install --system** command to set up the open-vm-tools service to run as a systemd service.
- **System container:** After you have used the **atomic** command to install the open-vm-tools container, you can use the systemd **systemctl** command to manage the service.

Note that the open-vm-tools container image is not made to run in environments other than a RHEL or RHEL Atomic virtual machine in a VMware environment.

## 6.4.2. Getting and Running the open-vm-tools System Container

To use an open-vm-tools system container image on a RHEL Server or RHEL Atomic system, you need to pull it, install it and enable it. The identity of the currently supported open-vm-tools container is:

```
registry.access.redhat.com/rhel7/open-vm-tools
```

The procedure below illustrates how to pull, install, and run the open-vm-tools container.

1. **Pull the open-vm-tools container:** While logged into the RHEL or RHEL Atomic system, get the open-vm-tools container by running the following command:

```
# atomic pull --storage=ostree
registry.access.redhat.com/rhel7/open-vm-tools
```

This pulls the open-vm-tools system container from the Red Hat Registry to the ostree storage area on the local system. By setting ostree storage, the docker storage area is not used and the docker daemon and docker command won't see the pulled open-vm-tools container image.

2. **Install the open-vm-tools container:** Type the following to do a default installation of the open-vm-tools container so it is set up as a systemd service.

```
# atomic install --system rhel7/open-vm-tools
Extracting to /var/lib/containers/atomic/open-vm-tools.0
systemctl daemon-reload
systemd-tmpfiles --create /etc/tmpfiles.d/open-vm-tools.conf
systemctl enable open-vm-tools
```

3. **Start the open-vm-tools service:** Use the **systemctl** command to start and enable the installed open-vm-tools service as you would any other systemd service.

```
# systemctl start open-vm-tools
# systemctl enable open-vm-tools
```

4. **Check open-vm-tools with runc:** To make sure the open-vm-tools container is running, you can use the **runc list** command as you would use **docker ps** to see containers running under docker:

```
# runc list
ID                PID      STATUS  BUNDLE                CREATED
open-vm-tools    4521    running /sysroot/ostree/de... 2017-04-
07T18:03:01.913246491Z
```

### 6.4.3. Removing the open-vm-tools Container and Image

If you are done with the open-vm-tools container image, you can stop and remove the container, then uninstall the image:

```
# atomic containers delete open-vm-tools
Do you wish to delete the following images?

ID                NAME                IMAGE_NAME                STORAGE
ovirt-guest-    open-vm-tools    registry.access.redhat.com ostree

Confirm (y/N) y
systemctl stop open-vm-tools
systemctl disable open-vm-tools
systemd-tmpfiles --remove /etc/tmpfiles.d/open-vm-tools.conf
# atomic uninstall registry.access.redhat.com/rhel7/open-vm-tools
Do you wish to delete the following images?

IMAGE                STORAGE
registry.access.redhat.com/rhel7/open-vm-tools    ostree
Confirm (y/N) y
```

To learn more about how the open-vm-tools container was built, refer to [Containerizing open-vm-tools](#). Using the instructions in that article allows you to build your own open-vm-tools container, using custom configuration settings. For more information on system containers, see [Introduction to System Containers](#).

