Red Hat Enterprise Linux Atomic Host 7
Getting Started with Containers

Getting Started with Containers

Red Hat Atomic Host Documentation Team

Getting Started with Containers

Legal Notice

Abstract

Containers and Container Development

Table of Contents

CHAPTER 1. INTRODUCTION TO LINUX CONTAINERS

1.1. OVERVIEW

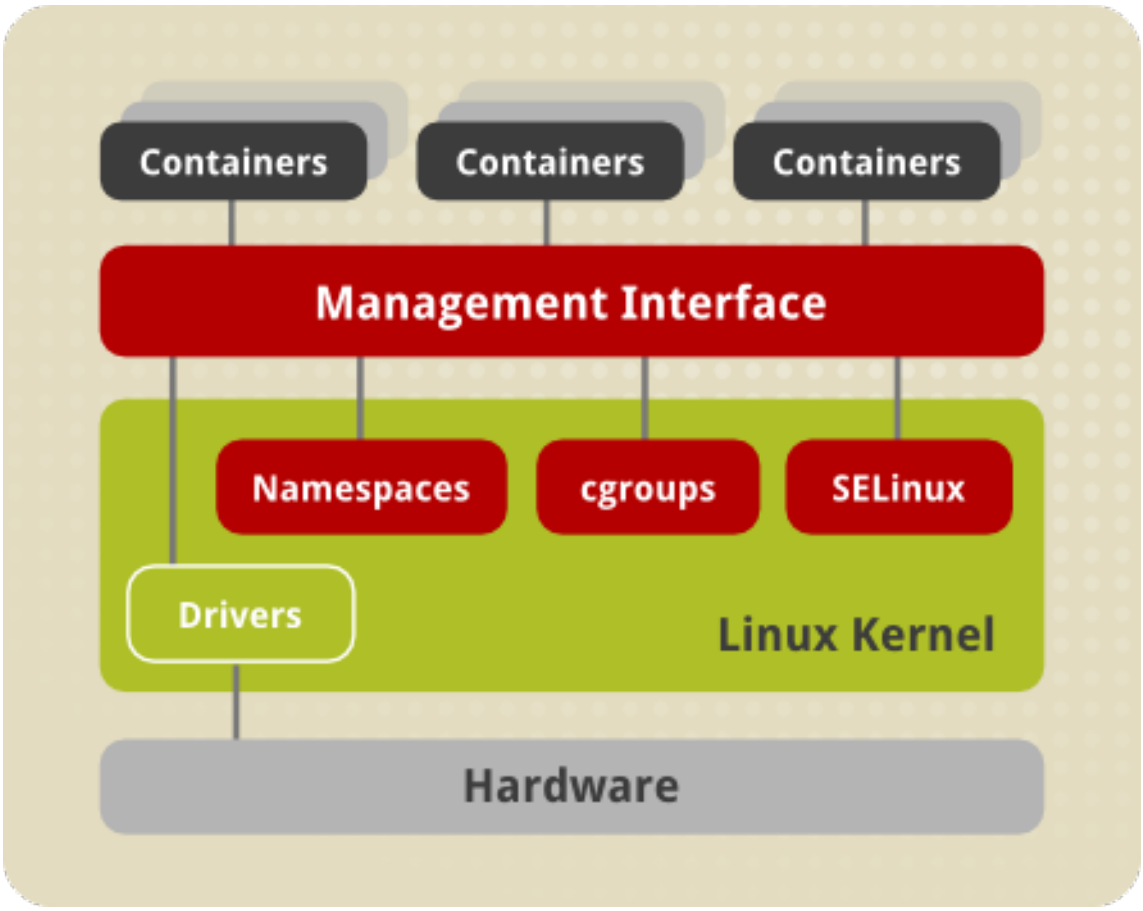
Linux Containers have emerged as a key open source application packaging and delivery technology, combining lightweight application isolation with the flexibility of image-based deployment methods.

Red Hat Enterprise Linux 7 implements Linux Containers using core technologies such as Control Groups (Cgroups) for Resource Management, Namespaces for Process Isolation, SELinux for Security, enabling secure multi-tenancy and reducing the potential for security exploits

1.2. LINUX CONTAINERS ARCHITECTURE

Several components are needed for Linux Containers to function correctly, most of them are provided by the Linux kernel. Kernel *namespaces* ensure process isolation and *cgroups* are employed to control the system resources. *SELinux* is used to assure separation between the host and the container and also between the individual containers. *Management interface* forms a higher layer that interacts with the aforementioned kernel components and provides tools for construction and management of containers.

The following scheme illustrates the architecture of Linux Containers in Red Hat Enterprise Linux 7:



Namespaces

The kernel provides process isolation by creating separate **namespaces** for containers. Namespaces enable creating an abstraction of a particular global system resource and make it appear as a separated instance to processes within a namespace. Consequently, several containers

can use the same resource simultaneously without creating a conflict. There are several types of namespaces:

- **Mount namespaces** isolate the set of file system mount points seen by a group of processes so that processes in different mount namespaces can have different views of the file system hierarchy. With mount namespaces, the mount() and umount() system calls cease to operate on a global set of mount points (visible to all processes) and instead perform operations that affect just the mount namespace associated with the container process. For example, each container can have its own **`/tmp`** or **`/var`** directory or even have an entirely different userspace.

- **UTS namespaces** isolate two system identifiers – nodename and domainname, returned by the uname() system call. This allows each container to have its own hostname and NIS domain name, which is useful for initialization and configuration scripts based on these names. You can test this isolation by executing the **`hostname`** command on the host system and a container – the results will differ.

- **IPC namespaces** isolate certain interprocess communication (IPC) resources, such as System V IPC objects and POSIX message queues. This means that two containers can create shared memory segments and semaphores with the same name, but are not able to interact with other containers memory segments or shared memory.

- **PID namespaces** allow processes in different containers to have the same PID, so each container can have its own init (PID1) process that manages various system initialization tasks as well as containers life cycle. Also, each container has its unique **`/proc`** directory. Note that from within the container you can monitor only processes running inside this container. In other words, the container is only aware of its native processes and can not "see" the processes running in different parts of the system. On the other hand, the host operating system is aware of processes running inside of the container, but assigns them different PID numbers. For example, run the **`ps -eZ | grep systemd$`** command on host to see all instances of systemd including those running inside of containers.

- **Network namespaces** provide isolation of network controllers, system resources associated with networking, firewall and routing tables. This allows container to use separate virtual network stack, loopback device and process space. You can add virtual or real devices to the container, assign them their own IP Addresses and even full iptables rules. You can view the different network settings by executing the **`ip addr`** command on the host and inside the container.

> **Note**
>
> There is another type of namespace called **user namespace**. User namespaces are similar to PID namespaces, they allow you to specify a range of host UIDs dedicated to the container. Consequently, a process can have full root privileges for operations inside the container, and at the same time be unprivileged for operations outside the container. For compatibility reasons, user namespaces are turned off in the current version of Red Hat Enterprise Linux 7, but will be enabled in the near future.

Control Groups (cgroups)

The kernel uses **cgroups** to group processes for the purpose of system resource management. Cgroups allocate CPU time, system memory, network bandwidth, or combinations of these among user-defined groups of tasks. In Red Hat Enterprise Linux 7, cgroups are managed with systemd slice, scope, and service units. For more information on cgroups, see the Red Hat Enterprise Linux 7 Resource Management Guide.

SELinux

SELinux provides secure separation of containers by applying SELinux policy and labels. It

integrates with virtual devices by using the **sVirt** technology. For more information see the Red Hat Enterprise Linux 7 SELinux Users and Administrators Guide.

Management Interface

1.3. SECURE CONTAINERS WITH SELINUX

From the security point of view, there is a need to isolate the host system from a container and to isolate containers from each other. The kernel features used by containers, namely cgoups and namespaces, by themselves provide a certain level of security. Cgroups ensure that a single container cannot exhaust a large amount of system resources, thus preventing some denial-of-service attacks. By virtue of namespaces, the **`/dev`** directory created within a container is private to each container, and therefore unaffected by the host changes. However, this can not prevent a hostile process from breaking out of the container since the entire system is not namespaced or containerized. Another level of separation, provided by SELinux, is therefore needed.

Security-Enhanced Linux (**SELinux**) is an implementation of a mandatory access control (MAC) mechanism, multi-level security (MLS), and multi-category security (MCS) in the Linux kernel. The **sVirt** project builds upon SELinux and integrates with Libvirt to provide a MAC framework for virtual machines and containers. This architecture provides a secure separation for containers as it prevents root processes within the container from interfering with other processes running outside this container. The containers created with Docker are automatically assigned with an SELinux context specified in the SELinux policy.

By default, containers created with libvirt tools are assigned with the **`virtd_lxc_t`** label (execute **`ps -eZ | grep virtd_lxc_t`**). You can apply sVirt by setting static or dynamic labeling for processes inside the container.

> **Note**
>
> You might notice that SELinux appears to be disabled inside the container even though it is running in enforcing mode on host system – you can verify this by executing the **`getenforce`** command on host and in the container. This is to prevent utilities that have SELinux awareness, such as **`setenforce`**, to perform any SELinux activity inside the container.

Note that if SELinux is disabled or running in permissive mode on the host machine, containers are not separated securely enough. For more information about SELinux, refer to Red Hat Enterprise Linux 7 SELinux Users and Administrators Guide, sVirt is described in Red Hat Enterprise Linux 7 Virtualization Security Guide.

> **Note**
>
> Note that currently it is not possible to run containers with SELinux enabled on the B-tree file system (Btrfs). Therefore, to use Docker with SELinux enabled, which is highly recommended, make sure the **`/var/lib/docker`** is not placed on Brtfs. In case you need to run Docker on Brtfs, disable SElinux by removing the *--selinux-enabled* entry from the **`/lib/systemd/system/docker.service`** configuration file.

1.4. CONTAINER USE CASES

1.4.1. Image-based Containers

Image-based containers package applications with individual runtime stacks, making the resultant containers independent from the host operating system. This makes it possible to run several instances of an application, each developed for a different platform. This is possible because the container run time and the application run time are deployed in the form of an image. For example, Runtime A in ? can stand for Red Hat Enterprise Linux 6.5, Runtime B could refer to version 6.6 and so on.

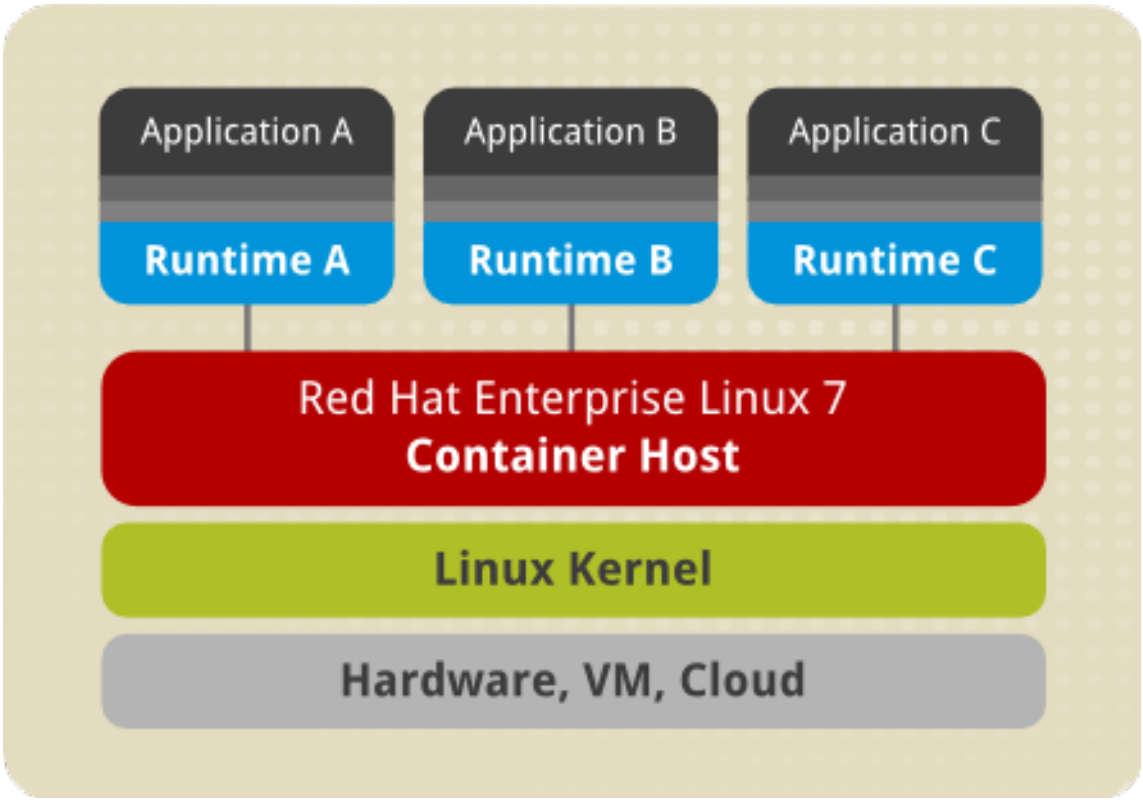
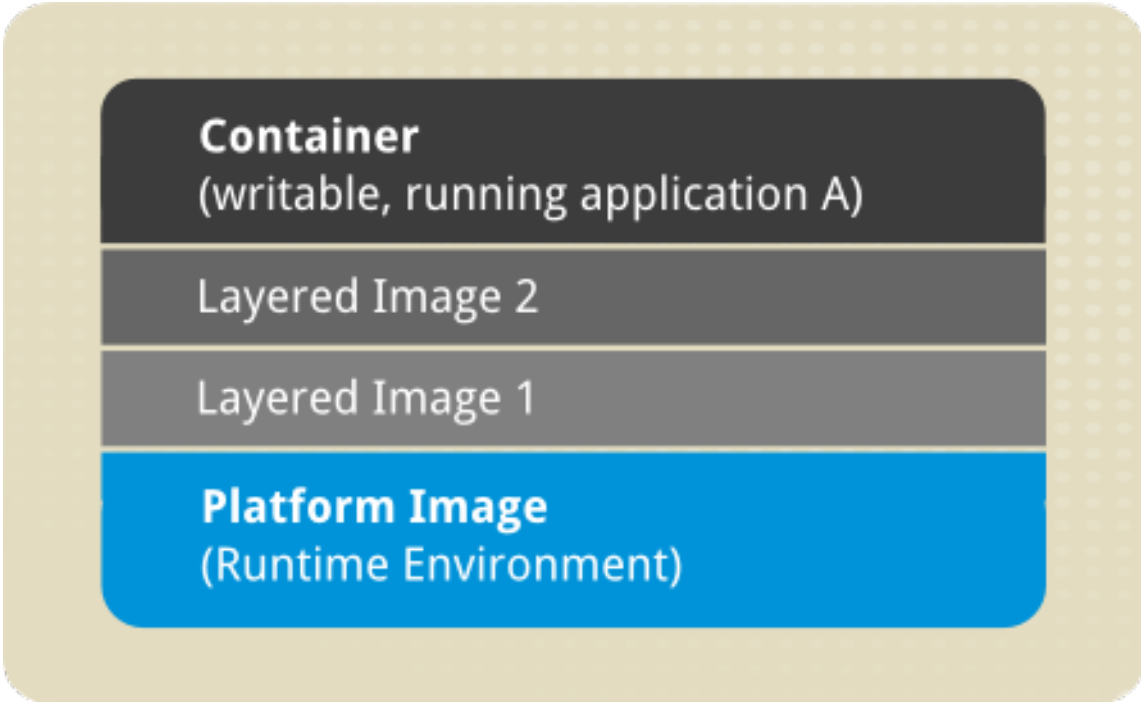


Image-based containers allow you to host multiple instances and versions of an application, with minimal overhead and increased flexibility. Such containers are not tied to the host-specific configuration, which makes them portable.

Docker format relies on the *device mapper thin provisioning* technology that is an advanced variation of LVM snapshots to implement copy-on-write in Red Hat Enterprise Linux 7.



The above image shows the fundamental components of any image-based container:

- **Container** – (in the narrow sense of the word) an active component in which an application runs. Each container is based on an *image* that holds necessary configuration data. When you launch a container from an image, a writable layer is added on top of this image. Every time you commit a container (using the **`docker commit`** command), a new image layer is added to store your changes.
- **Image** – a static snapshot of the containers' configuration. Image is a read-only layer that is never modified, all changes are made in top-most writable layer, and can be saved only by creating a new image. Each image depends on one or more parent images.
- **Platform Image** – an image that has no parent. Platform images define the runtime environment, packages and utilities necessary for a containerized application to run. The work with Docker usually starts by pulling the platform image. The platform image is read-only, so any changes are reflected in the copied images stacked on top of it. Red Hat provides a platform image for Red Hat Enterprise Linux 7 as well as Red Hat Enterprise Linux 6.

The images piled on top of the platform image create an *application layer* that contains software dependencies for the containerized application. For example, the layered images in ? could have added software dependencies required by the containerized application.

The whole container can be very large or it could be made really small depending on how many packages are included in the application layer. Further layering of the images is possible, such as software from 3rd party independent software vendors. From a user point of view there is still one container, but from an operational point of view there can be many layers behind it.

1.5. LINUX CONTAINERS COMPARED TO KVM VIRTUALIZATION

KVM virtual machines require a kernel of their own. Linux containers share the kernel of the host operating system. It is usually possible to launch a much larger number of containers than virtual machines on the same hardware.

Both Linux Containers and KVM virtualization have certain advantages and drawbacks that influence the use cases in which these technologies are typically applied:

KVM virtualization:

- KVM virtualization enables booting full operating systems of different kinds, even non-Linux systems. The resource-hungry nature of virtual machines (as compared to containers) means that the number of virtual machines that can be run on a host is lower than the number of containers that can be run on the same host.
- Running separate kernel instances generally provides separation and security. The unexpected termination of one of the kernels does not disable the whole system.
- Guest virtual machine is isolated from host changes, which allows to run different versions of the same application on the host and virtual machine. KVM also provides many useful features such as live migration. For more information on these capabilities, see Red Hat Enterprise Linux 7 Virtualization Deployment and Administration Guide.

Linux Containers:

- Linux Containers are designed to support isolation of one or more applications.
- System-wide changes are visible in each container. For example, if you upgrade an application on the host machine, this change will apply to all sandboxes that run instances of this application.

- Since containers are lightweight, a large number of them can run simultaneously on a host machine. The theoretical maximum is 6000 containers and 12,000 bind mounts of root file system directories.

1.6. ADDITIONAL RESOURCES

To learn more about general principles and architecture of Linux Containers, refer to the following resources.

Installed Documentation

- docker(1) — The manual page of the **`docker`** command.

Online Documentation

- Red Hat Enterprise Linux 7 Virtualization Deployment and Administration Guide — this topic instructs how to configure a Red Hat Enterprise Linux 7 host physical machine and how to install and configure guest virtual machines with different distributions, using the KVM hypervisor. Also included PCI device configuration, SR-IOV, networking, storage, device and guest virtual machine management, as well as troubleshooting, compatibility and restrictions.
- Red Hat Enterprise Linux 7 SELinux Users and Administrators Guide — The SELinux Users and Administrators Guide for Red Hat Enterprise Linux 7 documents the basics and principles upon which SELinux functions, as well as practical tasks to set up and configure various services.
- Get Started with Docker Formatted Container Images on Red Hat Systems — This quick start guide describes the essential Docker-related tasks along with a number of examples.
- Documentation on the Docker Project Site — The official documentation of the Docker project.

CHAPTER 2. GET STARTED ORCHESTRATING CONTAINERS WITH KUBERNETES

2.1. OVERVIEW

Kubernetes is a tool for orchestrating and managing Docker containers. This procedure lets you set up a single-system Kubernetes sandbox so you can try out:

- Using two containers and yaml files to deploy those containers with Kubernetes.
- Managing containers with Kubernetes.

This procedure results in a setup that provides a sandbox in which you can begin trying out Kubernetes and exploring how it works. In this procedure, services that are typically on a separate Kubernetes master system and two or more Kubernetes node systems are all running on a single system.

2.2. UNDERSTANDING KUBERNETES

While Docker defines the container format and builds and manages individual containers, an orchestration tool is needed to deploy and manage sets of containers. Kubernetes is a tool designed to orchestrate Docker containers. After building the container images you want, you can use a Kubernetes Master to deploy one or more containers in what is referred to as a pod. The Master pushes the containers in that pod to a Kubernetes Node where the containers run.

For this example, both the Kubernetes Master and Node are on the same computer, which can be either a RHEL 7 Server or RHEL 7 Atomic host. Kubernetes relies on a set of service daemons to implement features of the Kubernetes Master and Node. You need to understand the following about Kubernetes Masters and Node:

- **Master**: A Kubernetes Master is where you direct API calls to services that control the activities of the pods, replications controllers, services, nodes and other components of a Kubernetes cluster. Typically, those calls are made by running **kubectl** commands. From the Master, containers are deployed to run on Nodes.
- **Node**: A Node is a system providing the run-time environments for the containers.

Pod definitions are stored in configuration files (in yaml or json formats). Using the following procedure, you will set up a single RHEL 7 or RHEL Atomic system, configure it as a Kubernetes Master and Node, use yaml files to define each container in a pod, and deploy those containers using Kubernetes (**kubectl** command).

2.3. RUNNING CONTAINERS FROM KUBERNETES PODS

You need a RHEL 7 or RHEL Atomic system to build the Docker containers and orchestrate them with Kubernetes. There are different sets of service daemons needed on Kubernetes Master and Node systems. In this procedure, both sets of service daemons run on the same system.

Once the containers, system and services are in place, you use the **kubectl** command to deploy those containers so they run on the Kubernetes Node (in this case, that will be the local system).

Here’s how to do those steps:

2.3.1. Setting up to Deploy Docker Containers with Kubernetes

To prepare for Kubernetes, you need to install RHEL 7 or RHEL Atomic, disable firewalld, and get two containers.

1. **Install a RHEL 7 or RHEL Atomic system**: For this Kubernetes sandbox system, install a RHEL 7 or RHEL Atomic system, subscribe the system, then install and start the docker service. Refer here for information on setting up a basic RHEL or RHEL Atomic system to use with Kubernetes:

   Get Started with Docker Formatted Container Images on Red Hat Systems

2. **Install Kubernetes**: If you are on a RHEL 7 system, install the kubernetes and etcd packages. This will pull in kubernetes-node and kubernetes-master packages as well (those packages are already installed on RHEL Atomic):

   ```
   # yum install kubernetes etcd
   ```

3. **Disable firewalld**: If you are using a RHEL 7 host, be sure that the firewalld service is disabled (the firwalld service is not installed on an Atomic host). On RHEL 7, type the following to disable and stop the firewalld service:

   ```
   # systemctl disable firewalld
   # systemctl stop firewalld
   ```

4. **Get Docker Containers**: Build the following two containers by following the instructions in the Getting Started Guide for RHEL Atomic Host, and make images of those containers available on both nodes (in other words, make sure you can see the two images when you type docker images on either node):

   - Simple Apache Web Server in a Docker Container
   - Simple Database Server in a Docker Container

   After you build and test the containers, stop them (**docker stop mydbforweb** and **docker stop mywebwithdb**).

2.3.2. Starting Kubernetes

Because both Kubernetes Master and Node services are running on the local system, you don't need to change the Kubernetes configuration files. Master and Node services will point to each other on localhost and services are made available only on localhost.

1. **Configure apiserver file**: By default, the admission control feature of the kube-apiserver service requires that an account be set up for anyone trying to start a pod. This allows Kubernetes providers to track and limit usage by user accounts. For this all-in-one example, you can disable that feature by opening the **/etc/kubernetes/apiserver** file and changing the KUBE_ADMISSION_CONTROL value to remove "ServiceAccount" from the --admission_control options. The following lines show the original line (commented out) and the new line with ServiceAccount removed (the two lines should wrap, but they may appear broken):

   ```
   # KUBE_ADMISSION_CONTROL="--
   admission_control=NamespaceLifecycle,NamespaceExists,LimitRanger,
   SecurityContextDeny,ServiceAccount,ResourceQuota"
   ```

```
KUBE_ADMISSION_CONTROL="--
admission_control=NamespaceLifecycle,NamespaceExists,LimitRanger,
SecurityContextDeny,ResourceQuota"
```

2. **Start the Kubernetes Master service daemons**: You need to start several services associated with a Kubernetes master:

```
# for SERVICES in etcd kube-apiserver kube-controller-manager
kube-scheduler; do
    systemctl restart $SERVICES
    systemctl enable $SERVICES
    systemctl status $SERVICES
done
```

3. **Start the Kubernetes Node service daemons**: You need to start several services associated with a Kubernetes Node:

```
# for SERVICES in docker kube-proxy.service kubelet.service; do
    systemctl restart $SERVICES
    systemctl enable $SERVICES
    systemctl status $SERVICES
done
```

4. **Check the services**: Run the ss command to check which ports the services are running on:

```
# ss -tulnp | grep -E "(kube)|(etcd)"
```

5. **Test the etcd service**: Use the curl command as follows to check the etcd service:

```
# curl -s -L http://localhost:2379/version
{"etcdserver":"2.1.1","etcdcluster":"2.1.0"}
```

Note: Since RHEL Atomic Host 7.1.4, the etcd utility uses port 2379. If you are running an older version of RHEL Atomic Host, replace 2379 with 4001 in the above command.

2.3.3. Launching container pods with Kubernetes

With Master and Node services running on the local system and the two container images in place, you can now launch the containers using Kubernetes pods. Here are a few things you should know about that:

- **Separate pods**: Although you can launch multiple containers in a single pod, by having them in separate pods each container can replicate multiple instances as demands require, without having to launch the other container.
- **Kubernetes service**: This procedure defines Kubernetes services for the database and web server pods so containers can go through Kubernetes to find those services. In this way, the database and web server can find each other without know the IP address, port number, or even the node the pod providing the service is running on.

The following steps show how to launch and test the two pods:

IMPORTANT: It is critical that the indents in the yaml file be maintained. Spacing in yaml files are part of what keep the format cleaner (not requiring curly braces or other characters to maintain the structure).

1. **Create a Database Kubernetes service**: Create a **db-service.yaml** file to identify the pod providing the database service to Kubernetes.

```
Kubernetes:
apiVersion: v1
kind: Service
metadata:
  labels:
    name: db
  name: db-service
  namespace: default
spec:
  ports:
  - port: 3306
  selector:
    name: db
```

2. **Create a Database server replication controller file**: Create a **db-rc.yaml** file that you will use to deploy the Database server pod. Here is what it could contain:

```
kind: "ReplicationController"
apiVersion: "v1"
metadata:
  name: "db-controller"
spec:
  replicas: 1
  selector:
    name: "db"
  template:
    spec:
      containers:
        - name: "db"
          image: "dbforweb"
          ports:
            - containerPort: 3306
    metadata:
      labels:
        name: "db"
      selectorname: "db"
  labels:
    name: "db"
```

3. **Create a Web server Kubernetes Service file**: Create a **webserver-service.yaml** file that you will use to deploy the Web server pod. Here is what it could contain:

```
Kubernetes:
apiVersion: v1
kind: Service
metadata:
  labels:
    name: webserver
```

```
  name: webserver-service
  namespace: default
spec:
  ports:
  - port: 80
  selector:
    name: webserver
```

4. **Create a Web server replication controller file**: Create a **webserver-rc.yaml** file that you will use to deploy the Web server pod. Here is what it could contain:

```
kind: "ReplicationController"
apiVersion: "v1"
metadata:
  name: "webserver-controller"
spec:
  replicas: 1
  selector:
    name: "webserver"
  template:
    spec:
      containers:
        - name: "apache-frontend"
          image: "webwithdb"
          ports:
            - containerPort: 80
    metadata:
      labels:
        name: "webserver"
        uses: db
  labels:
    name: "webserver"
```

5. **Orchestrate the containers with kubectl**: With the two yaml files in the current directory, run the following commands to start the pods to begin running the containers:

```
# kubectl create -f db-service.yaml
services/db-service
# kubectl create -f db-rc.yaml
replicationcontrollers/db-controller
# kubectl create -f webserver-service.yaml
services/webserver-service
# kubectl create -f webserver-rc.yaml
replicationcontrollers/webserver-controller
```

6. **Check rc, pods, and services**: Run the following commands to make sure that the replication controllers, pods, and services are all running:

```
# kubectl get rc
CONTROLLER             CONTAINER(S)      IMAGE(S)    SELECTOR
REPLICAS
db-controller          db                dbforweb    name=db
1
webserver-controller   apache-frontend   webwithdb
name=webserver   1
# kubectl get pod
```

```
NAME                         READY     STATUS    RESTARTS   AGE
db-controller-tiu56          1/1       Running   0          48m
webserver-controller-s5b81   1/1       Running   0          33m
# kubectl get service
NAME                LABELS          SELECTOR         IP(S)
PORT(S)
db-service          name=db         name=db
10.254.232.194   3306/TCP
kubernetes          component=...   <none>           10.254.0.1
443/TCP
webserver-service   name=webserver  name=webserver
10.254.104.45    80/TCP
```

7. **Check containers**: If both containers are running and the Web server container can see the Database server, you should be able to see the pods with the kubectl command and run the curl command to see that everything is working, as follows:

```
# kubectl get pods
# curl http://localhost:80/cgi-bin/action
<html>
<head>
<title>My Application</title>
</head>
<body>
<h2>RedHat rocks</h2>
<h2>Success</h2>
</body>
</html>
```

If you have a Web browser installed on the localhost, you can open that Web browser to see a better representation of the few lines of output. Just open the browser to this URL: **http://localhost/cgi-bin/action**.

2.4. EXPLORING KUBERNETES PODS

If something goes wrong along the way, there are several ways to determine what happened. One thing you can do is to examine services inside of the containers. To do that, you can look at the logs inside the container to see what happened. Run the following command (replacing the last argument with the pod name you want to examine).

Another problem that people have had comes from forgetting to disable firewalld. If firewalld is active, it could block access to ports when a service tries to access them between your containers. Make sure you have run **systemctl stop firewalld ; systemctl disable firewalld** on your host.

If you made a mistake creating your two-pod application, you can delete the replication controllers and the services. (The pods will just go away when the replication controllers are removed.) After that, you can fix the yaml files and create them again. Here's how you would delete the replication controllers and services:

```
# kubectl delete rc webserver-controller
replicationcontrollers/webserver-controller
# kubectl delete rc db-controller
replicationcontrollers/db-controller
```

```
# kubectl delete service webserver-service
services/webserver-service
# kubectl delete service db-service
```

Remember to not just delete the pods. If you do, without removing the replication controllers, the replication controllers will just start new pods to replace the ones you deleted.

The example you have just seen is a simple approach to getting started with Kubernetes. Because it involves only one master and one node on the same system, it is not scalable. To make a scalable configuration, I recommend you start with the procedure described in: Creating a Kubernetes Cluster to Run Docker Formatted Container Images

CHAPTER 3. CREATING A KUBERNETES CLUSTER TO RUN DOCKER FORMATTED CONTAINER IMAGES

3.1. OVERVIEW

Using Kubernetes, you can run and manage docker-formatted containers from one system (a Kubernetes master) and deploy them to run on other systems (Kubernetes nodes). Kubernetes itself consists of a set of system services that allow you to launch and manage Docker containers in what are referred to as **pods**. Those pods can run across multiple nodes and be interconnected by Kubernetes **services**. Once defined, pods can quickly scale up or down to meet demand.

This procedure will help you:

- Set up three RHEL Atomic Host or Red Hat Enterprise Linux 7 systems to use as a Kubernetes master and two Kubernetes nodes (previously called minions).
- Configure networking between containers using Flannel.
- Create data files (in yaml format) that define services, pods and replication controllers.

The services you start during this procedure include etcd, kube-apiserver, kubecontroller-manager, kube-proxy, kube-scheduler, and kubelet. To launch containers from yaml files and otherwise manage your Kubernetes environment, you run the **kubectl** command. This procedure assumes you are running either of the following systems:

- **RHEL 7.2 Atomic Host**: If you are not at RHEL 7.2, run **atomic host upgrade** and reboot your RHEL Atomic host.
- **RHEL 7.2 Server**: Your RHEL servers must include at least these versions (if they do not, run **yum upgrade kubernetes etcd docker flannel**) :
  - kubernetes-1.2
  - etcd-2.2
  - docker-1.6
  - flannel-0.2

If you have used an earlier version of this topic, refer to the Upgrading a Kubernetes Cluster article for information on upgrading that configuration to this latest software.

NOTE: Kubernetes and related services are built into RHEL Atomic Host. Because the long-term goal of Atomic is to be as lean as possible, the plan is to eventually remove Kubernetes software from the base RHEL Atomic system and into a set of Docker-formatted containers. To help you with that transition, most features for setting up a Kubernetes master are now available in containers. So you are given that choice to run some Kubernetes master services from containers in the course of this procedure.

3.2. PREPARING TO DEPLOY CONTAINERS WITH KUBERNETES

To get started, you need to obtain the latest RHEL Atomic Host or RHEL 7 Server installation media and install one of those operating systems on each of the three nodes. Instructions in Getting Started with Red Hat Enterprise Linux Atomic Host provide information on setting up Atomic to run on bare metal or in one of several different virtual environments. After that, you need to do some system configuration before setting up Kubernetes.

1. **Get Installation Media**: Get RHEL Atomic Host or RHEL Server media as follows:
   - RHEL Atomic Host 7.2 media: Get the RHEL Atomic Host 7.2 media you choose by going to the Red Hat Enterprise Linux Downloads and selecting from available media.
   - RHEL 7.2 Server media: Get the Red Hat Enterprise Linux 7.2 Server media you choose by going to the Red Hat Enterprise Linux Downloads and selecting from available media.
2. **Install RHEL or RHEL Atomic Host systems**: This procedure requires three or more RHEL Atomic host systems be installed. During installation, be sure to:
   - Set network interfaces so they can communicate with the other systems in the Kubernetes cluster.
   - Have enough disk space to handle the size and number of containers you want to use.
   - Reboot the system when installation is complete and do some further configuration.
3. **Configure Network Time Protocol**: All systems in the cluster (master and nodes) need to have their time synced together. Refer to the RHEL 7 System Administrator's Guide for instruction on setting up Network Time Protocol daemon (ntpd) or Chrony Daemon (chronyd) on those systems.
4. **Configure DNS or /etc/hosts**: Either make sure the systems can reach each other over DNS or add their names and IP addresses to **/etc/hosts**. For example, entries in the **/etc/hosts** file on each system could appear as follows:

   ```
   192.168.122.119   master
   192.168.122.196   node1
   192.168.122.27    node2
   ```

5. **Subscribe each system**: Subscribe all three systems using the **subscription-manager** command as follows:

   ```
   # subscription-manager register --auto-attach --username=
   <rhnuser> --password=<password>
   ```

6. **Upgrade each system**: Upgrading to the latest software is done differently, depending on whether you are working on a RHEL Server or RHEL Atomic Host system. Also, RHEL Atomic Host has Docker already installed, but to get Docker on a RHEL Server you need to enable a couple of repositories and install the docker package:

   On a RHEL Atomic Host, system type the following:

   ```
   # atomic host upgrade
   # reboot
   ```

   On a RHEL Server system, type the following:

   ```
   # yum upgrade -y
   # subscription-manager repos --enable=rhel-7-server-extras-rpms
   # subscription-manager repos --enable=rhel-7-server-optional-rpms
   # yum install docker
   ```

7. **Disable firewalld**: If you are using a RHEL 7 host, be sure that the firewalld service is disabled (the firewalld service is not installed on RHEL Atomic Host). On RHEL 7, type the following to disable and stop the firewalld service:

```
# systemctl disable firewalld
# systemctl stop firewalld
```

8. **Start and Enable docker**: To make sure the docker services is running on each system (including the master, if you plan to run Kubernetes from a container on the master), run the following

```
# systemctl restart docker
# systemctl enable docker
# systemctl status docker
* docker.service - Docker Application Container Engine
   Loaded: loaded (/usr/lib/systemd/system/docker.service;
enabled; vendor preset: disabled)
  Drop-In: /usr/lib/systemd/system/docker.service.d
           |-flannel.conf
   Active: active (running) since Wed 2015-11-04 15:37:10 EST;
22h ago
...
```

9. **Get Docker Containers**: Build the following two images: Building a Simple Apache Web Server in a Docker Container and Building a Simple Database Server in a Docker Container

10. **Import the containers to each node**: Copy each image to each node and load them to each node. Starting from the system where you built the images, do the following:

```
# docker export mydbforweb > dbforweb.tar
# docker export mywebwithdb > webwithdb.tar
# scp dbforweb.tar webwithdb.tar node1:/tmp
# scp dbforweb.tar webwithdb.tar node2:/tmp
```

On each node, do the following

```
# cat /tmp/webwithdb.tar | docker import - webwithdb
# cat /tmp/dbforweb.tar | docker import - dbforweb
# docker images | grep web
dbforweb  latest  2e48be49fec4  About a minute ago  568.7 MB
webwithdb latest  5f2daa2e04e9  2 minutes ago       408.6 MB
```

NOTE: As an alternative to importing the containers to each node, you can create a private container registry and have the yaml files we create later point to that registry when launching the containers.

After you build and test the containers, stop them (**docker stop mydbforweb** and **docker stop mywebwithdb**).

The basic setup of your three RHEL Atomic Hosts should be complete. You are ready to start setting up Kubernetes.

3.3. SETTING UP KUBERNETES

With the three systems in place, the next thing is to set up Kubernetes. This procedure varies slightly depending on whether you are:

- Setting up a master or a node

- Using RHEL Server or RHEL Atomic Host systems
- Running master services from systemd services on the host or from containers on the host (containerized versions of node services are not yet available)

IMPORTANT: Be sure to configure and start the master before you start the nodes. If the nodes come up and the master is not yet up, the nodes may not be properly registered with Kubernetes.

3.4. SETTING UP KUBERNETES ON THE MASTER

The following steps describe how to set up and run the services on the Kubernetes master.

1. **Install Kubernetes**: If you are on a RHEL 7 system, install the kubernetes and etcd packages on the master. Both are already installed on RHEL Atomic:

```
# yum install kubernetes-master etcd
```

2. **Configure the etcd service**: Edit the **/etc/etcd/etcd.conf**. The etcd service needs to be configured to listen on all interfaces to ports 2380 (ETCD_LISTEN_PEER_URLS) and port 2379 (ETCD_LISTEN_CLIENT_URLS), and listen on 2380 on localhost (ETCD_LISTEN_PEER_URLS). The resulting uncommented lines in this file should appear as follows:

```
ETCD_NAME=default
ETCD_DATA_DIR="/var/lib/etcd/default.etcd"
ETCD_LISTEN_CLIENT_URLS="http://0.0.0.0:2379"
ETCD_LISTEN_PEER_URLS="http://localhost:2380"
ETCD_ADVERTISE_CLIENT_URLS="http://0.0.0.0:2379"
```

3. **Edit the Kubernete config file**: Edit the **/etc/kubernetes/config** file and change the KUBE_MASTER line to identify the location of your master server (it points to 127.0.0.1, by default). Leave other settings as they are. The changed line appears as follows:

```
KUBE_MASTER="--master=http://master.example.com:8080"
```

4. **Configure the Kubernetes apiserver**: Edit the **/etc/kubernetes/apiserver** and add a new KUBE_ETCD_SERVERS line (as shown below), then review and change other lines in the apiserver configuration file. Change KUBE_API_ADDRESS to listen on all network addresses(0.0.0.0), instead of just localhost. Set an address range for the KUBE_SERVICE_ADDRESS that Kubernetes can use to assign to services (see a description of this address below). Finally, remove the term "ServiceAccount" from the KUBE_ADMISSION_CONTROL instruction (below is the new line, with the original line above it commented out). Here are examples:

```
KUBE_API_ADDRESS="--insecure-bind-address=0.0.0.0"
KUBE_ETCD_SERVERS="--etcd_servers=http://127.0.0.1:2379"
KUBE_SERVICE_ADDRESSES="--service-cluster-ip-range=10.254.0.0/16"
# KUBE_ADMISSION_CONTROL="--
admission_control=NamespaceLifecycle,NamespaceExists,LimitRanger,
SecurityContextDeny,ServiceAccount,ResourceQuota"
KUBE_ADMISSION_CONTROL="--
admission_control=NamespaceLifecycle,NamespaceExists,LimitRanger,
SecurityContextDeny,ResourceQuota"
```

Here are a few things you should know about the KUBE_SERVICE_ADDRESSES address range you use:

- The address range is used by Kubernetes to assign to Kubernetes services.
- In the example just shown, the address range of 10.254.0.0/16 consumes a set of 10.254 subnets that can be assigned by Kubernetes as needed. For example, 10.254.1.X, 10.254.2.X and so on.
- Make sure this address range isn't used anywhere else in your environment.
- Each address range that is assigned is only used within a node and is not routable outside of that node.
- This address range must be different than the range used by flannel. (Flannel address ranges are assigned to pods.)

5. **Start master services**: You have the choice of either running the kubernetes services from the host's file system or from containers. The **etcd** service should be run from the host at the moment. Choose which way you want to go and follow the appropriate commands below:

   - **From the host** To run the Kubernetes master services directly from the host's file system, you need to enable and start several systemd services. From the master, run the following **for** loop to start and enable Kubernetes systemd services on the master:

     ```
     # for SERVICES in etcd kube-apiserver kube-controller-manager
     kube-scheduler; do
         systemctl restart $SERVICES
         systemctl enable $SERVICES
         systemctl status $SERVICES
         done
     ```

   - **From containers**: Instead of running Kubernetes master services from the host (using systemctl), you can run them from containers. While a fully-containerized Kubernetes is a long-term goal, some of the components are still being developed. The procedure here shows a way to manually try out Kubernetes services run from containers. Keep an eye out for a more permanent solution in the Running Kubernetes from Containers article as that procedure becomes available.

     To run containerized Kubernetes master services, you need to make sure the services are stopped and disabled from systemd. Then you need to configure the container services as follows:

     - **Disable Kubernetes master systemd services**: Run the following **for** loop on the master. As a result, all symlinks should be removed and all services should show a inactive or failed. Be sure to leave **etcd** as a systemd service:

       ```
       # for SERVICES in kube-apiserver kube-controller-manager
       kube-scheduler; do
           systemctl stop $SERVICES
           systemctl disable $SERVICES
           systemctl is-active $SERVICES
           done
       # systemctl restart etcd ; systemctl enable etcd
       ```

     - **Get the kubernetes master containers**: To pull the containers you need, run the following:

```
# docker pull rhel7/kubernetes-controller-mgr
# docker pull rhel7/kubernetes-apiserver
# docker pull rhel7/kubernetes-scheduler
```

- **Start the kubernetes master services**: To start the kubernetes services on the master, type the following:

```
# docker run -v /etc/kubernetes/:/etc/kubernetes --net=host -d
\
   rhel7/kubernetes-apiserver

# docker run -v /etc/kubernetes/:/etc/kubernetes --net=host -d
\
   rhel7/kubernetes-controller-mgr

# docker run -v /etc/kubernetes/:/etc/kubernetes --net=host -d
\
   rhel7/kubernetes-scheduler
```

- **Check kubernetes master services**: Because the kubernetes services are not running as systemd services, you cannot use **systemctl** to check that the services are running. To check if they are running, you can try the **ps** command:

```
# ps -ef | grep kube
root  2566   1507  0 11:26 ?  00:00:26 /usr/bin/kube-apiserver --
logtostderr=true ...
root  2619   1507  0 11:26 ?  00:00:13 /usr/bin/kube-controller-
manager ...
root  2671   1507  0 11:26 ?  00:00:01 /usr/bin/kube-scheduler --
logtostderr=true ...
```

3.5. SETTING UP KUBERNETES ON THE NODES

On each of the two Kubernetes nodes (node1.example.com and node2.example.com in this example), you need to edit several configuration files and start and enable several Kubernetes systemd services:

1. **Install Kubernetes**: If your nodes are RHEL 7 systems, install the kubernetes-node package on each node. Kubernetes is already installed on RHEL Atomic:

```
# yum install kubernetes-node
```

2. **Edit /etc/kubernetes/config**: Edit the KUBE_MASTER line in this file to identify the location of your master (it is 127.0.0.1, by default). Leave other settings as they are.

```
KUBE_MASTER="--master=http://master.example.com:8080"
```

3. **Edit /etc/kubernetes/kubelet**: In this file on each node, modify KUBELET_ADDRESS (0.0.0.0 to listen on all network interfaces), KUBELET_HOSTNAME (replace hostname_override with the hostname or IP address of the local system: node1 or node2), set KUBELET_ARGS to "--register-node=true", and KUBELET_API_SERVER (set --api_servers=http://master.example.com:8080 or other location of the master), as shown below:

```
KUBELET_ADDRESS="--address=0.0.0.0"
KUBELET_HOSTNAME="--hostname-override=node?"
KUBELET_ARGS="--register-node=true"
KUBELET_API_SERVER="--api_servers=http://master.example.com:8080"
```

4. **Edit /etc/kubernetes/proxy**: No settings are required in this file. If you have set KUBE_PROXY_ARGS, you can comment it out:

```
# KUBE_PROXY_ARGS="--master=http://master.example.com:8080"
```

5. **Start the Kubernetes nodes systemd services**: On each node, you need to start several services associated with a Kubernetes node:

```
# for SERVICES in docker kube-proxy.service kubelet.service; do
    systemctl restart $SERVICES
    systemctl enable $SERVICES
    systemctl status $SERVICES
done
```

6. **Check the services**: Run the **netstat** command on each of the three systems to check which ports the services are running on. The etcd service should only be running on the master.

   - On master:

     ```
     # netstat -tulnp | grep -E "(kube)|(etcd)"
     ```

   - On nodes:

     ```
     # netstat -tulnp | grep kube
     ```

     NOTE: Although the firewalld service is not installed in Atomic (and should be disabled on RHEL) for this procedure, it is possible that you might be using iptables rules directly to block access to ports on your system. If you have done that, the **netstat** command just shown displays the ports you need to have accessible on each system for the Kubernetes cluster to work. So you can create individual firewall rules to open access to those ports.

7. **Test the etcd service**: Use the curl command from any of the three systems to check that the etcd service is running and accessible:

```
# curl -s -L http://master.example.com:2379/version
{"etcdserver":"2.2.0","etcdcluster":"2.2.0"}
```

8. **Check the nodes**: From the master, type the following command to make sure the master is communicating with the two nodes:

```
# kubectl get nodes
NAME              LABELS
STATUS
node1.example.com kubernetes.io/hostname=node1.example.com Ready
node2.example.com kubernetes.io/hostname=node2.example.com Ready
```

At this point, you have a configured Kubernetes cluster. Next, you can begin using this setup to deploy containers by configuring networking, then deploying Kubernetes services, replication

controllers, and pods.

3.6. SETTING UP FLANNEL NETWORKING FOR KUBERNETES

The flannel package contains features that allow you to configure networking between the master and nodes in your Kubernetes cluster. You configure the flanneld service by creating and uploading a json configuration file with your network configuration to your etcd server (on the master). You then configure the flanneld systemd service on the master and each node to point to that etcd server and start the flanneld service. This procedure describes how to do that.

NOTE: The etcd service must be running before flanneld can start. So, if flanneld and docker are failing to start on the nodes, make sure that etcd is up and running on the master, then try to restart each node.

For this example, flannel assigns an address range of **10.20.0.0/16** to be used by all nodes in the Kubernetes environment. This allows 24 separate subnets within that range to be assigned to network interfaces on the master and both nodes. For the three systems in this example, the steps below resulted in the following address ranges being assigned to flannel.1 and docker0 interfaces:

- master:
  - flannel.1: 10.20.21.0/16
- node1:
  - flannel.1: 10.20.26.0/16
  - docker0: 10.20.26.1/24
- node2:
  - flannel.1: 10.20.37.0/16
  - docker0: 10.20.37.1/24

IMPORTANT: Because the docker0 interface is probably already in place when you run this procedure, the IP address range assigned by flanneld to docker0 will not immediately take effect. To get the flanneld address ranges to take effect, you need to stop docker (**systemctl stop docker**), delete the docker0 interface, and restart the docker interface (**systemctl start docker**). When you get to that step, the procedure simply asks you to reboot.

1. **Install flannel**: The flannel package is preinstalled for RHEL Atomic. If you are using RHEL 7, however, you can install the flannel package on the master and each node as follows:

   ```
   # yum install flannel
   ```

2. **Create Flannel configuration file**: On the master, identify a set of IP addresses and network type in a Flannel configuration file (in json format). For this example, we created a file named **flannel-config.json** that contains the following content:

   ```
   {
     "Network": "10.20.0.0/16",
     "SubnetLen": 24,
     "Backend": {
       "Type": "vxlan",
       "VNI": 1
     }
   ```

```
}
```

3. **Upload Flannel configuration to etcd service**: On the master, to upload the flannel configuration file to the etcd service, type the following:

```
# etcdctl set coreos.com/network/config < flannel-config.json
  {"action":"set","node":
{"key":"/coreos.com/network/config","value":"{\n\"Network\":
\"10.20.0.0/16\",\n\"SubnetLen\": 24,\n\"Backend\": {\n\"Type\":
\"vxlan\",\n\"VNI\": 1\n
}\n}\n","modifiedIndex":10,"createdIndex":10},"prevNode":
{"key":"/coreos.com/network/config","value":"","modifiedIndex":9,
"createdIndex":9}}
```

   Then check that the upload completed properly:

```
# etcdctl get coreos.com/network/config
{
  "Network": "10.20.0.0/16",
  "SubnetLen": 24,
  "Backend": {
    "Type": "vxlan",
    "VNI": 1
  }
}
```

4. **Configure flanneld overlay network**: When the flanneld systemd service starts up, it reads the **/etc/sysconfig/flanneld** file for options to pass to the flanneld daemon. On the master and both nodes, edit **/etc/sysconfig/flanneld** to insert the name or IP address of the system containing the etcd service (master). Leave the FLANNEL_ETCD_KEY line as it is:

```
FLANNEL_ETCD="http://master.example.com:2379"
FLANNEL_ETCD_KEY="/coreos.com/network"
```

5. **Start flanneld on master and nodes**: First on the master, then on the two nodes, start and enable the flanneld service as follows:

```
# systemctl restart flanneld
# systemctl enable flanneld
# systemctl status flanneld
```

6. **Check flannel.1 network interfaces**: Run the following commands from any system to check that the flannel.1 network interface is configured properly on each system. The first command runs the ip a command on each node to check for the flannel.1 interface. The second command shows how the subnet is configured for flannel on each system.

```
# for i in 1 2; do ssh root@node$i ip a l flannel.1; done | grep
'inet '
root@node1's password: password
    inet 10.20.26.0/16 scope global flannel.1
root@node2's password: password
    inet 10.20.37.0/16 scope global flannel.1
# for i in master node1 node2; \
     do echo --- $i ---; ssh root@$i cat
/run/flannel/subnet.env; done
```

```
root@master's password: password
--- master ---
FLANNEL_SUBNET=10.20.21.1/24
FLANNEL_MTU=1450
root@node1's password: password
--- node1 ---
FLANNEL_SUBNET=10.20.26.1/24
FLANNEL_MTU=1450
root@node2's password: password
--- node2 ---
FLANNEL_SUBNET=10.20.37.1/24
FLANNEL_MTU=1450
```

7. **Reboot nodes**: For the docker systemd service to pick up the flannel changes, and to make sure the network interfaces all come up properly, run the following on each node to reboot it:

```
# systemctl reboot
```

8. **Start a container to get flannel addresses** On each node, check the address range of the flannel network, then run an image to make sure it picks up addresses from that network:

```
# ip a | grep flannel
3: flannel.1:  mtu 1450 qdisc noqueue state UNKNOWN
inet 10.20.26.0/16 scope global flannel.1
# docker run -d --name=mydbforweb dbforweb
# docker inspect --format='{{.NetworkSettings.IPAddress}}'
mydbforweb
10.20.26.2
# docker stop mydbforweb
# docker rm mydbforweb
```

3.7. LAUNCHING SERVICES, REPLICATION CONTROLLERS, AND CONTAINER PODS WITH KUBERNETES

With the Kubernetes cluster in place, you can now create the yaml files needed to set up Kubernetes services, define replication controllers and launch pods of containers. Using the two containers described earlier in this topic (Web and DB), you will create the following types of Kubernetes objects:

- **Services**: Creating a Kubernetes service lets you assign a specific IP address and port number to a label. Because pods and IP addresses can come and go with Kubernetes, that label can be used within a pod to find the location of the services it needs.
- **Replication Controllers**: By defining a replication controller, you can set not only which pods to start, but how many replicas of each pod should start. If a pod stops, the replication controller starts another to replace it.
- **Pods**: A pod loads one or more containers, along with options associated with running the containers.

In this example, we create the service objects (yaml files) needed to allow the two pods to communicate with each other. Then we create replication controllers that identify the pods that launch and maintain the Web server and database server containers described earlier. For copies of all the yaml and json configuration files used in this topic, download the following tarball:

kube_files.tar

1. **Deploy Kubernetes Service for Database server**: On the master, create a Kubernetes service yaml file that identifies a label to tie the database server container to a particular IP address and port. Here is an example using a file called **db-service.yaml**:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    name: db
  name: db-service
  namespace: default
spec:
  ports:
  - port: 3306
  selector:
    name: db
```

   This service in not directly accessed from the outside world. The webserver container will access it. The selector and labels name is set to db. To start that service, type the following on the master:

```
# kubectl create -f db-service.yaml
service "db-service" created
```

2. **Deploy Kubernetes ReplicationController for Database server**: Create a replication controller yaml file that identifies then number of database pods to have running (two in this case). In this example, the file is named **db-rc.yaml**:

```
kind: ReplicationController
apiVersion: v1
metadata:
   name: db-controller
spec:
  replicas: 2
  selector:
    name: db
  template:
    spec:
      containers:
      - name: db
        image: dbforweb
        ports:
          - containerPort: 3306
    metadata:
      labels:
        name: "db"
      selectorname: "db"
  labels:
    name: "db"
```

   To start the replication controller for the webserver pod, type the following:

```
# kubectl create -f db-rc.yaml
replicationcontroller "db-controller" created
```

3. **Create Kubernetes Service yaml file for Web server and start it**: On the master, create a Kubernetes service yaml file that identifies a label to tie the Web server container to a particular IP address and port. Here is an example using a file called **webserver-service.yaml**:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    name: webserver
  name: webserver-service
  namespace: default
spec:
  ports:
  - port: 80
  publicIPs:
  - 192.168.122.15
  selector:
    name: webserver
```

The portalIP must be in the range set in the **/etc/kubernetes/apiserver** file (10.254.100.0/24, in this case). The publicIPs value should align with an IP address associated with an external network interface on one of the nodes (on node2, the IP address for the eth0 interface is 192.168.122.15). That will make the service available to the outside world. The selector and labels name is set to webserver. To start that service, type the following on the master:

```
# kubectl create -f webserver-service.yaml
service "webserver-service" created
```

With the with the apiserver and the two services we just added running, run the following command to see those services:

```
# kubectl get services
NAME              LABELS
SELECTOR
          IP(S)          PORT(S)
db-service        name=db
name=db
          10.254.192.67  3306/TCP
kubernetes        component=apiserver,provider=kubernetes
          10.254.255.128 443/TCP
kubernetes-ro     component=apiserver,provider=kubernetes
          10.254.139.205 80/TCP
webserver-service name=webserver
name=webserver
          10.254.134.105 80/TCP
```

4. **Deploy Kubernetes ReplicationController for Web server**: Create a replication controller yaml file that identifies the number of webserver pods to have running. Based on the following yaml file, the replication controller will try to keep two pods labeled "webserver" running at all times. The pod definition is inside the replications controller yaml file, so no

separate pod yaml file is needed. Any running pod with the webserver id will be taken by the replication controller as fulfilling the requirement (regardless of how the pod was started). Here's an example of a file called **webserver-rc.yaml**:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: webserver-controller
spec:
  replicas: 2
  selector:
    name: webserver
  template:
    spec:
      containers:
        - name: apache-frontend
          image: webwithdb
          ports:
            - containerPort: 80
    metadata:
      labels:
        name: webserver
        uses: db
  labels:
    name: "webserver"
```

To start the replication controller for the webserver pod, type the following:

```
# kubectl create -f webserver-rc.yaml
replicationcontroller "webserver-controller" created
```

3.8. CHECKING KUBERNETES

To check that the application is working, you can run a couple of **curl** commands to get data from the Web server. To check that you cluster is working properly, there are some **kubectl** commands you can run.

1. **Check Web server**: Run the following commands to get data from the Web server. Use the pod address to access the web service:

```
# kubectl get endpoints | grep web
NAME                ENDPOINTS                          AGE
webserver-service   10.20.11.3:80,10.20.55.2:80        3d
# curl http://10.20.11.3/index.html
The Web Server is Running
# curl http://10.20.11.3/cgi-bin/action
<html>
<head>
<title>My Application</title>
</head>
<body>
```

```
<h2>RedHat rocks</h2>
<h2>Success</h2>
</body>
</html>
```

2. **Check the Kubernetes Cluster**: Here are some commands you can run from the master to check the cluster:

```
# kubectl get pods
# kubectl get replicationControllers
# kubectl get services
# kubectl get nodes
```

3. **Check Kubernetes Nodes and Pause Containers**: If a pod remains in a Pending state for a long time, it might be because the container the pod is trying to launch is not able to start on the selected node. To check out the containers running on a particular node, log into that node, and type the **docker ps** command:

```
# docker ps
CONTAINER ID        IMAGE
COMMAND                  CREATED             STATUS
PORTS               NAMES
b28a863e0aca        dbforweb
"/usr/bin/mysqld_safe"   2 hours ago         Up 2 hours
k8s_db...
b138c90519a1        webwithdb
"/usr/sbin/httpd -D F"   2 hours ago         Up 2 hours
k8s_apache...
872c5acf3ec8        beta.gcr.io/google_containers/pause:2.0
"/pause"                 2 hours ago         Up 2 hours
k8s_POD...
e117af93e4f4        beta.gcr.io/google_containers/pause:2.0
"/pause"                 2 hours ago         Up 2 hours
k8s_POD...
```

The output above shows that we successfully launched the two replication controllers from the master. The dbforweb and webwithdb containers are running on this node. Notice, however, that two other containers, running the command **/pause** appear in the list of running containers as well.

A **pause** container is a special type of container. The one shown here comes from the upstream Kubernetes project. For each pod, a pause container is used to keep the networking namespace reserved on a node when it is the last one in a pod, so that namespace can remain in place as pods come and go. All the default **pause** container does is sleep. This container gets pulled in automatically by Kubernetes.

A potential problem with the **pause** container is that if you want to run your Kubernetes cluster in a way that is disconnected from the registry from which this needs to be pulled, you need to make sure the **pause** container is pulled before you disconnect.

There are other **pause** containers available to use instead of the default one shown in the previous example. An official pause container from Red Hat called **rhel7/pod-infrastructure** will soon be available to use.

3.9. CLEANING UP KUBERNETES

If you are done using your cluster, you should take it down in a particular way, to make sure it is done neatly.

The order is important. If you delete a pod before deleting the replicationController, the pod is restarted. Here is an example:

```
# kubectl delete replicationControllers webserver-controller
# kubectl delete replicationControllers db-controller
# kubectl delete services webserver-service
# kubectl delete services db-service
# kubectl get pods
POD                                    IP          CONTAINER(S)    ...
89d8577d-bceb-11e4-91c4-525400a6f1a0 10.20.26.4  apache-frontend ...
8d30f2de-bceb-11e4-91c4-525400a6f1a0 10.20.26.5  db              ...
# kubectl delete pods 89d8577d-bceb-11e4-91c4-525400a6f1a0
# kubectl delete pods 8d30f2de-bceb-11e4-91c4-525400a6f1a0
```

If you plan to continue using your Kubernetes cluster, run the following on each node to make sure the services are up and running:

```
# for SERVICES in docker kube-proxy.service kubelet.service; do
    systemctl restart $SERVICES
    systemctl status $SERVICES
    done
```

If you are done with the nodes, you can remove them from the master as follows:

```
# kubectl delete node node1.example.com
# kubectl delete node node2.example.com
```

3.10. ATTACHMENTS

kube_files.tar

CHAPTER 4. TROUBLESHOOTING KUBERNETES

4.1. OVERVIEW

Kubernetes provides a set of features for deploying and managing applications in sets of docker formatted containers. this topic will help you troubleshoot problems in creating and managing Kubernetes pods, replication controllers, and services, as well as the resulting containers.

For the purpose of illustrating troubleshooting techniques, this topic uses the containers and configuration deployed in the Creating a Kubernetes Cluster to Run Docker Formatted Container Images article. Techniques described here should apply to Kubernetes running on Red Hat Enterprise Linux 7.1 and RHEL Atomic Host systems.

4.2. UNDERSTANDING KUBERNETES TROUBLESHOOTING

Before you begin troubleshooting Kubernetes, you should have an understanding of the Kubernetes components being investigated. These include:

- **Master**: The system from which you manage your Kubernetes environment.
- **Nodes**: One or more systems on which containers are deployed by Kubernetes (nodes were previously called minions).
- **Pods**: A pod defines one or more containers to run, as well as options to the docker run command for each container and labels to define the location of services.
- **Services**: A service allows a container within a Kubernetes environment to find an application provided by another container by name (label), without knowing its IP address.
- **Replication controllers**: A replication controller lets you designate that a certain number of pods should be running. (New pods are started until the required number is reached and if a pod dies, a new pod is run to replace it.)
- **Networking (flanneld)**: The flanneld service lets you configure IP address ranges and related setting to be used by Kubernetes. This feature is optional. yaml and json files: The Kubernetes elements we'll work with are actually created from configuration files in yaml or json formats. this topic focuses primarily on yaml-formatted files.

You will troubleshoot the components just described using these commands in particular:

- **kubectl**: The **kubectl** command (run from the master) lets you create, delete, get (list information) and perform other actions on Kubernetes pods, services and replication controllers. You'll use this command to test your yaml/json files, as well as see the state of the different Kubernetes components.
- **systemctl**: Specific **systemd** services must be configured with Kubernetes to facilitate communications between master and nodes. Those services must also be active and enabled.
- **journalctl**: You can use the **journalctl** command to check Kubernetes systemd services to follow the processing of those services. You can run it on both the master and nodes to check for Kubernetes failures on those systems. All daemon logging in kubernetes uses the systemd journal.

- **etcdctl** or **curl**: The **etcd** daemon manages the storage of information for the Kubernetes cluster. This service can run on the master or on some other system. You can use the etcdctl command in RHEL to query that information. In RHEL Atomic, the etcdctl command is not included, so you can use the curl command instead to query the etcd service.

4.3. PREPARING CONTAINERIZED APPLICATIONS FOR KUBERNETES

Some of the things you should consider before deploying an application to Kubernetes are described below.

4.3.1. Networking Constraints

All Applications are not equally kuberizable, because there are limitations on the type of applications that can be run as Kubernetes services. In Kubernetes, a service is a load balanced proxy whose IP address is injected into the iptables of clients of that service. Therefore, you should verify that the application you intend to "kuberize":

- Can support network address translation or NAT-ing across its subprocesses.
- Does not require forward and reverse DNS lookup. Kubernetes does not provide forward or reverse DNS lookup to the clients of a service.

If neither of these restrictions apply, or if the user can disable these checks, you can continue on.

4.3.2. Preparing your Containers

Depending on the type of software you are running you may wish to take advantage of some predefined environment variables that are provided for clients of Kubernetes services.

For example, given a service named **db**, if you launch a Pod in Kubernetes that uses that service, Kubernetes will inject the following environment variables into the containers in that pod:

```
DB_SERVICE_PORT_3306_TCP_PORT=3306
DB_SERVICE_SERVICE_HOST=10.254.100.1
DB_SERVICE_PORT_3306_TCP_PROTO=tcp
DB_SERVICE_PORT_3306_TCP_ADDR=10.254.100.1
DB_SERVICE_PORT_3306_TCP=tcp://10.254.100.1:3306
DB_SERVICE_PORT=tcp://10.254.100.1:3306
DB_SERVICE_SERVICE_PORT=3306
```

NOTE: Notice that the service name (db) is capitalized in the variables (DB). If there were dashes (-) in the name, they would be converted to underscores (_).

To see these and other shell variables, use **docker exec** to open a shell to the active container and run **env** to see the shell variables:

```
# docker exec -it <container_ID> /bin/bash
[root@e7ea67..]# env
...
WEBSERVER_SERVICE_SERVICE_PORT=80
KUBERNETES_RO_SERVICE_PORT=80
KUBERNETES_SERVICE_PORT=443
KUBERNETES_RO_PORT_80_TCP_PORT=80
```

```
KUBERNETES_SERVICE_HOST=10.254.255.128
DB_SERVICE_PORT_3306_TCP_PORT=3306
DB_SERVICE_SERVICE_HOST=10.254.100.1
WEBSERVER_SERVICE_PORT_80_TCP_ADDR=10.254.100.50
...
```

When starting your client applications you may want to leverage those variables. For example, when you create a Web server/database application as described in Creating a Kubernetes Cluster to Run Docker Formatted Container Images, the action script that the Web server deploys pulls in the value of DB_SERVICE_SERVICE_HOST so it knows the IP address it can use to access the database service.

If you are debugging communications problems between containers, viewing these shell variables is a great way to see each container's view of the addresses of services and ports.

4.4. DEBUGGING KUBERNETES

Before you start debugging Kubernetes, it helps to have a high level of understanding of how Kubernetes works. When you submit an application to Kubernetes, here's generally what happens:

1. Your kubectl command line is sent to the kube-apiserver (on the master) where it is validated.
2. The kube-scheduler process (on the master) reads your yaml or json file and assigns pods to nodes (nodes are systems running the kubelet service).
3. The kublet service (on a node) converts the pod manifest into one or more **docker run** calls.
4. The **docker run** command tries to start up the identified containers on available nodes.

So, to debug a kubernetes deployed app, you need to confirm that:

1. The Kubernetes service daemon (systemd) processes are running.
2. The yaml or json submission is valid.
3. The kubelet service is receiving a work order from the kube-scheduler.
4. The kubelet service on each node is able to successfully launch each container with docker.

Note

The above list is missing the kube-controller-manager which is important if you do things like create a replication controller, but you see no pods being managed by it. Or you have registered nodes with the cluster, but you are not getting information about their available resources, etc.

Also note, there is a movement upstream to an all-in-one hyperkube binary, so terminology here may need to change in the near future.

4.4.1. Inspecting and Debugging Kubernetes

From the Kubernetes master, inspect the running Kubernetes configuration. As noted earlier, the troubleshooting examples in this section are done on the configuration created from the Creating a Kubernetes Cluster to Run Docker Formatted Container Images article.

We'll start by showing you how this configuration should look when everything is working. Then we'll show you how to the setup might break in various ways and how you can go about fixing it.

4.4.2. Querying the State of Kubernetes

Using **kubectl** is the simplest way to manually debug the process of application submission, service creation, and pod assignment. To see what pods, services, and replication controllers are active, run these commands on the master:

```
# kubectl get pods
POD             IP          CONTAINER(S)     IMAGE(S)  HOST
LABELS                                         STATUS
4e04dd3b-c...   10.20.29.3  apache-frontend  webwithdb
node2.example.com/  name=webserver,selectorname=webserver,uses=db
Running
5544eab2-c...   10.20.48.15 apache-frontend  webwithdb
node1.example.com/  name=webserver,selectorname=webserver,uses=db
Running
1c971a09-c...   10.20.29.2  db               dbforweb
node2.example.com   name=db,selectorname=db
Running
1c97a755-c...   10.20.48.14 db               dbforweb
node1.example.com/  name=db,selectorname=db
Running
# kubectl get services
NAME               LABELS                                    SELECTOR
IP                  PORT
webserver-service  name=webserver
name=webserver  10.254.100.50       80
db-service         name=db                                   name=db
10.254.100.1        3306
kubernetes         component=apiserver,provider=kubernetes
10.254.92.19        443
kubernetes-ro      component=apiserver,provider=kubernetes
10.254.206.141      80
# kubectl get replicationControllers
CONTROLLER             CONTAINER(S)        IMAGE(S)
SELECTOR                 REPLICAS
webserver-controller   apache-frontend     webwithdb
selectorname=webserver   2
db-controller          db                  dbforweb
selectorname=db          2
```

Here's information to help you interpret this output:

- Pods are either in Waiting or Running states. The fact that all four pods are running here is a good sign.
- The replication controller successfully started two apache-frontend and two db containers. They were distributed across node1 and node2.

- The **uses** label for apache-frontend lets that container find the db container through the db-service Kubernetes service.
- The services listing identifies the IP address and port number for each service that can be requested from pods by each service's label name.
- The kubernetes and kubernetes-ro services provide access to the kube-apiserver systemd service.

If something goes wrong in the process of getting to this state, the following sections will help you troubleshoot problems.

4.5. TROUBLESHOOTING KUBERNETES SYSTEMD SERVICES

Kubernetes is implemented using a set of service daemons that run on Kubernetes masters and nodes. If these systemd services are not working properly, you will experience failures. Things you should know about avoiding or fixing potential problems with Kubernetes systemd services are described below.

4.5.1. Checking that Kubernetes systemd Services are Up

A Kubernetes cluster that consists of a master and one or more nodes (minions) needs to initialize a particular set of systemd services. You should verify that the following services are running on the master and on each node:

- **Start Master first**: The services on the master should come before starting the services on the nodes. The nodes will not start up properly if the master is not already up.
- **Master services**: Services include: kube-controller-manager, kube-scheduler, flanneld, etcd, and kube-apiserver. The flanneld service is optional and it is possible to run the etcd services on another system.
- **Node services**: Services include: docker kube-proxy kubelet flanneld. The flanneld service is optional.

Here's how you verify those services on the master and each node:

Master: On your kubernetes master server, this will tell you if the proper services are active and enabled (flanneld may not be configured on your system):

```
# for SERVICES in etcd flanneld kube-apiserver kube-controller-manager
kube-scheduler;
    do echo --- $SERVICES --- ; systemctl is-active $SERVICES ;
    systemctl is-enabled $SERVICES ; echo "";  done
--- etcd ---
active
enabled
--- flanneld ---
active
enabled
--- kube-apiserver ---
active
enabled
--- kube-controller-manager ---
active
```

```
enabled
--- kube-scheduler ---
active
enabled
```

Minions (nodes): On each node, make sure the proper services are active and enabled:

```
# for SERVICES in flanneld docker kube-proxy.service kubelet.service; \
do echo --- $SERVICES --- ; systemctl is-active $SERVICES ; \
systemctl is-enabled $SERVICES ; echo "";  done
--- flanneld ---
active
enabled
--- docker ---
active
enabled
--- kube-proxy.service ---
active
enabled
--- kubelet.service ---
active
enabled
```

If any of the master or node systemd services are disabled or failed, here's what to do:

- Try to enable or activate the service as described in the Starting Kubernetes Services section of the Kubernetes Cluster article.
- Check the systemd journal on the system where a service is failing and look for hints on how to fix the problem. One way to do that is to use the journalctl with the command representing the service. For example:

  ```
  # journalctl -l -u kubelet
  # journalctl -l -u kube-apiserver
  ```

- If the services still don't start, refer to the Kubernetes Cluster article again and check that each service's configuration file is set up properly.

4.5.2. Checking Firewall for Kubernetes

There is no iptables or firewalld service installed on RHEL Atomic. So, by default, there are no firewall filter rules blocking access to Kubernetes services. However, if you have a firewall running on a RHEL host or if you have added iptables firewall rules to your Kubernetes master or nodes to filter incoming ports, you need to make sure that the ports that need to be exposed on those systems are not blocked.

The following is the output of a **netstat** command, showing which ports Kubernetes and related services are listening on a Kubernetes nodes:

```
# netstat -tupln
tcp6       0      0 :::10249          :::*          LISTEN
125528/kube-proxy
tcp6       0      0 :::10250          :::*          LISTEN
125536/kubelet
```

NOTE: The kube-proxy service listens on random ports. This is not a problem on RHEL Atomic systems, since there is on filtering firewall service used by default. However, if you add a firewall to RHEL Atomic or use a default RHEL system, you can request that kube-proxy listen on specific ports in the service definition and then open those ports in the firewall.

Here is **netstat** output on a Kubernetes master:

```
tcp        0      0 192.168.122.249:7080  0.0.0.0:* LISTEN   636/kube-
apiserver
tcp6       0      0 :::8080               :::*      LISTEN
636/kube-apiserver
tcp        0      0 127.0.0.1:10252       0.0.0.0:* LISTEN
7541/kube-controller
tcp        0      0 127.0.0.1:10251       0.0.0.0:* LISTEN
7590/kube-scheduler
tcp6       0      0 :::4001               :::*      LISTEN   941/etcd
tcp6       0      0 :::7001               :::*      LISTEN   941/etcd
```

Look at the output in the third column. That shows you the IP addresses (

represents all interfaces) and port number that each service is listening on. Make sure you open ports to each of those services.

4.5.3. Checking Kubernetes yaml or json Files

You set up your Kubernetes environment (pods, services, and replication controllers) by loading information from yaml or json files using the **kubectl create** command. Failures can result from those files being improperly formatted or missing needed information.

The following sections contain tips for fixing problems that occur from broken yaml or json files.

4.5.3.1. Troubleshooting Kubernetes Service Creation

A Kubernetes service (created with **kubectl**), attaches an IP address and port to a label. A pod that needs to use that service can refer to that service by the label, so it doesn't need to know the IP address and port numbers directly. The following is an example of a service file named db-service.yaml, followed by a list of problems that can occur when you try to create a service:

```
id: "db-service"
kind: "Service"
apiVersion: "v1"
port: 3306
portalIP: "10.254.100.1"
selector:
  name: "db"
labels:
  name: "db"
```

```
# kubectl create -f db-service
# kubectl get services
NAME          LABELS                                    SELECTOR  IP
PORT
db-service    name=db                                   name=db
10.254.100.1   3306
kubernetes-ro component=apiserver,provider=kubernetes     10.254.186.33
```

```
80
kubernetes    component=apiserver,provider=kubernetes     10.254.198.9
443
```

NOTE: If you don't see the kubernetes-ro and kubernetes services, try restarting the kube-scheduler systemd service (**systemctl restart kube-scheduler.service**).

If you don't see output similar to what was just shown, read the following:

- If the service seemed to create successfully, but the LABELS an SELECTOR were not set, the output might look as follows:

  ```
  # kubectl get services
  NAME         LABELS     SELECTOR  IP             PORT
  db-service             10.254.100.1   3306
  ```

  Check that the name: fields under selector: and labels: are each indented two spaces. In this case I deleted the two blank spaces before each **name: "db"** line and their values were not used by **kubectl**.

- If you forget that you have already created a Service and try to create it again or if some other service has already allocated an IP address you identified in your service yaml, your new attempt to create the service will result in this message:

  ```
  create.go:75] service "webserver-service" is invalid: spec.portalIP:
      invalid value '10.254.100.50': IP 10.254.100.50 is already
  allocated
  ```

  You can either use a different IP address or stop the service that is currently consuming that port, if you don't need that service.

- The following error noting that the "Service" object isn't registered can occur for a couple of reasons:

  ```
  7338 create.go:75] unable to recognize "db-service.yaml": no object
  named "Services" is registered
  ```

  In the above example, "Service" was misspelled as "Services". If it does correctly say "Service", then check that the apiVersion is correct. A similar error occurred when the invalid value "v99" was used as the apiVersion. Instead of saying "v99" doesn't exist, it says it can't find the object "Service".

    - Here is a list of error messages that occur if any of the fields from the above example is missing:

      - **id: missing**: service "" is invalid: name: required value ''
      - **kind: missing**: unable to recognize "db-service.yaml": no object named "" is registered
      - **apiVersion: missing**: service "" is invalid: [name: required value '', spec.port: invalid value '0']
      - **port: missing**: service "db-service" is invalid: spec.port: invalid value '0'
      - **portalIP: missing**: No error is reported because portalIP is not required

- **selector: missing**: No error is reported, but SELECTOR field is missing and service may not work.
- **labels: missing**: Not an error, but LABELS field is missing and service may not work.

4.5.3.2. Troubleshooting Kubernetes Replication Controller and Pod creation

A Kubernetes Pod lets you associate one or more containers together, assign run options to each container, and manage those containers together as a unit. A replication controller lets you designate how many of the pods you identify should be running. The following is an example of a yaml file that defines a Web server pod and a replications controller that ensures that two instances of the pod are running.

```
id: "webserver-controller"
kind: "ReplicationController"
apiVersion: "v1"
metadata:
  name: "webserver-controller"
spec:
  replicas: 1
  selector:
    name: "webserver"
  template:
    spec:
      containers:
        - name: "apache-frontend"
          image: "webwithdb"
          ports:
            - containerPort: 80
    metadata:
      labels:
        name: "webserver"
        uses: db
  labels:
    name: "webserver"
```

```
# kubectl create -f webserver-service.yaml
# kubectl get pods
POD         IP          CONTAINER(S)     IMAGE(S)    HOST
       LABELS                                          STATUS
f28980d...  10.20.48.4  apache-frontend  webwithdb   node1.example.com/
       name=webserver,selectorname=webserver,uses=db      Running
f28a0a8...  10.20.29.9  apache-frontend  webwithdb   node2.example.com/
       name=webserver,selectorname=webserver,uses=db      Running
# kubectl get replicationControllers
CONTROLLER           CONTAINER(S)    IMAGE(S)   SELECTOR
REPLICAS

webserver-controller apache-frontend webwithdb  selectorname=webserver
2
```

NOTE: I truncated the pod name and wrapped the long lines in the output above.

If you don’t see output similar to what was just shown, read the following:

- **id: missing**: If a generated set of numbers and letters appears in the CONTROLLER column instead of "webserver-controller", your yaml file is probably missing the id line.
- **apiVersion set wrong**: If you see the message "unable to recognize "webserver-rc.yaml": no object named "ReplicationController" is registered", you may have an invalid apiVersion value or misspelled ReplicationController.
- **selectorname: missing**: If you see the message "replicationController "webserver-controller" is invalid: spec.selector: required value 'map[]'", there is no selectorname set after the replicaSelector line. If the selectorname is not indented properly, you will see a message like, "unable to get type info from "webserver-rc.yaml": couldn't get version/kind: error converting YAML to JSON: yaml: line 7: did not find expected key."

4.6. TROUBLESHOOTING TECHNIQUES

If you want to look deeper into what is going on with your Kubernetes cluster, see the following techniques for investigating further.

4.6.1. Crawling and fixing the etcd database

The etcd service provides the database that Kubernetes uses to coordinate information across the cluster. There are ways to view the database directly and fix problems in it (or clear the database if it is beyond repair).

Displaying data from the etcd database: You can query most information you need from the etcd database using kubectl get commands. However, if this database seems to be inconsistent with the way you believe your configuration should be, you can directly query the etcd database using the etcdctl command.

Use the **etcdctl** command with the ls option to list the directory structure of the database. To get values, use the get option. For example, to see the root of the database, type the following:

```
# etcdctl ls /
/coreos.com
/registry
```

To list Kubernetes events in the etcd database, type this:

```
# etcdctl ls /registry/events/default/

/registry/events/default/kube-apiserver-
master.example.com.141b97ea661dd5d6
/registry/events/default/kube-controller-manager-
master.example.com.141b97ea66acbffe
/registry/events/default/kube-apiserver-
master.example.com.141b97ea661dd5d6
...
```

To see the data associated with a particular event, use the **get** option (replace the **kube-apiserver** argument with an entry from your /registry/events/default/ directory):

```
# etcdctl get \
  /registry/events/default/kube-apiserver-
master.example.com.141b97ea661dd5d6 | \
  python -mjson.tool
```

```
{
    "apiVersion": "v1",
    "count": 1374,
    "firstTimestamp": "2015-11-30T21:25:38Z",
    "involvedObject": {
        "apiVersion": "v1",
        "fieldPath": "spec.containers{kube-apiserver}",
        "kind": "Pod",
        "name": "kube-apiserver-master.example.com",
        "namespace": "default",
        "uid": "7a5070c30bf6d637383c8a8cd96ade4e"
    },
    "kind": "Event",
    "lastTimestamp": "2015-12-01T01:14:45Z",
    "message": "Failed to pull image \"rhel/kubernetes-apiserver\":
       Error: image rhel/kubernetes-apiserver:latest not found",
    "metadata": {
        "creationTimestamp": "2015-12-01T01:14:45Z",
        "name": "kube-apiserver-master.example.com.141b97ea661dd5d6",
        "namespace": "default",
        "resourceVersion": "14562",
        "uid": "e841fdad-97c8-11e5-98bb-52540016f96d"
...
```

The output above is piped to a python json.tool formatting module, to make it easier to read. You can see that kube-apiserver is defined as a Pod and that an event associated with it failed because the wrong name was given to the container.

Here are a few other interesting **ls** examples with **etcdctl**. These show the top-level entries in the registry, the names of the minions, the names of the services, and the subnets in use:

```
# etcdctl ls /registry/
/registry/serviceaccounts
/registry/services
/registry/events
/registry/minions
/registry/namespaces
/registry/pods
/registry/ranges
# etcdctl ls /registry/minions
/registry/minions/master.example.com
/registry/minions/node1.example.com
/registry/minions/node2.example.com
# etcdctl ls /registry/services/endpoints/default/
/registry/services/endpoints/default/db-service
/registry/services/endpoints/default/kubernetes
/registry/services/endpoints/default/kubernetes-ro
/registry/services/endpoints/default/webserver-service
# etcdctl ls /coreos.com/network/subnets/
/coreos.com/network/subnets/10.20.1.0-24
/coreos.com/network/subnets/10.20.24.0-24
/coreos.com/network/subnets/10.20.56.0-24
```

Here are a few **get** examples with **etcdctl**. The first lets you see the host IP address and other information about node1. The next example shows information on a subnetwork assigned by flannel:

```
# etcdctl get /registry/minions/node1.example.com | python -mjson.tool

{
    "apiVersion": "v1",
    "kind": "Node",
    "metadata": {
        "creationTimestamp": "2015-11-30T19:17:47Z",
        "labels": {
            "kubernetes.io/hostname": "node1.example.com"
        },
        "name": "node1.example.com",
        "resourceVersion": "15734",
        "selfLink": "/api/v1/nodes/node1.example.com",
        "uid": "0aaf53f8-9797-11e5-b4e2-52540016f96d"
    },
    "spec": {
        "externalID": "node1.example.com"
    },
    "status": {
        "addresses": [
            {
                "address": "192.168.122.227",
                "type": "LegacyHostIP"
            }
        ],
        "capacity": {
            "cpu": "1",
            "memory": "1999956Ki",
            "pods": "40"
        },
...

# etcdctl get /coreos.com/network/subnets/10.20.49.0-24 | python -
mjson.tool
{
    "BackendData": {
        "VtepMAC": "2a:be:2d:76:47:b6"
    },
    "BackendType": "vxlan",
    "PublicIP": "192.168.122.65"
}
```

NOTE: If etcdctl is not available, you can use the **curl** command instead. For example, to see the root of the database with **curl**, use this instead of the **etcdctl ls /** command: **curl -L http://master:2379/v2/keys/ | python -mjson.tool**. Use that form of the curl command to display both directory and key values. If you believe that a node is not able to connect to the etcd service on the master, you could use the following **curl** command to test that connection from the node:

```
# curl -s -L http://master.example.com:2379/version
{"etcdserver":"2.1.1","etcdcluster":"2.1.0"}
```

Fixing the etcd database: It is possible to correct problems with your etcd database if information gets out of sync. There is are **etcdctl update** and **etcdctl set** commands for changing the contents of a key. However, if you are not careful, changing these values can cause more problems than they fix.

However, if your etcd database become completely unuseable, you can clear it and start over again. The way to do that is to run the **etcd** daemon with the **-f** option.

WARNING: Before you clear the etcd database, try using **kubectl delete** command to try to remove the offending services, pods, replicationControllers or minions. If you still feel you need to clear the database, keep in mind that if you do so, you need to recreate everything from scratch.

To clear the etcd database, type the following:

```
# etcd -f
```

4.6.2. Deleting Kubernetes components

How you stop and delete components in Kubernetes matters. Because Kubernetes is designed to get things to a particular state, simply deleting a container or a pod will often just cause another one to be started. So, if you need to delete some or all of the components in your Kubernetes environment, I recommend you following the instructions in the "Cleaning Up Kubernetes" section in the following article: Creating a Kubernetes Cluster: Cleaning up Kubernetes.

If you do delete components out of order, here's what you can expect:

- **I deleted a pod, but it started up again**: If you don't stop the replication controllers first, the pods will be restarted. Stop the replication controllers (**docker stop replicationControllers webserver-controller**), then stop the pods.
- **I stopped and deleted a container, but it started up again**: With a Kubernetes cluster, you should not stop a container directly with **docker stop**. The replication controller will start a new container to restart the one you stopped.

4.6.3. Pods Stuck in the "WAITING" state.

PODS can be stuck in the waiting state for some time period. Here are some possible causes:

- **Pulling the Docker image is taking a while**: To confirm this, you can ssh directly into the minion which the pod is assigned, and run:

  ```
  # journalctl -f -u docker
  ```

  This should show logs of docker pulling down your image. Note requests to pull dockerhub images may fail intermittently, but the kubelets will continue retrying.

- **PODs are unassigned**: If a pod remains unassigned, confirm that nodes are available to the master by running kubectl get minions. It is possible that the node may just be down or otherwise unreachable. Unassigned pods can also result from setting the replication count higher then what the cluster can provide.
- **Container Pod dies right after starting**: In some cases, if the Dockerfile you created is not written properly to start a service, or the docker CMD operation is failing, you might see the POD immediately dying after it starts. Try testing the container image with a **docker run** command, to make sure that the container itself isn't broken.
- **Check output from container**: Messages output from containers can be viewed with the kubectl log command. This can be useful for debugging problems with the applications running within the container. Here is how to list available pods and view log messages for the one you want:

```
# kubectl get pods
POD                                    IP         CONTAINER(S)
IMAGE(S)            HOST                LABELS
STATUS
e1f4b268-e87d-11e4-926b-5254001aa4ee  10.20.24.3 db
dbforweb node1.example.com/   name=db,selectorname=db
Running
# kubectl log e1f4b268-e87d-11e4-926b-5254001aa4ee
2015-04-28T16:09:36.953130209Z 150428 12:09:36 mysqld_safe Logging
to '/var/log/mariadb/mariadb.log'.
2015-04-28T16:09:37.137064742Z 150428 12:09:37 mysqld_safe Starting
mysqld daemon with databases from /var/lib/mysql
```

- **Check container output from docker**: Some errors don't percolate all the way up to the kubelet. You can look directly in the docker logs for an exited container to observe why this might be happening. Here's how:

    - Log into the node that's having trouble running a container
    - Run this command to look for an exited run:

    ```
    # docker ps -a
    61960bda2927  rhel7/rhel-tools:latest "/usr/bin/bash" 47 hours
    ago
          Exited (0) 43 hours ago      myrhel-tools4
    ```

    - Check all the output from the container with docker logs:

    ```
    # docker logs 61960bda2927
    ```

You should be able to see the entire output from the container session. So, for example, if you opened a shell in a container, you will see all the commands you ran from that shell when you run docker logs.

CHAPTER 5. YAML IN A NUTSHELL

5.1. OVERVIEW

YAML — which stands for “YAML Ain’t Markup Language” — is a human-friendly data serialization standard, similar in scope to JSON (Javascript Object Notation). Unlike JSON, there are only a handful of special characters used to represent mappings and bullet lists, the two basic types of structure, and indentation is used to represent substructure.

5.2. BASICS

The YAML format is line-oriented, with two top-level parts, *HEAD* and *BODY*, separated by a line of three hyphens.

```
HEAD
---
BODY
```

The head holds configuration information and the body holds the data. this topic does not discuss the configuration aspect; all the examples here show only the data portion. In such cases, the “---” is optional.

The most basic data element is one of:

1. A number
2. A Unicode string
3. A boolean value, spelled either true or false
4. In a key/value pair context, a missing value is parsed as nil

Comments start with a **“#”** (hash, U+23) and go to the end of the line.

Indentation is whitespace at the start of the line. You are strongly encouraged to avoid **TAB** (U+09) characters and use a series of **SPACE** (U+20) characters, instead.

5.3. LISTS

A list is a series of lines, each beginning with the same amount of indentation, followed by a hyphen, followed by a list element. Lists cannot have blank lines. For example, here is a list of three elements, the third of which has a comment:

```
- top shelf
- middle age
- bottom dweller   # stability is important
```

Note: The third element is the string “bottom dweller” and does not include the whitespace between “dweller” and the comment.

WARNING: Lists cannot normally nest directly; there should be an intervening mapping (described below). In the following example, the list’s second element seems, due to the indentation (two **SPACE** characters), to host a sub-list:

```
- top
- middle
  - highish middle
  - lowish middle
- bottom
```

In reality, the second element is actually parsed as a single string. The input is equivalent to:

```
- top
- middle - highish middle - lowish middle
- bottom
```

The newlines and indentation are normalized to a single space.

5.4. MAPPINGS

To write a mapping (also known as an associative array or hash table), use a **":"** (colon, **U+3A**) followed by one or more **SPACE** characters between the key and the value:

```
square:   4
triangle: 3
pentagon: 5
```

All keys in a mapping must be unique. For example, this is invalid YAML for two reasons: the key square is repeated, and there is no space after the colon following **triangle**:

```
square:  4
triangle:3       # invalid key/value separation
square:  5       # repeated key
```

Mappings can nest directly, by starting the sub-mapping on the next line with increased indentation. In the next example, the value for key **square** is itself a mapping (keys **sides** and **perimeter**), and likewise for the value for key **triangle**. The value for key **pentagon** is the number 5.

```
square:
  sides:     4
  perimeter: sides * side-length
triangle:
  sides:     3
  perimeter: see square
pentagon:    5
```

The following example shows a mapping with three key/value pairs. The first and third values are **nil**, while the second is a list of two elements, “highish middle” and “lowish middle”.

```
top:
middle:
  - highish middle
  - lowish middle
bottom:
```

5.5. QUOTATION

Double-quotation marks (also known as “double-quotes”) are useful for forcing non-string data to be interpreted as a string, for preserving whitespace, and for suppressing the meaning of colon. To include a double-quote in a string, escape it with **`“\”** (backslash, **U+5C**). In the following example, all keys and values are strings. The second key has a colon in it. The second value has two spaces both preceding and trailing the visible text.

```
"true" : "1"
"key the second (which has a \":\" in it)" : "  second  value  "
```

For readability when double-quoting the key, you are encouraged to add whitespace before the colon.

5.6. BLOCK CONTENT

There are two kinds of block content, typically found in the value position of a mapping element: newline-preserving and folded. If a block begins with **“|”** (pipe, **U+7C**), the newlines in that block are preserved. If it begins with **“>”** (greater-than, **U+3E**), consecutive newlines are folded into a single space. The following example shows both kinds of block content as the values for keys **good-bye** and **anyway**.

```
hello: world

good-bye: |
    first line

    third
    fourth and last


anyway: >
    nothing is guaranteed
    in life
lastly:
```

Using \n (backslash-n) to indicate newline, the values for keys **good-bye** and **anyway** are, respectively:

```
first line\n\nthird\nfourth and last\n

nothing is guaranteed in life\n
```

Note that the newlines are preserved in the **good-bye** value but folded into a single space in the **anyway** value. Also, each value ends with a single newline, even though there are two blank lines between “fourth and last” and “anyway”, and no blank lines between “in life” and “lastly”.

5.7. COMPACT REPRESENTATION

Another, more compact, way to represent lists and mappings is to begin with a start character, finish with an end character, and separate elements with **“,”** (comma, **U+2C**).

For lists, the start and end characters are **“[”** (left square brace, **U+5B**) and **“]”** (right square brace, **U+5D**), respectively. In the following example, the values in the mapping are identical:

```
one:
  - echo
  - hello, world!
two: [ echo, "hello, world!" ]
```

Note: The double-quotes around the second list element of the second value; they prevent the comma from being misinterpreted as an element separator. (If we remove them, the list would have three elements: "echo", "hello" and "world!".)

For mappings, the start and end characters are **“{”** (left curly brace, **U+7B**) and **“}”** (right curly brace, **U+7D**), respectively. In the following example, the values of both one and two are identical:

```
one:
  roses: red
  violets: blue

two: { roses: red, violets: blue }
```

5.8. ADDITIONAL INFORMATION

There is much more to YAML, not described in this topic: directives, complex mapping keys, flow styles, references, aliases, and tags. For detailed information, see the official YAML site, specifically the latest (version 1.2 at time of writing) specification.

CHAPTER 6. GET STARTED PROVISIONING STORAGE IN KUBERNETES

6.1. OVERVIEW

This section explains how to provision storage in Kubernetes.

Before undertaking the exercises in this topic, you must have Kubernetes set up.

If you do not have Kubernetes set up, follow the instructions in Get Started Orchestrating Containers with Kubernetes.

6.2. KUBERNETES PERSISTENT VOLUMES

This section provides an overview of Kubernetes Persistent Volumes. The example below explains how to use **`nginx`** to serve content from a persistent volume.

This section assumes that you understand the basics of Kubernetes and that you have a Kubernetes cluster up and running.

A Persistent Volume (PV) in Kubernetes represents a real piece of underlying storage capacity in the infrastructure. Before using Kubernetes to mount anything, you must first create whatever storage that you plan to mount. Cluster administrators must create their GCE disks and export their NFS shares in order for Kubernetes to mount them.

Persistent volumes are intended for "network volumes" like GCE Persistent Disks, NFS shares, and AWS ElasticBlockStore volumes. HostPath was included for ease of development and testing. You'll create a local HostPath for this example.

IMPORTANT! In order for HostPath to work, you will need to run a single node cluster. Kubernetes does not support local storage on the host at this time. There is no guarantee that your pod will end up on the correct node where the HostPath resides.

```
// this will be nginx's webroot
$ mkdir /tmp/data01
$ echo 'I love Kubernetes storage!' > /tmp/data01/index.html
```

Create physical volumes by posting them to the API server.

```
$ kubectl create -f examples/persistent-volumes/volumes/local-01.yaml
$ kubectl get pv

NAME        LABELS       CAPACITY            ACCESSMODES
STATUS              CLAIM
pv0001      map[]        10737418240         RWO
Available
```

6.2.1. Requesting storage

Users of Kubernetes request persistent storage for their pods. The nature of the underlying provisioning need not be known by users. Users must know that they can rely on their claims to storage and that they can manage that storage's lifecycle independently of the many pods that may use it.

Claims must be created in the same namespace as the pods that use them.

```
$ kubectl create -f examples/persistent-volumes/claims/claim-01.yaml
$ kubectl get pvc

NAME                LABELS              STATUS              VOLUME
myclaim-1           map[]
```

A background process will attempt to match this claim to a volume. The state of your claim will eventually look something like this:

```
$ kubectl get pvc

NAME        LABELS    STATUS    VOLUME
myclaim-1   map[]     Bound     f5c3a89a-e50a-11e4-972f-80e6500a981e


$ kubectl get pv

NAME                LABELS              CAPACITY
ACCESSMODES         STATUS    CLAIM
pv0001              map[]               10737418240         RWO
Bound     myclaim-1 / 6bef4c40-e50b-11e4-972f-80e6500a981e
```

6.2.2. Using your claim as a volume

Claims are used as volumes in pods. Kubernetes uses the claim to look up its bound PV. The PV is then exposed to the pod.

```
$ kubectl create -f examples/persistent-volumes/simpletest/pod.yaml

$ kubectl get pods

POD       IP           CONTAINER(S)   IMAGE(S)   HOST
LABELS    STATUS    CREATED
mypod     172.17.0.2   myfrontend     nginx      127.0.0.1/127.0.0.1
<none>    Running   12 minutes>


$ kubectl create -f examples/persistent-volumes/simpletest/service.json
$ kubectl get services

NAME              LABELS                                    SELECTOR
IP           PORT(S)
frontendservice   <none>
name=frontendhttp   10.0.0.241   3000/TCP
kubernetes        component=apiserver,provider=kubernetes   <none>
10.0.0.2     443/TCP
kubernetes-ro     component=apiserver,provider=kubernetes   <none>
10.0.0.1     80/TCP
```

6.2.3. Next steps

Query your service endpoint to see the content that nginx is now serving. If you get a "forbidden" errror, disable SELinux (# setenforce 0).

```
# curl 10.0.0.241:3000
I love Kubernetes storage!
```

6.3. VOLUMES

Kubernetes abstracts various storage facilities as "volumes".

Volumes are defined in the **volumes** section of a pod's definition. The source of the data in the volumes is either (1) a remote NFS share, (2) an iSCSI target, (3) an empty directory, or (4) a local directory on the host.

It is possible to define multiple volumes in the **volumes** section of the pod's definition. Each volume must have a unique name (within the context of the pod) that is used during the mounting procedure as a unique identifier within the pod.

These volumes, once defined, can be mounted into containers that are defined in the **containers** section of the pod's definition. Each container can mount several volumes; on the other hand, a single volume can be mounted into several containers. The **volumeMounts** section of the container definition specifies where the volume should be mounted.

6.3.1. Example

```
apiVersion: v1
kind: Pod
metadata:
  name: nfs-web
spec:
  volumes:
    # List of volumes to use, i.e. *what* to mount
    - name: myvolume
      < volume details, see below >
    - name: mysecondvolume
      < volume details, see below >

  containers:
    - name: mycontainer
      volumeMounts:
        # List of mount directories, i.e. *where* to mount
        # We want to mount 'myvolume' into /usr/share/nginx/html
        - name: myvolume
          mountPath: /usr/share/nginx/html/
        # We want to mount 'mysecondvolume' into /var/log
        - name: mysecondvolume
          mountPath: /var/log/
```

6.4. KUBERNETES AND SELINUX PERMISSIONS

Kubernetes, in order to function properly, must have access to a directory that is shared between the host and the container. SELinux, by default, blocks Kubernetes from having access to that shared directory. Usually this is a good idea: no one wants a compromised container to access the

host and cause damage. In this situation, though, we want the directory to be shared between the host and the pod without SELinux intervening to prevent the share.

Here's an example. If we want to share the directory **/srv/my-data** from the Atomic Host to a pod, we must explicitly relabel **/srv/my-data** with the SELinux label **svirt_sandbox_file_t**. The presence of this label on this directory (which is on the host) causes SELinux to permit the container to read and write to the directory. Here's the command that attaches the **svirt_sandbox_file_t** label to the **/srv/my-data** directory:

```
$ chcon -R -t svirt_sandbox_file_t /srv/my-data
```

The following example steps you through the procedure:

Step One

Define this container, which uses **/srv/my-data** from the host as the HTML root:

```
{
 "apiVersion": "v1",
 "kind": "Pod",
 "metadata": {
   "name": "host-test"
 },
 "spec": {
   "containers": [
     {
       "name": "host-test",
       "image": "nginx",
       "privileged": false,
       "volumeMounts": [
         {
           "name": "srv",
           "mountPath": "/usr/share/nginx/html",
           "readOnly": false
         }
       ]
     }
   ],
   "volumes": [
     {
       "name": "srv",
       "hostPath": {
         "path": "/srv/my-data"
       }
     }
   ]
 }
}
```

Step Two

Run the following commands on the container host to confirm that SELinux denies the nginx container read access to **/srv/my-data**:

```
$ mkdir /srv/my-data
$ echo "Hello world" > /srv/my-data/index.html
$ curl <IP address of the container>
```

You'll get the following output:

```
<html>
<head><title>403 Forbidden</title></head>
...
```

STEP THREE

Apply the label **svirt_sandbox_file_t** to the directory **/srv/my-data**:

```
$ chcon -R -t svirt_sandbox_file_t /srv/my-data
```

STEP FOUR

Use **curl** to access the container and to confirm that the label has taken effect:

```
$ curl <IP address of the container>
Hello world
```

If the **curl** command returned "Hello world", the SELinux label has been properly applied.

BZ#1222060[tracks this issue, if you're interested.

6.5. NFS

In order to test this scenario, you must already have prepared NFS shares. In this example, you will mount the NFS shares into a pod.

The following example mounts the NFS share into **/usr/share/nginx/html/** and runs the **nginx** webserver.

STEP ONE

Create a file named **nfs-web.yaml**:

```
apiVersion: v1
kind: Pod
metadata:
  name: nfs-web
spec:
  volumes:
    - name: www
      nfs:
        # Use real NFS server address here.
        server: 192.168.100.1
        # Use real NFS server export directory.
        path: "/www"
        readOnly: true
  containers:
    - name: web
      image: nginx
      ports:
        - name: web
          containerPort: 80
```

```
          protocol: tcp
      volumeMounts:
          # 'name' must match the volume name below.
          - name: www
            # Where to mount the volume.
            mountPath: "/usr/share/nginx/html/"
```

STEP TWO

Start the pod:

```
$ kubectl create -f nfs-web.yaml
```

Kubernetes mounts **`192.168.100.1:/www`** into **`/usr/share/nginx/html/`**` inside the nginx container and runs it.

STEP THREE

Confirm that the webserver receives data from the NFS share:

```
$ curl 172.17.0.6
Hello from NFS
```

Troubleshooting

403 Forbidden error: if you receive a "403 Forbidden" response from the webserver, make sure that SELinux allows Docker containers to read data over NFS by running the following command:

```
$ setsebool -P virt_use_nfs 1
```

6.6. ISCSI

STEP ONE Make sure that the iSCSI target is properly configured. Make sure that all Kubernetes nodes have sufficient privileges to attach a LUN from the iSCSI target.

STEP TWO Create a file named **`iscsi-web.yaml`**, containing the following pod definition:

```
apiVersion: v1
kind: Pod
metadata:
  name: iscsi-web
spec:
  volumes:
    - name: www
      iscsi:
        # Address of the iSCSI target portal
        targetPortal: "192.168.100.98:3260"
        # IQN of the portal
        iqn: "iqn.2003-01.org.linux-iscsi.iscsi.x8664:sn.63b56adc495d"
        # LUN we want to mount
        lun: 0
        # Filesystem on the LUN
        fsType: ext4
        readOnly: false
  containers:
```

```
    - name: web
      image: nginx
      ports:
        - name: web
          containerPort: 80
          protocol: tcp
      volumeMounts:
          # 'name' must match the volume name below.
          - name: www
            # Where to mount he volume.
            mountPath: "/usr/share/nginx/html/"
```

STEP THREE

Create the pod:

```
$ kubectl create -f iscsi-web.yaml
```

STEP FOUR

Kubernetes logs in to the iSCSI target, attaches LUN 0 (typically as **`/dev/sdXYZ`**), mounts the filesystem specified (in our example, it's ext4) to **`/usr/share/nginx/html/`** inside the nginx container, and runs it.

STEP FIVE

Check that the web server uses data from the iSCSI volume:

```
$ curl 172.17.0.6
Hello from iSCSI
```

6.7. GOOGLE COMPUTE ENGINE

Google Compute Engine Persistent Disk (GCE PD)

If you are running your cluster on Google Compute Engine, you can use a Persistent Disk as your persistent storage source. In the following example, you will create a pod which serves html content from a GCE PD.

STEP ONE

If you have the GCE SDK set up, create a persistent disk using the following command:

```
$ gcloud compute disks create --size=250GB {Persistent Disk Name}
```

Otherwise you can create the disk through the GCE web interface. If you want to set up the GCE SDK follow the instructions here.

STEP TWO

Create a file named **`gce-pd-web.yaml`**:

```
apiVersion: v1
kind: Pod
metadata:
  name: gce-web
```

```
spec:
  containers:
    - name: web
      image: nginx
      ports:
        - name: web
          containerPort: 80
          protocol: tcp
      volumeMounts:
        - name: html-pd
          mountPath: "/usr/share/nginx/html"
  volumes:
    - name: html-pd
      gcePersistentDisk:
        # Add the name of your persistent disk below
        pdName: {Persistent Disk Name}
        fsType: ext4
```

STEP THREE

Create the pod:

```
$ kubectl create -f gce-pd-web.yaml
```

Kubernetes will create the pod and attach the disk but it will not format and mount it. This is due to a bug which will be fixed in future versions of Kubernetes. To work around this proceed to the next step.

STEP FOUR

Format and mount the persistent disk.

STEP FIVE

The disk will be attached to the virtual machine and a device will appear under **`/dev/disk/by-id/`** with the name **`scsi-0Google_PersistentDisk_{Persistent Disk Name}`**. If this disk is already formatted and contains data proceed to the next step otherwise run the following command as root to format it:

```
$ mkfs.ext4 /dev/disk/by-id/scsi-0Google_PersistentDisk_{Persistent
Disk Name}
```

STEP SIX

When the disk is formatted, mount it in the location expected by Kubernetes. Run the following commands as root:

```
# mkdir -p /var/lib/kubelet/plugins/kubernetes.io/gce-
pd/mounts/{Persistent Disk Name} && mount /dev/disk/by-id/scsi-
0Google_PersistentDisk_{Persistent Disk Name}
/var/lib/kubelet/plugins/kubernetes.io/gce-pd/mounts/{Persistent Disk
Name}
```

Note: The mkdir command and the mount command must be run in quick succession as above because Kubernetes clean up will remove the directory if it sees nothing mounted there.

STEP SEVEN

Now that the disk is mounted it must be given the correct SELinux context. As root run the following:

```
$ sudo chcon -R -t svirt_sandbox_file_t
/var/lib/kubelet/plugins/kubernetes.io/gce-pd/mounts/{Persistent Disk
Name}
```

STEP EIGHT

Create some data for your webserver to serve:

```
$ echo "Hello world" >  /var/lib/kubelet/plugins/kubernetes.io/gce-
pd/mounts/{Persistent Disk Name}/index.html
```

STEP NINE

You should now be able to get HTML from the pod:

```
$ curl {IP address of the container}
Hello World!
```

CHAPTER 7. GET STARTED WITH DOCKER FORMATTED CONTAINER IMAGES

7.1. OVERVIEW

Docker has quickly become one of the premier projects for containerizing applications. This topic provides a hands-on approach to start using Docker in Red Hat Enterprise Linux 7 and RHEL Atomic by setting up a Docker registry, getting and using Docker images, and working with Docker containers.

7.2. BACKGROUND

The Docker project provides the means of packaging applications in lightweight containers. Running applications within Docker containers offers the following advantages:

- **Smaller than Virtual Machines**: Because Docker images contain only the content needed to run an application, saving and sharing is much more efficient with Docker containers than it is with virtual machines (which include entire operating systems)
- **Improved performance**: Likewise, since you are not running an entirely separate operating system, a container will typically run faster than an application that carries with it the overhead of a whole new virtual machine.
- **Secure**: Because a Docker container typically has its own network interfaces, file system, and memory, the application running in that container can be isolated and secured from other activities on a host computer.
- **Flexible**: With an application's run time requirements included with the application in the container, a Docker container is capable of being run in multiple environments.

Currently, you can run Docker containers on Red Hat Enterprise Linux 7 (RHEL 7) and Red Hat Enterprise Linux Atomic (based on RHEL 7) systems. If you are unfamiliar with RHEL Atomic, you can learn more about it from Getting Started with Red Hat Enterprise Linux Atomic Host or the upstream Project Atomic site. Project Atomic produces smaller derivatives of RPM-based Linux distributions (RHEL, Fedora, and CentOS) that is made specifically to run Docker containers in OpenStack, VirtualBox, Linux KVM and several different cloud environments.

This topic will help you get started with the initial release of Docker in RHEL 7 and RHEL Atomic. Besides offering you some hands-on ways of trying out Docker, it also describes how to:

- Access RHEL-based Docker images from the Red Hat Customer Portal
- Incorporate RHEL-entitled software into your containers

Later releases of this topic will help you:

- Leverage RHEL security features to ensure safe deployment of your containers
- Find tools to standardize how you build Docker images
- Offer tips for building containers in ways that are compliant with security protocols

If you are interested in more details on how Docker works, refer to the following:

- **Release Notes**: Refer to the Linux Containers with Docker Format section of the RHEL 7 Release Notes for an overview of Docker features in RHEL 7.

- **Docker Project Site**: From the Docker site, you can learn about Docker from the What is Docker? page and the Getting Started page. There is also a Docker Documentation page you can refer to.
- **Docker README**: After you install the docker package, refer to the README.md file in the /usr/share/doc/docker-1* directory.
- **Docker man pages**: Again, with docker installed, type **`man docker`** to learn about the docker command. Then refer to separate man pages for each docker option (for example, type **`man docker-image`** to read about the **`docker image`** option).

NOTE: Currently, to run the docker command in RHEL 7 and RHEL Atomic you must have root privilege. In the procedure, this is indicated by the command prompt appearing as a hash sign (#). Configuring sudo will work, if you prefer not to log in directly to the root user account.

7.3. GETTING DOCKER IN RHEL 7

To get an environment where you can develop Docker containers, you can install a Red Hat Enterprise Linux 7 system to act as a development system as well as a container host. The docker package itself is stored in a RHEL Extras repository (see the Red Hat Enterprise Linux Extras Life Cycle article for a description of support policies and life cycle information for the Red Hat Enterprise Linux Extras channel).

Using the RHEL 7 subscription model, if you want to create Docker images or containers, you must properly register and entitle the host computer on which you build them. When you use **yum install** within a container to add packages, the container automatically has access to entitlements available from the RHEL 7 host, so it can get RPM packages from any repository enabled on that host.

1. **Install RHEL Server edition**: If you are ready to begin, you can start by installing a Red Hat Enterprise Linux system (Server edition) as described in the following: Red Hat Enterprise Linux 7 Installation Guide
2. **Register RHEL**: Once RHEL 7 is installed, register the system using Subscription Management tools and install the docker package. Also enable the software repositories needed. (Replace pool_id with the pool ID of your RHEL 7 subscription.) For example:

   NOTE: For this topic, we show the docker and docker-registry services running on the same host system. It is possible, in fact likely when the Docker registry is being used by multiple clients, to have docker-registry installed and running on a separate system. If that is the case, the docker-registry package is not required on a system running docker.

   ```
   # subscription-manager register --username=rhnuser --
   password=rhnpasswd
   # subscription-manager list --available  Find pool ID for RHEL
   subscription
   # subscription-manager attach --pool=pool_id
   # subscription-manager repos --enable=rhel-7-server-extras-rpms
   # subscription-manager repos --enable=rhel-7-server-optional-rpms
   ```

   NOTE: For information on the channel names required to get docker packages for Red Hat Satellite 5, refer to Satellite 5 repo to install Docker on Red Hat Enterprise Linux 7.
3. **Install docker and docker-registry**: Install the docker package and, optionally, the docker-registry. (If not already installed, install device-mapper-libs and device-mapper-event-libs as well.)

```
# yum install docker docker-registry
# yum install device-mapper-libs device-mapper-event-libs
```

4. **Start docker**:

```
# systemctl start docker.service
```

5. **Enable docker**:

```
# systemctl enable docker.service
```

6. **Check docker status**:

```
# systemctl status docker.service
    docker.service - Docker Application Container Engine
       Loaded: loaded (/usr/lib/systemd/system/docker.service;
enabled)
       Active: active (running) since Thu 2014-10-23 11:32:11
EDT; 14s ago
         Docs: http://docs.docker.io
     Main PID: 2068 (docker)
       CGroup: /system.slice/docker.service
               └─2068 /usr/bin/docker -d --selinux-enabled -H
fd://
    ...
```

With the docker service running, you can obtain some Docker images and use the **docker** command to begin working with Docker images in RHEL 7.

7.4. GETTING DOCKER IN RHEL ATOMIC

RHEL Atomic is a light-weight Linux operating system distribution that was designed specifically for running containers. It contains the docker service, as well as some services that can be used to orchestrate and manage Docker containers, including Kubernetes and Etcd services.

Because RHEL Atomic is more like an appliance than a full-featured Linux system, it is not made for you to install RPM packages or other software on (other than the containers that you add into the system).

RHEL Atomic has a mechanism for updating existing packages, but not for allowing users to add new packages. Therefore, you should consider using a standard RHEL 7 server system to develop your applications (so you can add a full compliment of development and debugging tools), then use RHEL Atomic to deploy your containers into a variety of virtualization and cloud environment.

That said, you can install a RHEL Atomic system and use it to run, build, stop, start, and otherwise work with containers using the examples shown in this topic. To do that, use the following procedure to get and install RHEL Atomic.

1. **Get RHEL Atomic**: RHEL Atomic is available from the Red Hat Customer Portal. You have the option of running RHEL Atomic as a live image (in .qcow2 format) or installing RHEL Atomic from an installation medium (in .iso format). You can get RHEL Atomic in those (and other formats) from here:

   RHEL Atomic Host Downloads

Then follow the Getting Started with Red Hat Enterprise Linux Atomic Host instructions for setting up Atomic to run in one of several different virtual environments.

2. **Register RHEL Atomic**: Once RHEL Atomic is installed, register the system using Subscription Management tools. (This will allow you to run **atomic upgrade** to upgrade Atomic software, but it won't let you install additional packages using the yum command.) For example:

```
# subscription-manager register --username=rhnuser --
password=rhnpasswd --auto-attach
```

IMPORTANT: Running containers with the docker command, as described in this topic, does not specifically require you to register the RHEL Atomic system and attach a subscription. However, if you want to run **yum install** commands within a container, the container must get valid subscription information from the RHEL Atomic host or it will fail. If you need to enable repositories other than those enabled by default with the RHEL version the host is using, you should edit the **/etc/yum.repos.d/redhat.repo** file. You can do that manually within the container and set enabled=1 for the repository you want to use. You can also use **yum-config-manager**, a command-line tool for managing Yum repo files. You can use the following command to enable repos:

```
# yum-config-manager --enable REPOSITORY

You can also use **yum-config-manager** to display Yum global options,
add repositories and others. **yum-config-manager** is documented in
detail
in the Red Hat Enterprise Linux 7 System Administrator's Guide. Since
**redhat.repo** is a big file and editing it manually can be error
prone,
it is recommended to use **yum-config-manager**.
```

1. **Start using Docker**: RHEL Atomic comes with the docker package already installed and enabled. So, once you have logged in and subscribed your Atomic system, here is the status of docker and related software:

   - You can immediately begin running the docker command to work with Docker images and containers.
   - The docker-registry package is not installed. If you want to be able to pull and push images between your Atomic system and a private registry, you can install the docker-registry package on a RHEL 7 system (as described next) and access that registry to store your own container images.
   - The kubernetes package, used to orchestrate Docker containers, is installed on RHEL Atomic, but it is not enabled by default. You need to enable and start several Kubernetes-related services to be able to orchestrate containers in RHEL Atomic with Kubernetes.

7.5. WORKING WITH DOCKER REGISTRIES

A Docker registry provides a place to store and share docker containers that are saved as images that can be shared with other people. With the docker package available with RHEL and RHEL Atomic, you can pull images from the Red Hat Customer Portal and push or pull images to and from your own private registry. You see what images are available to pull from the Red Hat Customer Portal (using **docker pull**) by searching the Red Hat Container Images Search Page.

This section describes how to start up a local registry, load Docker images to your local registry, and use those images to start up docker containers.

7.5.1. Creating a private Docker registry

One way to create a private Docker registry is to use the docker-registry service. If you installed the docker-registry package in RHEL 7 (it's not available in Atomic) as described earlier in this topic, you can enable and start the service as follows:

1. **Enable and start the docker-registry service**: Type the following to enable, start and check the status of the docker-registry service:

```
# systemctl enable docker-registry
# systemctl start docker-registry
# systemctl status docker-registry
docker-registry.service - Registry server for Docker
   Loaded: loaded (/usr/lib/systemd/system/docker-
registry.service; enabled)
   Active: active (running) since Thu 2014-10-23 13:40:26 EDT; 4s
ago
Main PID: 21031 (gunicorn)
   CGroup: /system.slice/docker-registry.service
           ├─21031 /usr/bin/python /usr/bin/gunicorn --access-
logfile - --debug ...
   ...
```

2. **Registry firewall issues**: The docker-registry service listens on TCP port 5000, so access to that port must be open to allow clients outside of the local system to be able to use the registry. This applies regardless of whether you are running docker-registry and docker or the same system or on different systems. You can open TCP port 5000 follows:

```
# firewall-cmd --zone=public --add-port=5000/tcp
# firewall-cmd --zone=public --add-port=5000/tcp --permanent
# firewall-cmd --zone=public --list-ports
5000/tcp
```

   or if have enabled a firewall using iptables firewall rules directly, you could find a way to have the following command run each time you boot your system:

```
iptables -A INPUT -m state --state NEW -m tcp -p tcp --dport 5000
-j ACCEPT
```

7.5.2. Getting images from remote Docker registries

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

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

The *<registry>* is a host that provides the docker-registry service on TCP *<port>* (default: 5000). Together, *<namespace>* and *<name>* identify a particular image controlled by *<namespace>* at that registry. Some registries also support raw *<name>*; for those, *<namespace>* is optional. When it is included, however, the additional level of hierarchy that *<namespace>* provides is useful to

distinguish between images with the same *<name>*. For example:

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

As of December, 2014, the only Docker registry that Red Hat supports is the one at registry.access.redhat.com. If you have access to a Docker image that is stored as a tarball, you can load that image into your Docker registry from your local file system.

docker pull: Use the pull option to pull an image from a remote registry. To pull the rhel base image from the Red Hat registry, type **docker pull registry.access.redhat.com/rhel**. To make sure that the image originates from the Red Hat registry, type the hostname of the registry, a slash, and the image name. The following command demonstrates this and pulls the **rhel** image from the Red Hat registry:

```
# docker pull registry.access.redhat.com/rhel
```

An image is identified by a repository name and a tag. The repository name **rhel**, when passed to the **docker pull** command without the name of a registry preceding it, is ambiguous and could result in the retrieval of an image that originates from an untrusted registry. To be more specific, you could add a tag, such as **latest** to form a name such as **rhel:latest**.

To see the images that resulted from the above **docker pull** command, type **docker images**:

```
# docker images
REPOSITORY                          TAG         IMAGE ID      CREATED
VIRTUAL SIZE
registry.access.redhat.com/rhel  0-21       e1f5733f050b  4 months ago
140.2 MB
registry.access.redhat.com/rhel  0          bef54b8f8a2f  4 months ago
139.6 MB
registry.access.redhat.com/rhel  0-23       bef54b8f8a2f  4 months ago
139.6 MB
registry.access.redhat.com/rhel  latest     bef54b8f8a2f  4 months ago
139.6 MB
```

docker load: If you have a container image stored as a tarball on your local system, you can load that image tarball so you can run it with the docker command on your local system. Here is how:

1. With the Docker image tarball in your current directory, you can load that tarball to the local system as follows:

```
# docker load -i rhel-server-docker-7.0-23.x86_64.tar.gz
```

2. To push that same image to the registry running on your localhost, tag the image with your hostname (or "localhost") plus the port number of the docker-registry service (TCP port 5000). **docker push** uses that tag information to push the image to the proper registry:

```
# docker tag bef54b8f8a2f localhost:5000/myrhel7
docker push localhost:5000/myrhel7
The push refers to a repository [localhost:5000/myrhel7] (len: 1)
Sending image list
Pushing repository localhost:5000/myrhel7 (1 tags)
bef54b8f8a2f: Image successfully pushed
Pushing tag for rev [bef54b8f8a2f] on
{http://localhost:5000/v1/repositories/myrhel7/tags/latest}
...
```

7.5.3. Investigating Docker images

If images have been pulled or loaded into your local registry, you can use the docker command **docker images** to view those images. Here's how to list the images on your local system:

```
# docker images
REPOSITORY   TAG      IMAGE ID      CREATED       VIRTUAL SIZE
redhat/rhel  latest   e1f5733f050b  4 weeks ago   140.2 MB
rhel         latest   e1f5733f050b  4 weeks ago   140.2 MB
redhat/rhel7 0        e1f5733f050b  4 weeks ago   140.2 MB
redhat/rhel7 0-21     e1f5733f050b  4 weeks ago   140.2 MB
redhat/rhel7 latest   e1f5733f050b  4 weeks ago   140.2 MB
rhel7        0        e1f5733f050b  4 weeks ago   140.2 MB
rhel7        0-21     e1f5733f050b  4 weeks ago   140.2 MB
rhel7        latest   e1f5733f050b  4 weeks ago   140.2 MB
```

NOTE: The default option to push an image or repository to the upstream Docker.io registry (**docker push**) is disabled in Red Hat version of the docker command. To push an image to a specific registry, identify the registry, its port number, and a tag that you designate in order to identify the image.

7.5.4. Investigating the Docker environment

Now that you have the docker and docker-registry services running, with a few containers available, you can start investigating the Docker environment and looking into what makes up a container. Run **docker** with the **version** and **info** options to get a feel for your Docker environment.

docker version: The version option shows which versions of different Docker components are installed. Notice that a newer docker package is available (**yum update docker** should take care of that in RHEL 7):

```
# docker version      Shows components/versions in use. Note that
docker needs updating here.
Client version: 1.2.0
Client API version: 1.14
Go version (client): go1.3.1
Git commit (client): 2a2f26c/1.2.0
OS/Arch (client): linux/amd64
```

```
Server version: 1.2.0
Server API version: 1.14
Go version (server): go1.3.1
Git commit (server): 2a2f26c/1.2.0
Last stable version: 1.3.0, please update docker
```

docker info: The info option lets you see the locations of different components, such as how many local containers and images there are, as well as information on the size and location of Docker storage areas.

```
# docker info
Containers: 3
Images: 5
Storage Driver: devicemapper
 Pool Name: docker-253:1-16826017-pool
 Pool Blocksize: 64 Kb
 Data file: /var/lib/docker/devicemapper/devicemapper/data
 Metadata file: /var/lib/docker/devicemapper/devicemapper/metadata
 Data Space Used: 1042.4 Mb
 Data Space Total: 102400.0 Mb
 Metadata Space Used: 1.3 Mb
 Metadata Space Total: 2048.0 Mb
Execution Driver: native-0.2
Kernel Version: 3.10.0-123.8.1.el7.x86_64
Operating System: Red Hat Atomic Host 7.0
```

7.5.5. Working with Docker containers

Docker images that are now on your system (whether they have been run or not) can be managed in several ways. The docker run command lets you say which command to run in a container. Once a container is running, you can stop, start, and restart it. You can remove containers you no longer need (in fact you probably want to).

Running Docker containers

When you execute a **docker run** command, you essentially spin up and create a new container from a Docker image. That container consists of the contents of the image, plus features based on any additional options you pass on the **docker run** command line.

The command you pass on the **docker run** command line sees the inside the container as its running environment so, by default, very little can be seen of the host system. For example, by default, the running applications sees:

- The file system provided by the Docker image.
- A new process table from inside the container (no processes from the host can be seen).
- New network interfaces (by default, a separate docker network interface provides a private IP address to each container via DHCP).

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

EXAMPLE #1 (Run a quick command): This docker command runs the **ip addr show eth0** command to see address information for the eth0 network interface within a container that is

generated from the RHEL image. Because this is a bare-bones container, we mount the **/usr/sbin** directory from the RHEL 7 host system for this demonstration (mounting is done by the -v option), because it contains the **ip** command we want to run. After the container runs the command, which shows the IP address (**172.17.0.2/16**) and other information about eth0, the container stops and is deleted (**--rm**).

```
# docker run -v /usr/sbin:/usr/sbin \
    --rm rhel /usr/sbin/ip addr show eth0
20: eth0:  mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 4e:90:00:27:a2:5d brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.10/16 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::4c90:ff:fe27:a25d/64 scope link tentative
       valid_lft forever preferred_lft forever
```

If you feel that this is a container you wanted to keep around and use again, consider assigning a name to it, so you can start it again later by name. For example, I named this container myipaddr:

```
# docker run -v /usr/sbin:/usr/sbin \
    --name=myipaddr rhel /usr/sbin/ip addr show eth0
20: eth0:  mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 4e:90:00:27:a2:5d brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.10/16 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::4c90:ff:fe27:a25d/64 scope link tentative
       valid_lft forever preferred_lft forever
# docker start -i myipaddr
22: eth0:  mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 4e:90:00:27:a2:5d brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.10/16 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::4c90:ff:fe27:a25d/64 scope link tentative
       valid_lft forever preferred_lft forever
```

EXAMPLE #2 (Run a shell inside the container): Using a container to launch a bash shell lets you look inside the container and change the contents. Here, I set the name of the container to **mybash**. The **-i** creates an interactive session and **-t** opens a terminal session. Without **-i**, the shell would open and then exit. Without **-t**, the shell would stay open, but you wouldn't be able to type anything to the shell.

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

```
# docker run --name=mybash -it rhel /bin/bash
[root@49830c4f9cc4/]#
```

Although there are very few applications available inside the base RHEL image, you can add more software using the **yum** command. With the shell open inside the container, run the following commands:

```
[root@49830c4f9cc4/]# cat /etc/redhat-release
Red Hat Enterprise Linux Server release 7.1 (Maipo)
[root@49830c4f9cc4/]# ps
bash: ps: command not found
[root@49830c4f9cc4/]# yum install -y procps
[root@49830c4f9cc4/]# ps -ef
```

```
UID      PID    PPID  C STIME TTY       TIME  CMD
root       1       0  0 15:36 ?     00:00:00  /bin/bash
root      46       1  0 15:43 ?     00:00:00  ps -ef
[root@49830c4f9cc4/]# exit
```

Notice that the container is a RHEL 7.1 container. The **ps** command is not included in the RHEL base image. However, you can install it with **yum** as shown above. To leave the container, type **exit**.

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

```
# docker ps -a
CONTAINER ID  IMAGE        COMMAND      CREATED        STATUS
PORTS  NAMES
49830c4f9cc4  rhel:latest  "/bin/bash"  2 minutes ago  Exited (0) 25
minutes ago        mybash
...
```

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

```
# docker start -ai mybash
[root@a0aee493a605/]#
```

EXAMPLE #3 (Bind mounting log files): One way to make log messages from inside a container available to the host system is to bind mount the host's /dev/log device inside the container. This example illustrates how to run an application in a RHEL container that is named **log_test** that generates log messages (just the logger command in this case) and directs those messages to the /dev/log device that is mounted in the container from the host. The **--rm** option removes the container after it runs.

```
# docker run --name="log_test" -v /dev/log:/dev/log --rm rhel logger
"Testing logging to the host"
# journalctl -b | grep Testing
Apr 22 16:00:37 node1.example.com logger[102729]: Testing logging to
the host
```

Investigating from outside of a Docker container

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

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

```
# docker ps
CONTAINER ID IMAGE       COMMAND     CREATED         STATUS
PORTS    NAMES
0d8b2ded3af0 rhel:latest "/bin/bash" 10 minutes ago  Up 3 minutes
mybash
```

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

```
# docker ps -a
CONTAINER ID  IMAGE       COMMAND    CREATED      STATUS
```

```
PORTS  NAMES
92b7ed0c039b  rhel:latest /bin/bash  2 days ago   Exited (0) 2 days ago
agitated_hopper
eaa96236afa6  rhel:latest /bin/bash  2 days ago   Exited (0) 2 days ago
prickly_newton
```

See the section "Working with Docker containers" for information on starting, stopping, and removing containers that exist on your system.

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

```
# docker inspect mybash
[{
    "Args": [],
        ...
        "Hostname": "a0aee493a605",
            "Image": "rhel",
            "Labels": {
                "Architecture": "x86_64",
                "Build_Host": "rcm-img04.build.eng.bos.redhat.com",
                "Name": "rhel-server-docker",
                "Release": "4",
                "Vendor": "Red Hat, Inc.",
                "Version": "7.1"
    ...
```

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

```
# docker inspect --format='{{.NetworkSettings.IPAddress}}' mybash
172.17.0.2
```

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

```
# docker inspect --format='{{.State.Pid}}' mybash
5007
# docker inspect --format='{{.HostConfig.PortBindings}}' mybash
map[8000/tcp:[map[HostIp: HostPort:8000]]]
```

Investigating within a running Docker container

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

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

Here is an example using **docker exec** to look into a running container named myrhel_httpd, then look around inside that container.

1. **Launch a container**: Launch a container such as the myrhel_httpd container described in Building an image from a Dockerfile or some other Docker container that you want to investigate. Type **docker ps** to make sure it is running:

```
# docker ps
CONTAINER ID   IMAGE              COMMAND               CREATED
STATUS        PORTS                NAMES
1cd6aabf33d9   rhel_httpd:latest  "/usr/sbin/httpd -DF  6 minutes
ago   Up 6 minutes  0.0.0.0:80->80/tcp   myrhel_httpd
```

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

```
# docker exec -it myrhel_httpd /bin/bash
[root@1cd6aabf33d9 /]# cat /etc/redhat-release
Red Hat Enterprise Linux Server release 7.1 (Maipo)
[root@1cd6aabf33d9 /]# ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 08:41 ?        00:00:00 /usr/sbin/httpd
-DFOREGROUND
apache       7     1  0 08:41 ?        00:00:00 /usr/sbin/httpd
-DFOREGROUND
...
root        12     0  0 08:54 ?        00:00:00 /bin/bash
root        35    12  0 08:57 ?        00:00:00 ps -ef
[root@1cd6aabf33d9 /]# df -h
Filesystem                          Size  Used Avail Use%
Mounted on
/dev/mapper/docker-253:0-540464...  9.8G  414M  8.8G   5%   /
tmpfs                               7.9G     0  7.9G   0%
/dev
shm                                 64M      0   64M   0%
/dev/shm
/dev/mapper/rhel_unused-root        137G   45G   92G  33%
/etc/hosts
tmpfs                               7.9G  8.0K  7.9G   1%
/run
tmpfs                               7.9G  184K  7.9G   1%
/run/secrets
tmpfs                               7.9G     0  7.9G   0%
/proc/kcore
[root@1cd6aabf33d9 /]# uname -r
3.10.0-229.1.2.el7.x86_64
[root@1cd6aabf33d9 /]# rpm -qa | less
redhat-release-server-7.1-1.el7.x86_64
basesystem-10.0-7.el7.noarch
nss-softokn-freebl-3.16.2.3-9.el7.x86_64
```

```
...
bash-4.2# free -m
              total        used        free      shared
buff/cache   available
Mem:          16046        6759         641          20
8645        8948
Swap:          8071           2        8069
[root@1cd6aabf33d9 /]# ip addr show eth0
44: eth0:  mtu 1500 qdisc pfifo_fast state UP
    link/ether 92:b1:31:b2:79:69 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.14/16 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::90b1:31ff:feb2:7969/64 scope link
       valid_lft forever preferred_lft forever
[root@1cd6aabf33d9 /]# exit
```

The commands just run from the bash shell (running inside the container) show you several things. The container holds a RHEL Server release 7.1 system. The process table (ps -ef) shows that the httpd command is process ID 1 (followed by five other httpd processes), /bin/bash is PID 12 and ps -ef is PID 35. Processes running in the host's process table cannot be seen from within the container. The container's file system consumes 414M of the 9.8G available root file system space.

There is no separate kernel running in the container (uname -r shows the host system's kernel: 3.10.0-229.1.2.el7.x86_64). The rpm -qa command lets you see the RPM packages that are included inside the container. In other words, there is an RPM database inside of the container. Viewing memory (free -m) shows the available memory on the host (although what the container can actually use can be limited using cgroups). The IP address in the container (172.17.0.14/16) is assigned to the container from the host system via DHCP. In this case, the host system has an interface named docker0 with an IP address of 172.17.42.1/16.

Starting and stopping containers

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

Starting containers: A docker container that doesn't need to run interactively can start with only the start option and the container ID or name:

```
# docker start myrhel_httpd
myrhel_httpd
```

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

```
# docker start -a -i agitated_hopper
bash-4.2# exit
```

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

```
# docker stop myrhel_httpd
myrhel_httpd
```

The **stop** option sends a SIGTERM signal to terminate a running container. If the container doesn't

stop after a grace period (10 seconds by default), docker sends a SIGKILL signal. You could also use the **docker kill** command to kill a container (SIGKILL) or send a different signal to a container. Here's an example of sending a SIGHUP signal to a container (if supported by the application, a SIGHUP causes the application to re-read its configuration files):

```
# docker kill --signal="SIGHUP" myrhel_httpd
```

Removing containers

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

```
# docker rm goofy_wozniak
```

You can remove multiple containers on the same command line:

```
# docker rm clever_yonath furious_shockley drunk_newton
```

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

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

7.5.6. Creating Docker images

So far we have grabbed some existing docker container images and worked with them in various ways. To make the process of running the exact container you want less manual, you can create a Docker image from scratch or from a container you ran that combines an existing image with some other content or settings.

Creating an image from a container

The following procedure describes how to create a new image from an existing image (rhel:latest) and a set of packages you choose (in this case an Apache Web server, httpd).

NOTE: For the current Red Hat Docker release, the default RHEL 7 Docker image you pull from Red Hat will be able to draw on RHEL 7 entitlements available from the host system. So, as long as your Docker host is properly subscribed and the repositories are enabled that you need to get the software you want in your container (and have Internet access from your Docker host), you should be able to install packages from RHEL 7 software repositories.

1. **Install httpd on a new container**. Assuming you have loaded the **rhel** image from the Red Hat Customer Portal into your local system, and properly subscribed your host using Red Hat subscription management, the following command will:
   - Use that image as a base image
   - Get the latest versions of the currently installed packages (update)
   - Install the httpd package (along with any dependent packages)
   - Clean out all yum temporary cache files

```
# docker run -i rhel:latest /bin/bash -c "yum clean all;  \
    yum update -y; yum install -y httpd; yum clean all"
```

2. **Commit the new image**: Get the new container's ID or name (**docker ps -l**), then commit that container to your local repository. When you commit the container to a new image, you can add a comment (-m) and the author name (-a), along with a new name for the image (rhel_httpd). Then type **docker images** to see the new image in your list of images.

```
# docker ps -l
CONTAINER ID  IMAGE          COMMAND              CREATED
STATUS                     PORTS  NAMES
f6832df8da0a  redhat/rhel7:0 /bin/bash -c 'yum cl About a minute
ago  Exited (0) 13 seconds ago        backstabbing_ptolemy4
# docker commit -m "RHEL with httpd" -a "Chris Negus"
f6832df8da0a rhel_httpd
630bd3ff318b8a5a63f1830e9902fec9a4ab9eade7238835fa6b7338edc988ac
# docker images
REPOSITORY   TAG     IMAGE ID      CREATED         VIRTUAL SIZE
rhel_httpd   latest  630bd3ff318b  27 seconds ago  170.8 MB
redhat/rhel  latest  e1f5733f050b  4 weeks ago     140.2 MB
```

3. **Run a container from new image**: Using the image you just created, run the following **docker run** command to start the Web server (httpd) you just installed. For example:

```
# docker run -d -p 8080:80 rhel_httpd:latest \
        /usr/sbin/httpd -DFOREGROUND
```

   In the example just shown, the Apache Web server (httpd) is listening on port 80 on the container, which is mapped to port 8080 on the host.

4. **Check that container is working**: To make sure the httpd server you just launched is available, you can try to get a file from that server. Either open a Web browser from the host to address http://localhost:8080 or use a command-line utility, such as curl, to access the httpd server:

```
# curl http://localhost:8080
```

Building an image from a Dockerfile

Once you understand how images and containers can be created from the command line, you can try building containers in a more permanent way. Building container images from Dockerfile files is by far the preferred way to create Docker formatted containers, as compared to modifying running containers and committing them to images.

The procedure here involves creating a Dockerfile file that includes many of the features illustrated earlier:

- Choosing a base image
- Installing the packages needed for an Apache Web server (httpd)
- Mapping the server's port (TCP port 80) to a different port on the host (TCP port 8080)
- Launching the Web server

While many features for setting up a Docker development environment for RHEL 7 are in the works, here are a few issues you should be aware of as you build your own docker containers:

- **Entitlements**: Here are a few issues associated with Red Hat entitlements as they relate to containers:
  - If you subscribe your Docker host system using Red Hat subscription manager, when you build a Docker image on that host, the build environment automatically has access to the same Red Hat software repositories you enabled on the host.
  - To make more repositories available when you build a container, you can enable those repositories on the host or within the container.
  - Because the subscription-manager command is not supported within a container, enabling a repo inside the /etc/yum.repos.d/redhat.repo file is one way to enable or disable repositories. Installing the yum-utils package in the container and running the yum-config-manager command is another.
  - If you build a RHEL 6 container on a RHEL 7 host, it will automatically pick up RHEL 6 versions of the repositories enabled on your host.
  - For more information on Red Hat entitlements within containers, refer to the Docker Entitlements solution.
- **Updates**: Docker containers in Red Hat Enterprise Linux do not automatically include updated software packages. It is your responsibility to rebuild your Docker images on occasion to keep packages up to date or rebuild them immediately when critical updates are needed. The "RUN yum update -y" line shown in the Dockerfile example below is one way to update your packages each time the Docker image is rebuilt.
- **Images**: By default, docker build will use the most recent version of the base image you identify from your local cache. You may want to pull (docker pull command) the most recent version of an image from the remote Docker registry before you build your new image. If you want a specific instance of an image, make sure you identify the tag. For example, just asking for the image "centos" will pull the centos:latest image. If you wanted the image for CentOS 6, you should specifically pull the centos:centos6 image.

  - **Create project directories**: On the host system where you have the docker and docker-registry services running, create a directory for the project:

    ```
    # mkdir -p httpd-project
    # cd httpd-project
    ```

  - **Create the Dockerfile file**: Open a file named Dockerfile using any text editor (such as **vim Dockerfile**). Assuming you have registered and subscribed your host RHEL 7 system, here's an example of what the Dockerfile file might look like to build a Docker container for an httpd server:

    ```
    # My cool Docker image
    # Version 1

    # If you loaded redhat-rhel-server-7.0-x86_64 to your local
    registry, uncomment this FROM line instead:
    # FROM registry.access.redhat.com/rhel
    # Pull the rhel image from the local registry
    FROM registry.access.redhat.com/rhel

    MAINTAINER Chris Negus

    # Update image
    ```

```
RUN yum update -y
# Add httpd package. procps and iproute are only added to
investigate the image later.
RUN yum install httpd procps iproute -y
RUN echo container.example.com > /etc/hostname

# Create an index.html file
RUN bash -c 'echo "Your Web server test is successful." >>
/var/www/html/index.html'
```

- **Checking the Dockerfile syntax (optional)**: Red Hat offers a tool for checking a Dockerfile file on the Red Hat Customer Portal. If you like, you can go to the Linter for Dockerfile page and check your Dockerfile file before you build it.

- **Build the image**: To build the image from the Dockerfile file, you need to use the build option and identify the location of the Dockerfile file (in this case just a "." for the current directory):

  NOTE: Consider using the --no-cache option with docker build. Using --no-cache prevents the caching of each build layer, which can cause you to consume excessive disk space.

```
# docker build -t rhel_httpd .
Uploading context  2.56 kB
Uploading context
Step 0 : FROM registry.access.redhat.com/rhel
 ---> f5f7ddddef7d
Step 1 : MAINTAINER Chris Negus
 ---> Running in 3c605e879c72
 ---> 77828ebe8f6f
Removing intermediate container 3c605e879c72
Step 2 : RUN yum update -y
 ---> Running in 9f45bb262dc6
...
 ---> Running in f44ea9eb6155
 ---> 6a532e340ccf
Removing intermediate container f44ea9eb6155
Successfully built 6a532e340ccf
```

- **Run the httpd server in the image**: Use the following command to run the httpd server from the image you just build (named rhel_httpd in this example):

```
# docker run -d -t --name=myrhel_httpd \
        -p 80:80 -i rhel_httpd:latest \
        /usr/sbin/httpd -DFOREGROUND
```

- **Check that the server is running**: From another terminal on the host, type the following to check that you can get access the httpd server:

```
# netstat -tupln | grep 80
tcp6       0      0 :::80       :::*      LISTEN
26137/docker-proxy
# curl localhost:80
Your Web server test is successful.
```

Tagging Images

You can add names to images to make it more intuitive what they contain. Using the **docker tag** command, you essentially add an alias to the image, that can consist of several parts. Those parts can include:

registryhost/username/NAME:tag

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

```
# docker tag 474ff279782b myrhel7
```

In the previous example, the **rhel7** image had a image ID of 474ff279782b. Using **docker tag**, the name **myrhel7** now also is attached to the image ID. So you could run this container by name (rhel7 or myrhel7) or by image ID. Notice that without adding a :tag to the name, it was assigned :latest as the tag. You could have set the tag to 7.1 as follows:

```
# docker tag 474ff279782b myrhel7:7.1
```

To the beginning of the name, you can optionally add a user name and/or a registry name. The user name is actually the repository on Docker.io that relates to the user account that owns the repository. Tagging an image with a registry name was shown in Getting images from public Docker registries. Here's an example of adding a user name:

```
# docker tag 474ff279782b cnegus/myrhel7
# docker images | grep 474ff279782b
rhel7           latest  474ff279782b  7 months ago  139.6 MB
myrhel7         latest  474ff279782b  7 months ago  139.6 MB
myrhel7         7.1     474ff279782b  7 months ago  139.6 MB
cnegus/myrhel7  latest  474ff279782b  7 months ago  139.6 MB
```

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

Saving and Importing Images

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

```
# docker save -o myrhel7.tar myrhel7:latest
```

The **myrhel7.tar** file should now be stored in your current directory. Later, when you ready to reuse the tarball as a container image, you can import it to another docker environment as follows:

```
# cat myrhel7.tar | docker import - cnegus/myrhel7
```

Removing Images

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

```
# docker rmi rhel
```

You can remove multiple images on the same command line:

```
# docker rmi rhel fedora
```

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

```
# docker rmi $(docker images -a -q)
```

7.6. SUMMARY

At this point, you should be able to get Red Hat Docker installed with the docker and docker-registry services working. You should also have one or more Docker images to work with, as well as know how to run containers and build your own images.

CHAPTER 8. MANAGING STORAGE WITH DOCKER FORMATTED CONTAINERS

8.1. 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 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. However, using the loopback option is not recommended for production systems. During the planning phase, make sure of the following things:

1) You are running direct-lvm 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.

8.2. USING DOCKER-STORAGE-SETUP

The **docker-storage-setup** utility is installed with the *docker* package and 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 */usr/lib/docker-storage-setup/docker-storage-setup* configuration file. You can override these options by creating a file */etc/sysconfig/docker-storage-setup* using new values.

docker-storage-setup needs to know where the free space is to set up a thin pool. Following are some of the ways you can configure the system to make sure docker-storage-setup can setup an LVM thin pool.

See **man docker-storage-setup(1)** for more information. (Note that manual pages are not available by default on RHEL Atomic, you need to have the RHEL Atomic Tools container downloaded.)

8.2.1. LVM thin pool in the volume group containing the root volume

By default, docker-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.

8.2.2. LVM thin pool in a user specified volume group

docker-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
```

8.2.3. 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 docker-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
```

8.3. 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 docker-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.

8.3.1. 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.

8.3.1.1. 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".

8.3.1.2. 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
```

8.4. 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 **docker-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.

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.

It is very 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.

- **/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

8.4.1. 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
```

8.4.2. 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.

8.4.2.1. 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
```

8.4.2.2. 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
```

8.4.2.3. 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.

8.5. CHANGING DOCKER STORAGE CONFIGURATION

If you change the storage configuration of Docker, 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
```

8.6. 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**
>
> Note that OverlayFS is not POSIX-compliant (some of the file system semantics are different from standard file systems like ext4 and XFS) and does not yet support SELinux. Therefore, make sure your applications work with OverlayFS before enabling it with Docker. For more information on the use of OverlayFS with Docker, see Chapter 19. File Systems from the Red Hat Enterprise Linux 7.2 Release Notes.

The general way to enable the **`overlay`** Graph Driver for Docker is to disable SELinux and specify **`overlay`** in */etc/sysconfig/docker-storage-setup*.

Important

Changing the storage backend is a destructive operation. Before starting, be sure to back up your images with **`docker save`**. Afterwards, you can restore them from backup with **`docker load`**.

Stop docker and remove the current storage:

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

Disable SELinux, by removing the option **`--selinux-enabled`** from the **`OPTIONS`** variable in */etc/sysconfig/docker*:

```
# sed -i '/OPTIONS=/s/--selinux-enabled//' /etc/sysconfig/docker
```

Set **`STORAGE_DRIVER`** to **`overlay`** in */etc/sysconfig/docker-storage-setup*:

```
STORAGE_DRIVER=overlay
```

Restart docker-storage-setup, and then docker:

```
# systemctl start docker-storage-setup
# systemctl start docker
```

- **Kickstart**

  For a Kickstart installation, use the following commands in the **`%post`** section:

  ```
  %post
  sed -i '/OPTIONS=/s/--selinux-enabled//' /etc/sysconfig/docker
  echo "STORAGE_DRIVER=overlay" >> /etc/sysconfig/docker-storage-setup
  %end
  ```

- **cloud-init**

  For a cloud-init installation, include the following snippet in the *user-data* file:

  ```
  runcmd:
    - sed -i '/OPTIONS=/s/--selinux-enabled//' /etc/sysconfig/docker
    - echo "STORAGE_DRIVER=overlay" >> /etc/sysconfig/docker-storage-
  setup
  ```

8.7. 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 9. STARTING A CONTAINER USING SYSTEMD

To start a container automatically at boot time, first configure it 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:

```
[Unit]
Description=Redis container
Author=Me
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
```

After creating the unit file, use the **systemctl enable** command to start the container automatically.

To learn more about configuring services with systemd, refer to chapter called Managing Services with systemd in Red Hat Enterprise Linux 7 System Administrators Guide.

CHAPTER 10. RUNNING SUPER-PRIVILEGED CONTAINERS

10.1. 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, filesystems, 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.

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

10.2. 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). Y ou 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 filesystem 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.

10.3. 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 Tools Container, it knows to use **/host** as the root of the filesystem and not **/**. When you run other commands from within the RHEL 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:

10.3.1. Privileges

A privileged container runs applications as root user on the host by default. The container has this ability because it runs with a unconfined_t SELinux security context.

10.3.2. Mount Tables

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

10.3.3. 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

10.3.4. Inter-process communications

he 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 semaphone sets on the host.

CHAPTER 11. USING THE ATOMIC TOOLS CONTAINER IMAGE

11.1. OVERVIEW

The Red Hat Enterprise Linux Atomic Tools Container (RHEL Tools Container) is a docker-formatted image that includes hundreds 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 tools such as sosreport, kdump, and many others (most of which are not included with RHEL Atomic).

Using this topic, you will learn:

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

11.2. OVERVIEW OF RHEL TOOLS CONTAINER

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 while it is being deployed and run efficiently once it is deployed. This makes RHEL Atomic particularly suited for hosting containers in cloud environments.

One of the results of keeping down the size of Atomic is that many of the tools that you might expect in a standard RHEL system are not installed in Atomic. Compounding that issue is that fact that Atomic is not made to allow additional software packages to be installed (**yum install favorite_RPM** is not supported).

The solution to this problem is to bring the RHEL Tools Container into a RHEL Atomic system. You can do this either when the Atomic system is initially deployed or when you have a problem and need additional tools to troubleshoot it.

Here are a few facts you should know about the RHEL Tools Container:

- **It’s big**: As containers go, it is fairly large (currently about 1GB). This is because we want the container to include as many tools as possible for monitoring and troubleshooting Atomic. If you like, you can just put the container on an Atomic system as it is needed, if the space consumption is a problem during normal operations. (Just keep in mind that pulling in the container could take a bit of time at a point where you may be in a hurry to fix a problem).
- **Contains man pages**: This container offers a way to get RHEL documentation on to the container. Because the man command is not included in RHEL Atomic, the RHEL Tools Container offers a way of viewing man pages. All content in /usr/share/doc is also included for all the installed packages in the container.
- **Opens privileges**: Containers, by default, cannot see most of the Atomic host’s file system or namespaces (networking, IPC, process table, and so on). 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 though they were run directly on the host.
- **May behave differently**: You can expect, however, that some commands run from within the

container, even with privileges open, will behave differently than would the same commands run directly on a RHEL host system. Later, this topic describes some of the most useful tools that come in the RHEL Tools Container and tells how those commands might work differently than you expect when running them in a privileged container.

11.3. GETTING AND RUNNING THE RHEL TOOLS CONTAINER

The RHEL Tools Container is designed to run on RHEL Atomic hosts. So before you can use it, you need to install a RHEL Atomic system. Then you can get, load, and run the RHEL Tools Container, as described in the following procedure:

- **Install RHEL Atomic Host**: To install and configure a RHEL Atomic host, refer to the appropriate installation guide listed in the documentation.
- **Get RHEL Tools Image**: While logged into the RHEL Atomic host, get the RHEL Tools Container by pulling it with the **docker pull** command, as follows:

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

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

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

You now have a shell open inside the container, with all the tools in that container ready to run. When you are done, run exit. The next section contains examples of some commands you might want to run from your RHEL Tools container.

11.4. RUNNING COMMANDS FROM THE RHEL TOOLS CONTAINER

Refer to the following sections to get a feel for commands available within the RHEL Tools Container and understand how commands run inside or outside of the container may behave differently.

- **blktrace**: To use **blktrace** from within a container, you need to first mount the debugfs file system. Here's an example of mounting that file system and running 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
```

- **sosreport**: The **sosreport** command includes an atomic plugin that makes it container-aware. So you can simple run **sosreport** and have a report generated that would produce almost the exact same results you would get if you ran it directly on the host. From within the container, you could run:

```
# 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
```

Keep in mind that the report is copied to the **/var/tmp** directory on the host. Because the host's root file system is mounted on **/** within the container, the report is available in the **/host/var/tmp** directory within the container. So the report will be available after you close the container.

- **useradd**: If you want to add a user to do some non-root activities within the container, you can use the **useradd** command: steps to create the home directory:

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

- **strace**: Because you can see the host's process table from within the RHEL Tools Container, many commands that can take a process ID as an argument will work from within the container. Here's an example of 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 ...
```

11.5. TIPS FOR RUNNING RHEL TOOLS CONTAINER

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

- Unless you explicitly remove the container (**docker rm rhel-tools**), the container continues to exist on your system.
- Because that container continues to exist, any changes you make (for example, **yum install package**) will persist each time you run the container again. So **atomic run rhel7/rhel-tools** won't pull down any files or do any additional setup on the host the second time you run it.
- Notice that the image is identified by the name **rhel7/rhel-tools**, however, when the image is run, the running instance is referred to as a container named **rhel-tools**. The container is not removed by default, so to see the name of the container, even after it is stopped, type **docker ps -a**.
- Because the **rhel-tools** container is retained, even after it is removed, you cannot upgrade to a new version of the container without explicitly removing the old one. To do that, you should preserve any files from the container you want to keep (copy them somewhere on **/host**) then type **docker rm rhel-tools**. After that, proceed to do a new **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.
- The **subscription-manager** command is available on both the RHEL Atomic host and within the RHEL Tools Container. In Atomic you must assign a valid Red Hat subscription to the host. In the container, **subscription-manager** package is there to make the related man pages

available. But the **subscription-manager** command should not be run from within the container. Once the host is subscribed, however, you can use the **yum** commands within the container to add and manage software packages within the container.

- 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.

CHAPTER 12. USING THE ATOMIC RSYSLOG CONTAINER IMAGE

12.1. 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.

12.2. 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.

12.3. 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 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 filesystem 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 8docker 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
```

CHAPTER 13. USING ATOMIC SYSTEM ACTIVITY DATA COLLECTOR (SADC) CONTAINER IMAGE

13.1. OVERVIEW

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.

13.2. OVERVIEW OF THE SADC CONTAINER

Because sysstat package (which includes sar, iostat, sadc an 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.

13.3. 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.

13.4. TIPS FOR RUNNING 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
  ```