Red Hat Enterprise Linux Atomic Host 7

Recommended Practices for Container Development

Recommended Practices Guide for Container Development

Last Updated: 2017-11-30

Red Hat Enterprise Linux Atomic Host 7 Recommended Practices for Container Development

Recommended Practices Guide for Container Development

Legal Notice

Abstract

Fundamental container-related practices.

Table of Contents

CHAPTER 1. OVERVIEW

This guide provides recommendations for container development that are supported by Red Hat. Though containers, in their current Docker-driven implementation, are a new and rapidly-developing technology, this guide captures the state of container support within Red Hat. Because there are many use cases for containers, this guide provides general recommendations about fundamental container-related practices that are useful and supported by Red Hat.

1.1. CONTAINER PROVENANCE

Where do containers come from? How do we get them? How do we make sure that we get them in a secure manner?

Containers are built from images, and images are stored in repositories on registries. This topic discusses two registries that the command **docker pull** can access:

- The **`docker.io`** registry
- [The Red Hat Registry](http://registry.access.redhat.com/)

1.1.1. The Risks of "docker pull"

The **docker pull** command, used without the registry from which you are pulling, is a potentially dangerous command. Docker makes no distinction between retrieval of software and installation of software. This behavior is different from the case of an RPM: it is safe to use **wget** to retrieve an RPM that contains malware as long as you do not install the RPM. If is not safe, however, to pull malware using **docker pull** because retrieving an image is functionally equivalent to installing it.

For example, assume that containers are software that upon retrieval runs as privileged. Containers relinquish privilege only after having their settings manipulated. The isolation of containers is usually voluntary, and they are not isolated by default.

WARNING

Exercise caution when using the **docker pull** command.

If a container is not found in the Red Hat Registry, **docker pull** fails over to the **`docker.io`** registry. Red Hat does not verify the security or authenticity of containers from third-party sources, such as **`docker.io`**. See also the Image-Naming Conventions chapter.

When retrieving images from places other than the Red Hat Registry, avoid **`docker pull`**. If possible, use **`docker load`** and **`docker save`**. You can use **`docker load`** and **`docker save`** with tarballs and then verify the images.

Why would you want to use **`docker load`** and **`docker save`** instead of **`docker pull`**? **`docker load`** and **`docker save`** provide a way to avoid the security vulnerability introduced by exposing your system to third-party registries, which could happen when you run **`docker pull`**.

For more information on container provenance and exercising caution when using **`docker pull`**, see Red Hat's Security Blog post Before You Initiate a "docker pull".

- **docker pull**

The basic form of docker pull is: **`$ sudo docker pull repo/image:tag`** where **`repo`** and **`tag`** are optional. If **`repo`** and **`tag`** are not specified, docker will attempt to locate the image in the **`docker.io`** registry. For this reason, it is advisable always to explicitly name the registry from which you want to pull the image. If no registry is specified, docker attempts to find an image on the **`docker.io`** registry. If no tag is supplied, docker attempts by default to pull the latest image.

- **docker load**

The basic form of docker load is: **`$ sudo docker load -i input.tar`** where **`input.tar`** is a tar image to be loaded into your local container registry. The **`-i`** is optional and the **`input.tar`** file name is optional. If neither **`-i`** nor a file name is specified, **`docker load`** expects tar data on STDIN.

- **docker save**

The basic form of docker save is: **`$ sudo docker save -o output.tar`** where **`output.tar`** is a tar image to be loaded into your local container registry. The **`-o`** is optional and the **`output.tar`** file name is optional. If neither **`-o`** nor a file name is specified, **`docker save`** will output container data to STDOUT.

CHAPTER 2. BUILDING CONTAINERS

It is necessary to maintain a mechanism that permits you to update your containers, so that they can be patched when security vulnerabilities that apply to those containers are discovered. An understanding of how containers are built helps maintain them properly.

To minimize the risk of having security vulnerabilities in your containers, you need to make it possible to update your containers so that you can apply security patches to them. To make your containers reliably updateable you need to leverage as much of the existing packaging infrastructure and tooling as possible. This means that you will have to use Yum. In this respect, running containers is no different than running any other distribution. All the rules that you're familiar with apply: you should use RPM if you can.

CHAPTER 3. LABELS

Labels make it possible to embed commands in to container images instead of adding them as options to the command line. These are usually docker commands with multiple options that are required to run a particular container, and they often include running the container as privileged. The options are handled on a container-by-container basis and can make the command very long and hard to remember. Labels are another way to provide this information to docker.

Labels build on the metadata that is already available in Docker. This metadata is stored in a *.json* file that defines the container. Docker has a feature that lets you put name-value pairs into containers and labels build on the feature that permits name-value pairs to be inserted into containers.

3.1. LABELS EXAMPLE

Labels simplify running long docker commands on the command-line. The example below will help you understand labels.

Here, we create "run labels" that have a semantic meaning.

Suppose that you want to run the Chrome browser with GPU acceleration and pulse audio. To do this, your container needs access to the GPU. That would be easy enough, but we must consider "composition". X11 and Window Manager work together: that is composition. The most efficient way for these two programs to work together is to create one big frame buffer to which all compositing programs write. This is handled with a very long command, for example:

```
$ sudo docker run -v /dev/dri:/dev/dri \
-v /dev/snd:/dev/snd \
-v /dev/shm:/shm \
-ipc=container:foo_bar \
-privileged -e 'DISPLAY=:0' \
-u username rhel_chrome google-chrome
```

Let’s break this down:

`-v /dev/dri:/dev/dri`

video card component

`-v /dev/snd:/dev/snd`

sound card component

`-v /dev/shm:/dev/shm`

shared memory

`shm`

shared memory

`-ipc`

inter-process communication

`container:foo_bar` means that this label refers to another running container called **`foo_bar`**. This links the running container **`foo_bar`** to the running container **`fedora_chrome`**. This argument shares the interprocess communication namespaces of the two containers and ensures that the two containers refer to the same objects and that each of the two containers uses the same names to refer to those objects. This is important because they will be communicating over **`/dev/shm`**.

Shared memory is one way of providing inter-process communication.

`-e 'DISPLAY=:0'`

which X11 display the output should go to

`-u username`

this specifies the username

`rhel_chrome`

the name of the container

`google-chrome`

the command that the container will run

CHAPTER 4. IMAGE-NAMING CONVENTIONS

4.1. RED HAT'S NAMING POLICY

Red Hat has provided a consistent naming policy for Docker-formatted images in order to provide predictability for users.

Docker URLs in the V1 version of the protocol and registry format work similar to GitHub repository names. Their structure is:

```
REGISTRY[:PORT]/USER/REPO[:TAG]
```

The implicit default registry is **`docker.io`**. This means that relative URLs, such as "redhat/rhel" resolve to **`docker.io/redhat/rhel`**.

The special name "library/*" maps to direct, non-prefixed images. For instance: "docker.io/rhel".

4.2. REDIRECTION OF REQUESTS

Requests for Red Hat content are redirected from **`docker.io`** to **`registry.access.redhat.com`**. The following mappings describe this redirection:

- docker.io/rhel → registry.access.redhat.com/rhel (alias for rhel7)
- docker.io/rhel7 → registry.access.redhat.com/rhel7
- docker.io/rhel6 → registry.access.redhat.com/rhel6
- docker.io/redhat/* → registry.access.redhat.com/redhat/*

The version of Docker that ships with Red Hat Enterprise Linux 7.2 contacts the internal Red Hat registry by default. This allows Red Hat to segment the namespace by registry.

The registry content is a mapping of tagged names and does not involve any copying of content. This mapping is achieved by means of multiple symbolic links. Red Hat keeps these links up-to-date to ensure the integrity of your data.

The REPO is a repository containing a number of explicitly tagged and a number of hidden layers. An 'IMAGE' is a specific layer-complete branch within a repository. Often the terms 'image' and 'repository' are used synonymously, even thought they are not the same. For instance, different images can be tagged into a single repository.

CHAPTER 5. PROCESS MANAGEMENT

This section on process management covers two different situations:

1. the single-process container, which houses the files that the container needs to do its job
2. the managed-multiprocess container, which consists of several heterogeneous processes running at the same time

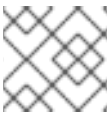
Apache is an example of a single-process container. Apache does all of its own logging, which means that it does not require an external container to do its logging. This means that using **systemd** in this case does not offer any benefits.

GNOME 3 is an example of a managed-multiprocess container. Attempting to manage this kind of container manually (for instance, trying to get upower and DBUS and logging working together) is more onerous than simply getting **systemd** to work.

5.1. RUNNING SYSTEMD IN A CONTAINER

In the case of the managed-multiprocess container, implement systemd to manage the several processes running within the container. To run systemd in a container, run a command similar to the following command:

```
$ sudo docker run -it IMAGE /bin/bash
```



NOTE

The above command (**-it**) launches an interactive-tty container running **systemd**.

For more historical context on running **systemd** in a container, see Dan Walsh's May 2014 blog post on the matter: Running systemd within a Docker Container.

5.1.1. journald and systemd in a container

Disable journal auditing in containers. Permit only the docker host to run **journald**. Journal auditing is a kernel feature, and only one instance of journald can use it at a time. (It is fine if the docker host is still running journald.)

5.1.2. Services and containers

Services work out of the box in containers. Services "just work" in containers. No special configuration is necessary for services to work in containers.

5.1.3. Managing Hardware that is Outside the Container

It is possible to create a container that uses **systemd** to manage outside hardware, but so configuring a container makes it non-portable because the container expects the specific hardware setup against which it is configured to be available for management.

5.1.4. systemd and Zombies

systemd solves the zombie problem. When processes die and become zombies, **`init 1`** reaps them. This is just as true inside a container as it is outside a container. Expect **systemd** inside a container to reap zombies in exactly the way you have come to expect it to reap zombies outside a container.

If you start a container without **systemd** (or, for instance, **sysv**), processes that die in that container will never be reaped.

CHAPTER 6. LINTER FOR DOCKERFILE

Linter for Dockerfile is a utility that checks Dockerfiles for errors helping you build valid Dockerfiles.

You need a Red Hat login and a valid subscription to access Linter for Dockerfile.

Access the Linter for Dockerfile utility here: https://access.redhat.com/labs/linterfordockerfile/

To validate your Dockerfile:

1. Upload a Dockerfile by either providing a URL to the Dockerfile or manually paste in the content of a Dockerfile
2. Select a specific profile (for instance, "Red Hat ISV").
3. Select **Check**.