

The next generation of container engines

# Podman

## IN ACTION

Daniel J Walsh





**MEAP Edition**  
**Manning Early Access Program**  
**Podman in Action**  
**The next generation of container engines**

**Version 3**

Copyright 2022 Manning Publications

For more information on this and other Manning titles go to  
[manning.com](https://manning.com)

# welcome

---

Thank you for purchasing the MEAP for *Podman in Action*. To get the most benefit from this book, you should have some understanding of how processes work on a Linux machine. Being able to work at the command line will be necessary. Having knowledge of Docker would be beneficial, but is not required.

When I first started working on container technology 20 years ago, we did not even call them containers, we called them sandboxes. I created a tool called the SELinux sandbox which used security tools like SELinux, the mount namespace and cgroups to control desktop applications access to the home directory, back in 2008. In 2013 when Docker was exploding on the scene, I was tasked with leading a team of engineers at Red Hat to work with the upstream Docker project. As soon as I started working on Docker, I recognized what a breakthrough the technology was, but I thought there were problems with its design. I did not like a centralized daemon running as root. There are better ways to run containers by taking advantage of more core concepts of the OS and this led to the creation of Podman and other container tools.

As we designed Podman we realized that the CLI and eventually the API had to match the Docker CLI and API, and then extend the technologies to take advantage of what we had learned from running containerized workloads over the years. With my background of 40 years of computer security, I wanted to take advantage of everything the OS provided to secure the containers, and you will learn a lot of this in the book.

Throughout this book you will learn about how Podman works with the CoreOS to take advantage of all of the features of the OS used to isolate containerized applications from each other. This isolation is from a security point of view, as well as from resource constraints, and convincing the applications that they are running on a dedicated system.

I believe this book is useful to developers building containerized applications as well as administrators learning how to run these containerized tools, but also to engineers just looking to learn about containers.

If you have a solid knowledge of Docker, you can skim over some of the chapters, and get to the sections that cover some of the key differentiators of Podman

Thank you again for your interest and for purchasing the MEAP!  
If you have any questions, comments, or suggestions, please share them in Manning's [livebook discussion forum](#).

-Dan Walsh

# *brief contents*

---

*1 Podman: next generation container engine*

## **PART 1 FOUNDATIONS**

*2 Command line*

*3 Volumes*

*4 Pods*

## **PART 2: DESIGN**

*5 Customization and configuration files*

*6 Rootless containers*

## **PART 3: ADVANCED TOPICS**

*7 Integration with SystemD*

*8 Working with Kubernetes*

*9 Podman as a service*

## **PART 4: CONTAINER SECURITY**

*10 Security container isolation*

*11 Security considerations*

## **APPENDIXES**

*A Podman-related container tools*

*B OCI runtimes*

*C Getting Podman*

*D Contributing to Podman*

*E Podman on macOS*

*F Podman on Windows*

# 1

## *Podman: next generation container engine*

### **This chapter covers**

- What is Podman
- Advantages of Podman over Docker
- Examples of using Podman

Starting this book is difficult, because so many people come into it with different expectations and experiences. I think it is important to level set everyone one and define terminology. In the container world terms like container orchestrator, container engine and most often container runtime get used interchangeably and I believe this leads to confusion. Figure 1.1 puts different open source container projects into their category.



**Figure 1.1** Different open source projects dealing with containers within their categories of orchestrators, Engines and runtimes.

Container orchestrators are software projects and products which orchestrate containers onto multiple different machines or nodes. These orchestrators communicate with container engines to run containers. The primary container orchestrator is Kubernetes which was originally designed to talk to the Docker daemon container engine, but using Docker is becoming obsolete as Kubernetes primarily uses CRI-O or containerd as its container engine. CRI-O and containerD are purpose built for running orchestrated Kubernetes containers. (CRI-O is covered in appendix A).

The OCI container runtimes configure different parts of the Linux kernel and then finally launch the containerized application. The two most commonly used container runtimes are runc and crun. See appendix B to understand the differences between the OCI container runtimes.

Container engines are primarily used for configuring containerized applications to run on a single local node. They can be launched directly by users, administrators and developers. They can also be launched out of SystemD unit files as boot as well as launched by container orchestrators like Kubernetes. As I mentioned above CRI-O and containerd are container engines used by Kubernetes to manage containers locally. They really are not intended to be

used directly by users. Docker and Podman are the primary container engines used by users to develop, manage and run containerized applications on a single machine. Podman is seldom used to launch containers for Kubernetes, and thus Kubernetes is not generally covered in this book.

This book shows how you can use Podman as a local container engine to launch containers on a single node, either locally or through a remote REST API.

Podman stands for Pod Manager. Pod is a concept popularized by the Kubernetes project. A pod is one or more containers sharing the same namespaces and cgroups (resource constraints). Pods are covered in Chapter 4. Podman runs individual containers as well as pods.



**Figure 1.2** Podman's logo, a group of Selkies, Ireland's concept of a mermaid. Selkies are half human and half seal, and a group of them is called a pod.

The podman project describes Podman as “a *daemonless container engine for developing, managing, and running OCI Containers on your Linux System. Containers can either be run as root or in rootless mode.*”

Podman often is described with the simple line “alias docker=podman” because Podman does almost everything that Docker can do with the same command line as Docker. But as you learn in this book, Podman can do so much more. Understanding Docker is not critical to understanding Podman, but is helpful.

**NOTE** The Open Container Initiative (OCI) is a standards body whose primary goal is creating open industry standards around container formats and runtimes. See more at <https://opencontainers.org>.

The Podman upstream project resides at github.com in the `containers` project, (<https://github.com/containers/podman>) along with other container libraries and container management tools like `Buildah` and `Skopeo` reside. (See appendix A for a description of some of these tools.)



**Figure 1.3** <https://github.com/containers/> containers is the developer site for Podman and other related container tools.

Podman runs images with the newer OCI format, described in section 1.1.2, as well as the legacy Docker (V2 and V1) format images. Podman runs any image available at container registries like docker.io and quay.io as well as the hundreds of other container registries. Podman pulls these images to a Linux host and launches them in the same way as Docker and Kubernetes. Podman supports all of the OCI Runtimes like `runc`, `crun`, `kata`, and `gvisor` (appendix B), just like Docker.

This book is intended for Linux administrators to help them understand the advantages of using Podman as their primary container engine. You will learn how to configure your systems as securely as possible, but still allow your users to work with containers. One of Podman's



primary use cases is to run containerized applications on single node environments, such as edge devices. Podman along with systemd allow you to manage the entire lifecycle of the application on nodes without human intervention. Podman's goal is to run containers naturally on a Linux box, taking advantage of all of the features of the Linux Platform.

**NOTE** Podman is available for many different Linux distributions and on Mac and Windows platforms. Please refer to appendix C on how to get Podman on your platform.

Application developers are also an intended audience for this book. Podman is a great tool for developers looking to containerize their applications in a secure manner. Podman allows developers to create Linux containers on all Linux distributions. In addition Podman is available on the Mac and Windows platforms where it can communicate with the Podman service running within a VM or on a Linux box available on the network. Podman in action shows you how to work with containers, build container images, and then convert their containerized applications into either single node services to run on edge devices or into Kubernetes based micro services.

Podman and the container tools are open source projects with contributors from many different companies, universities and organizations. Contributors come from all over the world. The projects are always looking to add new contributors to improve them, please refer to appendix D to see how you can join the effort.

Roadmap: In this chapter I first go over a brief overview of Containers and then I explain some key features that make Podman a great tool for working with containers.

## 1.1 A brief overview of containers

Containers are just groups of processes running on a Linux system, which are isolated from each other. Containers make sure that one group of processes do not interfere with other processes on the system. Rogue processes can't dominate system resources preventing other processes from performing their task. Hostile containers prevented from attacking other containers, stealing data or causing denial of service attacks. A final goal of containers is to allow applications to be installed with their own versions of shared libraries that do not conflict with applications requiring different versions of the same libraries. Allow applications to live in a virtualized environment, with a feel that they own the entire system.

Containers are isolated via:

1. Resource constraints (cgroups)

The cgroup man page, cgroup (<https://man7.org/linux/man-pages/man7/cgroups.7.html>). defines cgroups as:

Control groups, usually referred to as cgroups, are a Linux kernel feature which allow processes to be organized into hierarchical groups whose usage of various types of resources can then be limited and monitored.

Examples of resources controlled by Cgroups are:

- a) The amount of memory that a group of processes can use.
- b) The amount of CPU that processes can use.
- c) The amount of network resources a process can use.

The basic idea of cgroups is to control one group of processes from dominating certain system resources in such a way that another group of processes can't make progress on the system.

## 2. Security constraints

Containers are isolated from each other using many security tools available in the kernel. The idea is to block privilege escalation, and prevent a rogue group of processes from hostile acts against the system. Examples:

- a) Dropped Linux capabilities limit the power of root.
- b) SELinux controls access to the file system.
- c) Read-only access to kernel file systems.
- d) SECCOMP to limit the system calls available in the kernel
- e) User namespace to map one group of UIDs in the host to another, allowing access to limited root environments.

Table 1.1 Gives further information and links to find out more information about some of these security features.

**Table 1.1 Advances Linux Security features**

Component	Description	Reference
Linux Capabilities	Linux capabilities subdivide the power of root into distinct capabilities	The capabilities man page is a good overview of the capabilities available. <code>man capabilities</code> ( <a href="https://bit.ly/3A3Ppeg">https://bit.ly/3A3Ppeg</a> )
SELinux	Security Enhanced Linux (SELinux) is a Linux Kernel mechanism which labels every process and every file system object on the system. SELinux Policy defines the rules on how labeled processes interact with label objects. The Linux Kernel enforces the rules.	I wrote the <i>SELinux Coloring Book</i> which is a fun way to help you understand SELinux. ( <a href="https://bit.ly/33plEBD">https://bit.ly/33plEBD</a> ) If you really want to study the subject, check out the SELinux notebook. ( <a href="https://bit.ly/3GxGhkm">https://bit.ly/3GxGhkm</a> )
SECCOMP	SECCOMP is a Linux Kernel Mechanism to limit the number of syscalls to a group of processes on the system. You can remove potentially dangerous syscalls from being called by the processes.	The seccomp man page is a good source of additional information on SECCOMP. <code>man seccomp</code> ( <a href="https://bit.ly/3rnnim1">https://bit.ly/3rnnim1</a> )
User Namespace	The User Namespace allows you to have Linux Capabilities within the group of UIDs and GIDs assigned to the namespace, but not have root capabilities on the host.	The user namespace is fully explained in chapter 3.

### 3. Virtualization technologies (namespaces)

The Linux kernel has a concept called Namespaces which creates virtualized environments where one set of [processes](#) sees one set of resources while another set of processes sees a different set of resources. These virtualized environments eliminate processes' views into the rest of the system giving the processes the feel of a Virtual Machine without the overhead. Examples of namespaces are:

- Network namespace, which eliminates the access to the host network, but gives access to virtual network devices
- Mount namespace, eliminates the view of all of the file system, except the containers file system.
- Pid namespace eliminates the view of other processes on the system, container processes only see the processes within the container.

These container technologies have existed in the Linux Kernel for many years. Security tools for isolating processes started in Unix back in the 1970s, SELinux in 2001. Namespaces were introduced up around 2004 and cgroups around 2006.

**NOTE** Windows container images exist, but this book is concentrating on Linux based containers. Even when running Podman on Windows, you are still working with Linux Containers. Podman on MAC is covered in Appendix E. Podman on Windows is covered in Appendix F.

### 1.1.1 Container images: new way to ship software

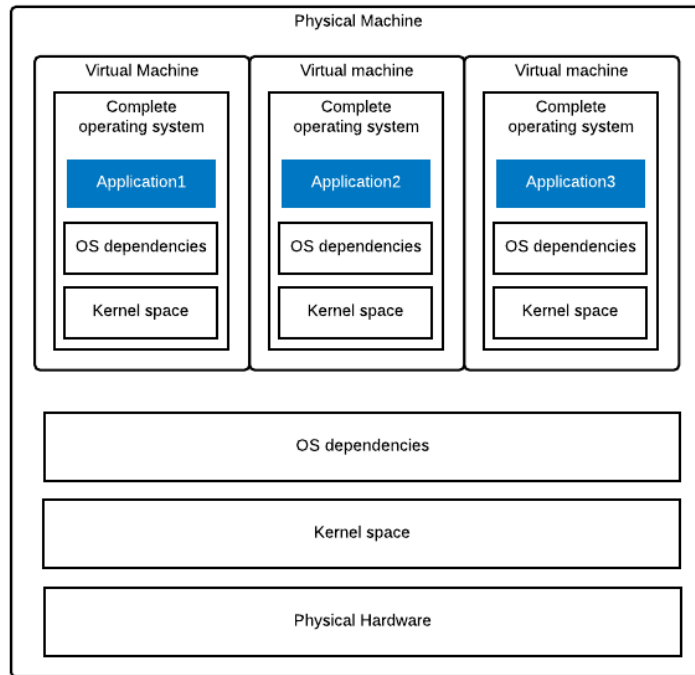
Containers really didn't take off until the Docker project introduced the concept of the container image and a container registry. Basically they created a new way to ship software.

Traditionally installing multiple software applications on a Linux system has led to a problem of dependency management. Before containers, you packaged software using package managers like RPM, Debian Packages, etc. These packages are installed on a host and share the content on the host including shared libraries. When developers test their code, everything might work fine when run on their host machine. Then the quality engineering team tests the software on a different machine with different packages and they see failures. Both teams need to work together to generate the proper requirements. Finally the software is shipped to customers who have many different configurations and software installed leading to further breakage of the application.

Container images solve the dependency management problem by bundling all of the software together into a unit. You ship all of the libraries, executables and configuration files together. The software is isolated from the host, via container technology. In the container image world, you ship all of the software needed to run your application. Usually the only part of the host system that your application is going to interact with is the host kernel.

The developer, quality engineers and customer all run the exact same containerized environment along with the application. This helps to guarantee consistency, and limit the number of bugs caused by misconfiguration.

Containers are often compared to virtual machines, in that they both have the ability to run multiple isolated applications on a single node. When using virtual machines, you need to manage the entire virtual machines operating system as well as the isolated application. You need to manage the lifecycle of the different kernel, init system, logging, security updates, backups etc. The system also has to deal with the overhead of the entire running operation system, not just the application. In the container world all you run is the containerized application, no overhead and no additional OS management. In the diagram below you see three applications running in three different virtual machines.



**Figure 1.4** Physical machine running three applications in three virtual machines.

With virtual machines you end up needing to manage four operations systems. Whereas with containers the three applications run with just their required user spaces. You end up managing just one operating system. As you see in figure 1.3

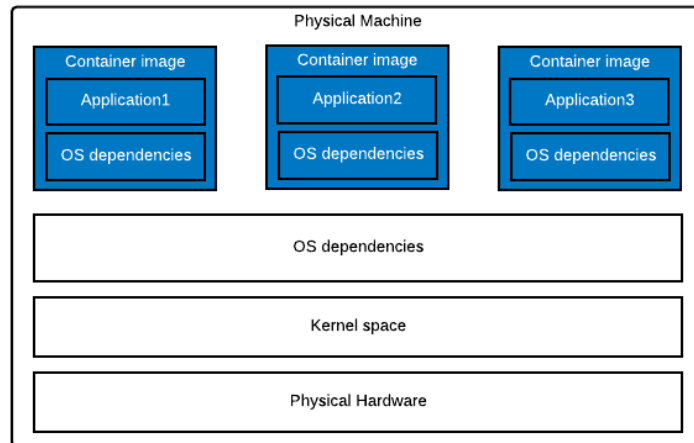
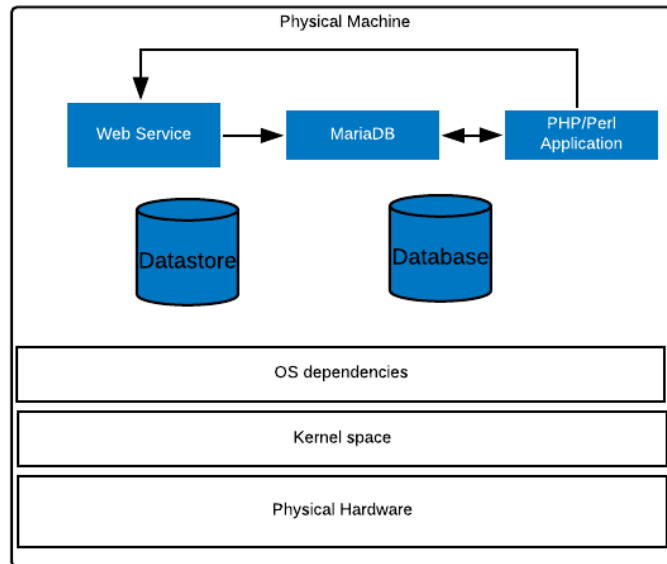


Figure 1.5 Physical machine running three applications in three containerized applications.

#### CONTAINER IMAGES LEAD TO MICROSERVICES

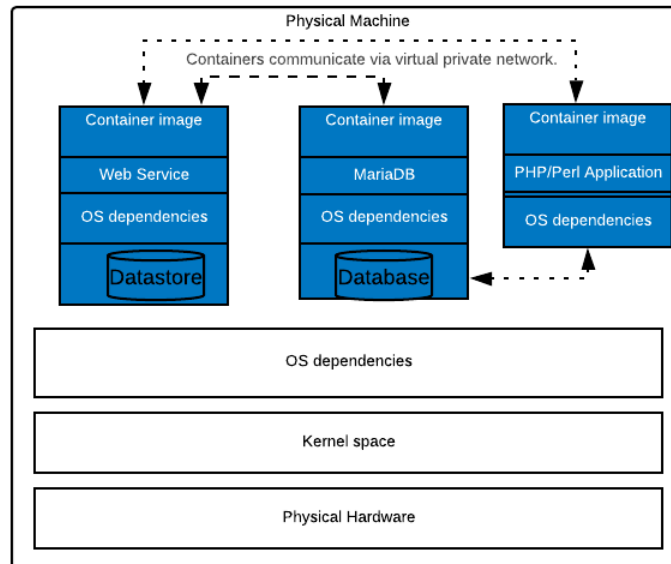
Packing applications inside of container images allows the installation of multiple applications on the same host with conflicting requirements. For example, one application might require a different version of the c library than another, which prevents them from being installed at the same time.



**Figure 1.6 Traditional LAMP Stack (Linux + Apache + MariaDB+PHP/PERL Application) running on a server.**

In containers they can have the correct c library within their container image with each image potentially having different versions of the library specific to the container's application. You can run applications from totally different distributions.

They make it easy to run multiple instances of the same application as you see below. Container images encourage the packaging of a single service or application into a single container. Containers allow you to easily wire multiple applications together via the network.



**Figure 1.7 LAMP stack packaged individually into micro service containers. As containers communicate via network, allows them to be easily moved to other VMs and makes reuse much easier.**

Instead of designing monolithic applications where you have a web front end, a load balancer, and a database, you can build three different container images and then wire them together to build microservices. Microservices allow you and other users to experiment with running multiple databases, web front ends and orchestrate them together. Containerized microservices make the sharing and reuse of software possible.

### 1.1.2 Container image format

A container image consists of three components:

A directory tree containing all of the software required to run your application. The directory is called a rootfs (root filesystem). The software is laid out like it was the root (/) of a linux system.

A JSON file which describes the contents of the rootfs. The executable to be run within the rootfs, the working directory, the environment variables to be used, the maintainer of the executable and other labels to help identify the content of the image are defined in the image JSON file.

You can see the JSON file using the `podman inspect` command. JSON file for the `ubi8` image on [registry.access.redhat.com](https://registry.access.redhat.com).



```
$ podman inspect docker://registry.access.redhat.com/ubi8
{
...
  "created": "2022-01-27T16:00:30.397689Z", #A
  "architecture": "amd64", #B
  "os": "linux", #C
  "config": {
    "Env": [ #D
      "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
      "container=oci"
    ],
    "Cmd": [ #E
      "/bin/bash"
    ],
    "Labels": { #F
      "architecture": "x86_64",
      "build-date": "2022-01-27T15:59:52.415605",
    }
  }
}
```

**#A** Date that the image was created

**#B** Architecture for this image

**#C** Operating system for this image

**#D** Environment variables that the developer of the image wants to be set within the container.

**#E** Default command to be executed when the container starts.

**#F** Labels to help describe the contents of the image. These fields can be free form and do not affect the way images are run, but can be used to search for and describe the image.

Another JSON file called a manifest list, which links multiple images together to support different architectures. This allows users on an `arm64` machine to pull an image with the same name as they would if they were on an `amd64` machine. Podman pulls the image based on the default architecture of the machine, using this manifest list. `Skopeo` is a tool which uses the same underlying libraries as Podman and is available at [github.com/containers/skopeo](https://github.com/containers/skopeo). (See appendix A). Skopeo provides lower level output examining the structures of a container image. In the following example use the `skopeo` command with the `--raw` option to examine images manifest specification JSON file for the `ubi8` image on `registry.access.redhat.com`.

```
$ skopeo inspect --raw docker://registry.access.redhat.com/ubi8
{
  "manifests": [
    {
      "digest":
      "sha256:cbc1e8cea8c78cfa1490c4f01b2be59d43ddbbad6987d938def1960f64bcd02c", #A
      "mediaType": "application/vnd.docker.distribution.manifest.v2+json", #B
      "platform": {
        "architecture": "amd64", #C
        "os": "linux" #D
      },
      "size": 737
    },
    {
      "digest": #E
      "sha256:f52d79a9d0a3c23e6ac4c3c8f2ed8d6337ea47f4e2dfd46201756160ca193308",
      "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
      "platform": {
        "architecture": "arm64",
        "os": "linux"
      },
      "size": 737
    },
    ...
  ]
}
```

**#A** Digest of the exact image that is pulled when the architecture and OS match

**#B** mediaType describes the type of the image, OCI, Docker ...

**#C** The architecture of this image digest "amd64"

**#D** The OS of this image digest "Linux"

**#E** This stanza points to a different image for a different architecture "arm64"

Images use the Linux tar utility to pack the rootfs and the JSON files together. These images are then stored on web servers called container registries. (docker.io, quay.io, artifactory ...)

Container engines like Podman can copy these images to a host and unpack them onto the file system. Then the engine merges the image's JSONs file, along with the engine's builtin defaults and user input to create a new container JSON file, the OCI runtime specification. The JSON file describes how to run the containerized application.

In the last step the container engine launches a small program called a container runtime, `runc`, `crun`, `kata`, `gvisor`, etc. The container runtime reads the container JSON and instruments the containers kernels cgroups, security constraints and namespaces before finally launching the primary process of the container.

### 1.1.3 Container standards

The Open Container Initiative (OCI) standards body defined the standard formats on how container images are stored and defined. The OCI also defined the standard on how container engines run containers. The OCI created the OCI Image Format which standardizes the format of the container images and the images JSON file. They also created the OCI Runtime Specification, which standardized the container's JSON file to be used OCI runtimes.

The OCI standards allow other container engines like Podman<sup>1</sup> to follow the standards and be able to work with all the images stored at container registries, and to run them in the exact same way as all other container engines including Docker. (See figure 1.7)

## 1.2 Why Podman when you have Docker?

I often get asked the question, “Why do you need Podman when you already have Docker?” Well one reason is that **open source is all about choice**. Operating systems have more than one editor, more than one shell, more than one file system, and more than one internet web browser. I believe that Podman’s design is fundamentally better than Docker’s, and brings features that advance the security and use of containers.

### Why have only one way to run containers?

One of Podman’s advantages was that it was created long after Docker existed. Podman developers look at ways to improve on Docker’s design from a totally different perspective. Because Docker was written as open source, Podman shares some of the code and takes advantage of new standards like the Open Container Initiative. Podman works with the open source community to concentrate on developing new features.

In the rest of this section, I cover some of these improvements. Table 1.2 describes and compares features available in Podman and Docker.

**Table 1.2 Podman and Docker feature comparison**

Feature	Podman	Docker	Description
Support all OCI and Docker images	✓	✓	Pull & run container images from container registries, ie quay.io, docker.io. See chapter 2.
Launch OCI container engines	✓	✓	Launch containers runc, crun,kata, gVisor OCI Container engines See appendix B.
Simple command line interface	✓	✓	Podman and Docker share the same CLI. See chapter 2.
Integration with systemd	✓	X	Podman supports running systemd inside of the container. As well as many systemd features. See chapter 7.
Fork/Exec Model	✓	X	Container is a child of the command

<sup>1</sup> Other container engines include Buildah, CRI-O, Containerd and many others.

Fully support user namespace	✓	X	Only Podman supports running containers in separate user namespaces. See chapter 6.
Client Server Model	✓	✓	Docker is a REST API daemon. Podman supports REST API via systemd socket activated service. See chapter 9.
Support docker-compose	✓	✓	Compose scripts work against both REST APIs. Podman's works in rootless mode. See chapter 9.
Support docker-py	✓	✓	docker-py python bindings work against both REST APIs. Podman's works in rootless mode. Podman also supports podman-py, for running advanced features. See chapter 9.
Daemon-less	✓	X	The Podman command runs like a traditional command line tool. While Docker requires multiple root running daemons.
Support Kubernetes like pods	✓	X	Podman supports running multiple containers within the same pod. See chapter 4.
Support Kubernetes yaml	✓	X	Podman can launch containers and pods based on Kubernetes yaml. It can also generate Kubernetes.yaml from running containers. See chapter 8.
Support docker swarm	X	✓	Podman believes the future for orchestrated multi node containers is Kubernetes and does not plan on implementing Swarm.
Customizable registries	✓	X	Podman allows you to configure registries for short name expansion. Docker is hard coded to docker.io when you specify a short name. See chapter 5.

Customizable defaults	✓	X	Podman supports fully customizing all of its defaults including security, namespaces, volumes ... See chapter 5.
Mac OS Support	✓	✓	Podman and Docker support running containers on a Mac via a VM running linux. See appendix E.
Windows Support	✓	✓	Podman and Docker support running containers on a Windows WSL2 or a VM running linux. See appendix F.
Linux Support	✓	✓	Podman and Docker are supported on all major linux distributions. See appendix C.
Containers aren't stopped on software upgrade.	✓	X	Podman is not required to remain running when containers are running. Since the Docker daemon is monitoring containers, by default when it stops all containers stop.

### 1.2.1 Rootless containers

Probably the most significant feature of Podman is its ability to run in rootless mode.

In many situations you do not want to give full root access to your users, but users and developers still need to run containers and build container images. Requiring root access prevents lots of security conscious companies from widespread adoption of Docker. **Podman, on the other hand, can run containers with no additional security features in linux other than a standard login account**

You can run the Docker client as a normal user by adding the user to the `docker` user group (/etc/group), but I believe that granting this access is one of the most dangerous things you can do on a Linux machine.

Access to the docker.socket allows you to gain full root access on the host by running the following command. In the command you are mounting the entire host operating system "/" on the /host directory within the container. The **--privileged** flag turns off all container security, and then you chroot to /host. After the chroot, you are in a root shell at / of the operating system, with full root privileges.

```
$ docker run -ti --name hacker --privileged -v /:/host ubi8 chroot /host
#
```

At this point you have full root privileges on the machine and you can do whatever you want. When you are done hacking the machine, you can simply execute the docker rm command to remove the container, and all records of what you did.

```
$ docker rm hacker
```

When Docker is configured with default file logging, all records of your launching the container are erased. I believe this is far worse than setting up sudo without root, in that at least with sudo, you have the chance to see that I ran sudo in your log files.

With Podman the processes running on the system are always owned by the user and have no capabilities greater than a normal user. Even if you break out of the container, the process is still running as your UID, and all action on the system are recorded in the audit logs. Users of Podman can not simply remove the container and cover up their tracks.

See the rootless chapter for more information.

**NOTE** Docker now has the ability to run rootless similarly to Podman, but almost no one runs it that way. Starting up multiple services in your home directory just to launch a single container has not caught on.

### 1.2.2 Fork/Exec Model

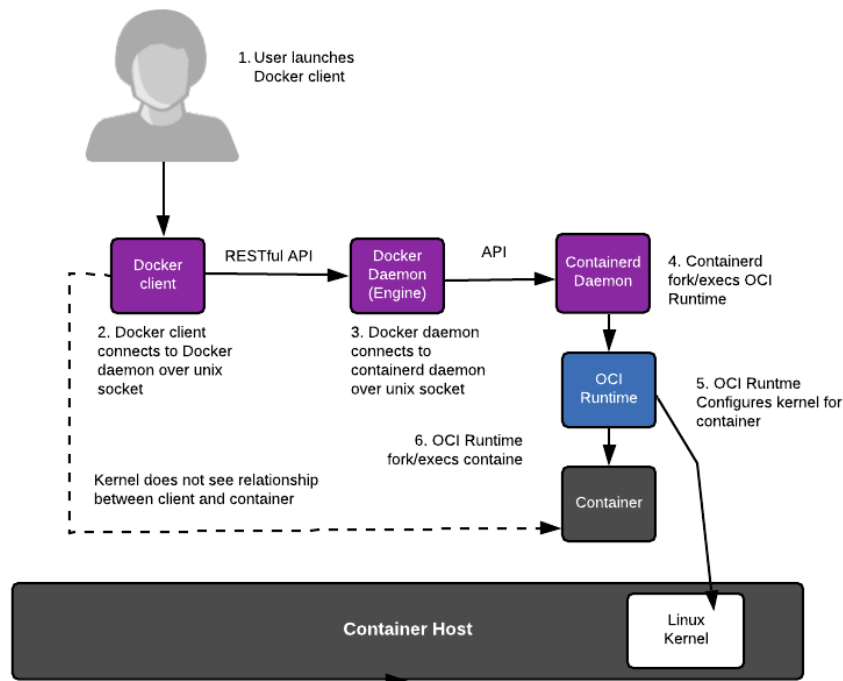
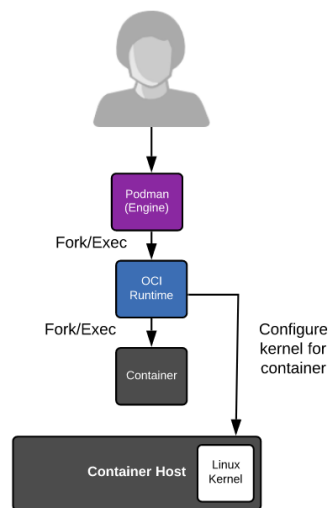


Figure 1.8 Docker client server architecture. Container is a direct descendant of containerD not the docker client. Kernel sees no relationship between the client program and the container.

Docker is built as a REST API Server. Fundamentally Docker is a client-server architecture including multiple daemons. When a user executes the Docker client, they execute a command line tool which connects to the Docker daemon. The Docker daemon then pulls images to its storage and then connects to the containerd daemon which actually finally executes an OCI Runtime which actually creates the container. The Docker daemon then is a communication platform which communicates reads and writes stdin, stdout and stderr of the initial process (PID1) created in the container. The daemon relays all of the output back to the Docker client. Users imagine that the container's processes are just children of the current session but there is a lot of communication going on behind the scenes.

The bottom line is the Docker client communicates with the Docker Daemon, which then communicates with the Containerd daemon, which finally launches an OCI Runtime like runc to launch Pid1 of the container. There is a lot of complexity to running containers in this way, and over the years failures in any of the Daemons have led to all containers shutting down, and often difficult to diagnose what happen



How Podman runs a container

**Figure 1.9 Podman fork/exec architecture. User Launches Podman which executes OCI runtime which then launches the container. Container is a direct descendant of Podman.**

The core Podman engineering team come from an operating system background more grounded in the Unix Philosophy.

Unix and C were designed with the fork/exec model of computing. Basically when you execute a new program, a parent program like the bash shell, forks a new process and then executes the new program as a child of the old program. Podman engineering thought that they could make containers simpler by building a tool that pulls container images from a

container registry, configure container storage and then launch an OCI Runtime which starts the container as a child of our container engine.

In the unix operating system, processes can share content via the file system, and inter process communication (IPC) mechanisms. These features of the operating system enable multiple container engines to share storage without requiring a daemon to be running to control access and share content. The engines do not need to communicate together other than using locking mechanisms provided by the operating systems file systems. Future chapters examine the advantages and disadvantages of this mechanism.

### 1.2.3 Daemon-less

Podman is fundamentally different from Docker, because it is daemon-less. Podman can run all of the same container images as Docker and launch containers with the same container runtimes. But Podman does this without having multiple continuously root running daemons.

Imagine you have a web service that you want to run at boot time. The web service is packaged up in a container, so you need a container engine. In the Docker case you need to set it up to be running on your machine with each of the daemons running and accepting connections. Next you launch the Docker client to start the web service. Now you have your containerized application running as well as all of the Docker daemons. In the Podman case, you use the Podman command to launch your container and Podman goes away. Your container continues to run without the overhead of running the multiple daemons. Less overhead is incredibly popular on low end machines like IOT devices and edge servers.

### 1.2.4 User Friendly Command Line

One of the great features of Docker is the simple command line interface. There have been other container command lines like `RKT`, `lxc` and `lxcfd`, but they have their own command line interfaces. The Podman team realized early on that it wouldn't gain market share if Podman had its own command line interface. Docker was the dominant tool and almost everyone who had played with containers had done it with its CLI. Also if you googled how to do something with a container, invariably you get an example using the Docker command line. Right from the start Podman had to match the Docker Command line. Quickly the motto began: if you want to replace Docker with Podman, all you had to do was: `alias docker=podman`

With this command you can continue to type in your Docker commands, but Podman actually runs your containers. If the Podman command line differs from Docker it is considered a bug in Podman, and users demand Podman be fixed to make the tools match.

There are a few commands like ``docker swarm`` that Podman doesn't support, but for the most part I believe Podman is a complete replacement for the Docker CLI.

Many distributions supply a package called `podman-docker`, which set's up the alias from `docker` to `podman` and links all of the man page. The alias means when you type ``docker ps``, the ``podman ps`` command runs. If you execute ``man docker ps`` the podman ps man pages shows up.



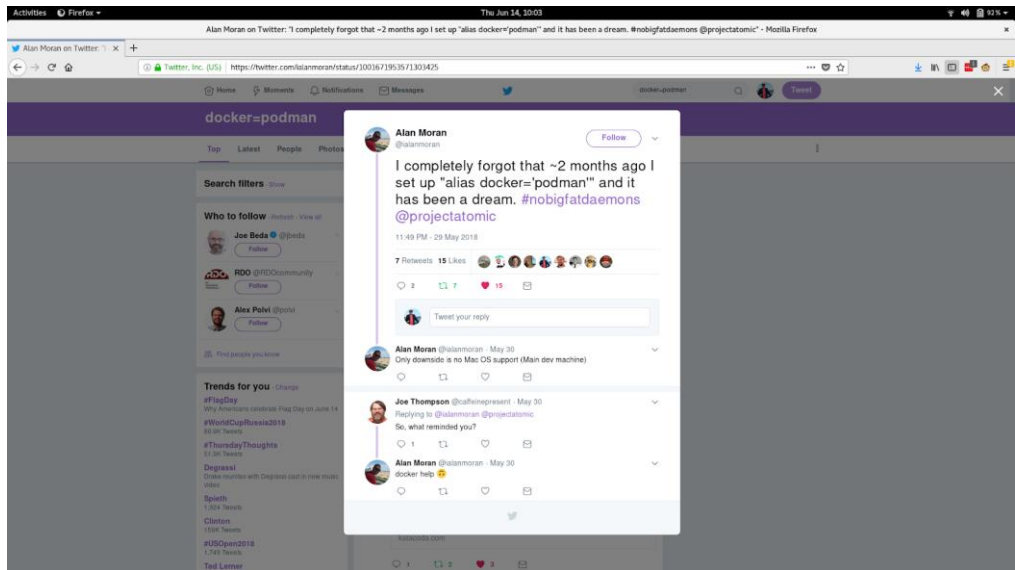


Figure 1.10 Twitter Tweet about “alias docker=podman”

Back in 2018, Alan Moran tweeted that *“I completely forgot that ~2 months ago I set up “alias docker= podman” and it has been a dream. #nobigfatdaemons...”*. Joe Thomson responded “So, what reminded you?” and Alan Moran answered “docker help”. And podman help showed up.

### 1.2.5 Support for REST API

Podman can be run as a socket activated REST API service. This allows remote clients to manage and launch Podman containers. Podman supports the Docker API as well as the Podman API for advanced Podman features. Through the use of the Docker API, Podman supports docker-compose and other users of the docker-py python bindings. This means that even if you built your infrastructure around using the Docker socket for launching containers, you can simply replace Docker with the Podman service and continue to use your existing scripts and tools. Chapter 9 covers the Podman service.

The Podman REST API also allows remote podman clients on Mac, Windows and Linux systems to interact with Podman containers on a Linux machine. Appendix E and F covers Podman use on Mac and Windows machines.

### 1.2.6 Integration with systemd

Systemd is the fundamental init system in the operating systems. The init process on a Linux system, is the first process that is started by the kernel on boot. Therefore the init system is the ancestor of all processes and can monitor them all. Podman wants to fully integrate the running of containers with the init system. Users want to use systemd to start and stop

containers at boot time. Containers should work with socket activation, systemd notifications to tell systemd when a container was fully activated, and be managed by systemd cgroups. Basically containers work as services in systemd unit files. Many developers want to run systemd within a container, in order to run multiple systemd defined services within a container.

But the upstream Docker community disagreed and denied all pull requests which attempted to integrate systemd into Docker. They believe that Docker should manage the lifecycle of the container, they do not want to accommodate users who want to run systemd in a container.



Figure 1.11 Docker employee badge at DockerCon EU

The upstream Docker community believed that the Docker daemon should be the controller of processes, it should manage the lifecycle of containers, and start and stop them at boot time, versus systemd. The problem is there are way more features in systemd than in Docker, like startup ordering, socket activation, service ready notifications etc.

When Podman was designed, the developers wanted to make sure it fully integrated with systemd. When you run systemd inside of a container, Podman sets up the container the way systemd expects and allows it to simply run as pid1 of the container with limited privileges. Podman allows you to run services within the container the same way they run them on a system or in a VM, via systemd unit files. Podman supports socket activation, service notifications and many other systemd unit file features. Podman makes it simple to generate systemd unit files with best practices for running containers within a systemd service. See chapter 7 on systemd integration.

The containers project, (<https://github.com/containers>) where Podman, container libraries, and other container management tools reside want to embrace all features of the Operating System, and want to fully integrate with it. Chapter 7 explains Podman integration with systemd.

### 1.2.7 Pods

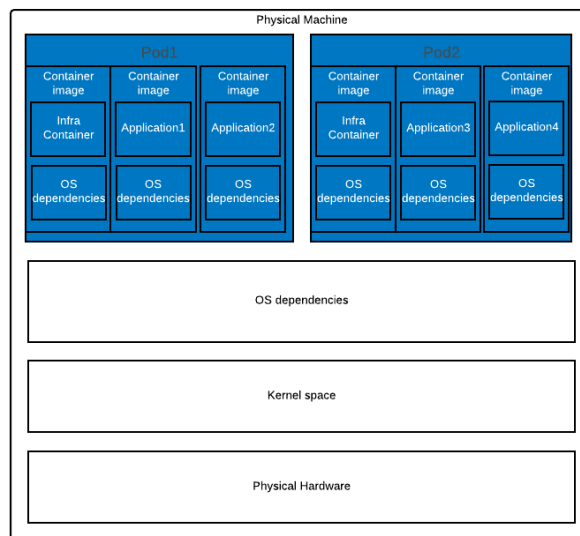


Figure 1.12 Two Pods running on a host. Each pod runs two different containers along with the infra container.

One advantage of Podman is actually described in its name, see chapter 4. Podman actually stands for Pod Manager. Pod was a concept originated in the Kubernetes project. As the official

Kubernetes documentation puts it: “A pod (as in a pod of seals, hence the logo, or pea pod) is a group of one or more containers, with shared storage/network resources, and a specification for how to run the containers.” Podman works with either a single container at a time like Docker, or it can manage groups of containers together in a Pod. One of the design goals of containers is to separate services into single containers, micro-services. Then you combine containers together to build larger services. Pods allow you to group multiple services together to form a larger service and be managed as a single entity. One of the goals of Podman is to allow you to experiment with Pods.

Podman has the `podman generate kube` command which allows you to generate Kubernetes yaml files from running containers and pods, as you can see in chapter 7. Similarly it has the `podman play kube` command which allows you to play Kubernetes yaml files and generate pods and containers on your host. I like to suggest that Podman is for running pod’s and containers on a single host, while Kubernetes is used to take your pods and containers and run them on multiple machines, and all through your infrastructure.

Other projects, like KIND (<https://kind.sigs.k8s.io/docs/user/rootless>), are experimenting with running pods with Podman under the guidance of Kubernetes.

## 1.2.8 Customizable Registries

Container engines like Podman support the concept of pulling images using short names, an image name, like `ubi8`, without specifying the registry in which it resides, `registry.access.redhat.com`. Complete image names include the name of the container registry they were pulled from, `registry.access.redhat.com/library/ubi8:latest`. Table 1.3 shows the components of the image name broken out.

**Table 1.3. Short name to container image name table**

Name	Registry	Repo	Name	Tag
Short name			<b>ubi8</b>	
Complete name	registry.access.redhat.com	library	<b>ubi8</b>	latest

Docker is hard coded to always pull from <https://docker.io> when using a short name. If you want to pull an image from a different container registry, you have to fully specify the image. In the following example I attempt to pull `ubi8/httpd-24` and it fails, because the container image is not on `docker.io`. The image is on `registry.access.redhat.com`.

```
# docker pull ubi8/httpd-24
Using default tag: latest
Error response from daemon: pull access denied for ubi8/httpd-24, repository does not exist
or may require 'docker login': denied: requested access to the resource is denied
```

So if I want to use `ubi8/httpd-24` I am forced to type the entire name including the registry.

```
# docker pull registry.access.redhat.com/ubi8/httpd-24
```

The Docker engine enables docker.io to have advantage over other container registries, as the preferred registry. Podman was designed to allow you to specify multiple registries, similar to what you can do with `dnf`, `yum`, and `apt` tools for installing packages. You can even remove docker.io totally from the list. If you attempt to pull `ubi8/httpd-24` with Podman, Podman presents you with a list of registries to choose from.

```
$ podman pull ubi8/httpd-24
? Please select an image:
  registry.fedoraproject.org/ubi8/httpd-24:latest
  ► registry.access.redhat.com/ubi8/httpd-24:latest
  docker.io/ubi8/httpd-24:latest
  quay.io/ubi8/httpd-24:latest
```

Once you make your decision, Podman records the short name alias and no longer prompts and uses the previously selected registry.

Podman supports lots of other features like blocking registries, only pulling signed images, setting up image mirrors, and specifying hard coded short names, so that specify short names map directly to the long names. (See chapter 5)

## 1.2.9 Multiple transports

Podman supports many different container image sources and targets called `transports` (See table 1.4). Podman can pull images from container registries and from local containers storage, but also supports images stored in OCI format, OCI Tar format, legacy Docker tar format, directory format, and even directly from the Docker daemon. Podman commands can easily run images from each of the formats.

**Table 1.4 Podman supported transports**

Transport	Description
container registry (docker)	References a container image stored in a remote container image registry, web site. Registries store and share container images. For example, docker.io, quay.io.
oci	A container image compliant with the Open Container Image Layout Specification. The manifest and layer tarballs as individual files are located in the local directory.
dir	A container image, compliant with the Docker image layout, similar to the `oci` transport but stores the files using the legacy "docker" format.
docker-archive	A container image in Docker image layout which is packed into a TAR archive.
oci-archive	A container image compliant with the Open Container Image Layout Specification which is packed into a TAR archive.
docker-daemon	An image stored in the Docker daemon's internal storage.
<b>container-storage</b>	A container image located in a local storage. Podman defaults to using container-storage for local images.

### 1.2.10 Complete customizability

Container engines tend to have lots of builtin constants, like the namespaces they run with, whether or not SELinux is enabled, which capabilities containers run with. With Docker most of these values are hard coded and can not be changed by default. Podman, on the other hand, has a very customizable configuration.

Podman has its builtin defaults but defines three locations for its configuration files to be stored:

- `/usr/share/containers/containers.conf`, which is where a distribution can define the changes that the distribution likes to use.
- `/etc/containers/containers.conf`, where they can set up their system overrides.
- `$HOME/.config/containers/containers.conf`, which can be specified only in rootless mode.

The configuration files allow you to configure Podman to run the way you want by default. You can even run with more security by default if you choose.

### 1.2.11 User Namespace support

Podman is fully integrated with the user namespace. Rootless mode relies on user namespaces, to allow for multiple UIDs to be assigned to a user. User namespace provides isolation between users on a system, so that you can have multiple rootless users running containers with multiple uids, all isolated from each other.

User namespace can be used to isolate containers from each other. Podman makes it simple to launch multiple containers, each one with a unique user namespace. The kernel then isolates the processes from host users as well as each other based on UID separation.

Docker only supports running containers in a single separate user namespace meaning all containers run within the same user namespace. Root in one container is the same as root in another container. It does not support running each container in a different user namespace, this means containers attack each other from a user namespace perspective. Even though Docker supports this mode, almost no one runs containers with Docker in a separate user namespace.

## 1.3 When not to use Podman

Like Docker, Podman is not a container orchestrator. Podman is a tool for running container workloads on a single host in either rootless or rootful mode. Higher level tools are required if you want to orchestrate running containers on multiple machines.

I believe the best tool for doing this now is Kubernetes. Kubernetes won the container orchestrator war when it comes to mind share. Docker has an orchestrator called Swarm, which had some popularity, but now seems to be out of favor. Because the Podman team believes that Kubernetes is the way to go for containers on multiple machines, Podman does not support swarm functionality.

Podman has been used for different orchestrators and is used for GRID/HPC computing and even open source developers have added it under Kubernetes front ends.

## 1.4 Summary

- Containers technology has been around for many years, but the introduction of container images and container registries, allows developers a better way to ship software
- Podman is an excellent container engine suitable for almost all of your single node container projects. It is useful for developing, building and running containerized applications
- Podman is as simple to use as Docker, with the exact same command line interface
- Podman supports a REST API, which allows remote tools and languages including Docker-compose to work with Podman containers.
- Podman includes such notable features over Docker like user namespace support, multiple transports, customizable registries, integration with systems, fork/exec model, out-of-the-box rootless mode
- Podman is a more secure way to run containers.



# 2

## Command line

### This chapter covers

- The Podman command line
- Running an OCI application
- Differences between containers and images
- Building an OCI Based Image

Podman is an excellent tool for running and building containerized applications. In this chapter, you'll get started by building a simple web application to demonstrate commonly used features of the Podman command line.

If you don't have Podman installed on your machine, you can jump to appendix C, Getting Podman, and then return here. This chapter assumes that Podman 4.1 or newer is already installed. Older versions of Podman probably work fine, but all examples were tested with Podman 4.1.

The example base image I use is the `registry.access.redhat.com/ubi8/httpd-24` image.

**NOTE** Universal Base Images (UBI) can be used anywhere, but container software that is maintained and vetted by Red Hat, and when run on a Red Hat based operating system, is fully supported. There are hundreds of Apache images which work similarly to this image, that you can also try out.

Chapter 2 shows how Podman is a great tool for working with containers. In this chapter I walk you through running the scenario you might use to build a containerized application. You launch a container, modify its contents, create an image and ship it to a registry. Then it explains how you can do this in an automated way, to maintain the security of your container image. Through it all you get exposed to many of the Podman command line interfaces, and get a good understanding of how to work with Podman.

If you are an experienced Docker user, you probably just want to skim through this chapter. You will know a lot of it, but there are many interesting features that Podman has like the ability to mount container images (section 2.2.10) and different transports (section 2.2.4.1).

**NOTE** Podman is an open source project under heavy development. Podman is packaged and provided on many different Linux distributions as well as Mac and Windows. These distributions might be shipping older versions of Podman without some of the current features covered in this book. Some examples in this book assume Podman v4.1 or later. If an example does not work, please attempt to update your version of Podman to the latest version. Refer to appendix C for further information on how to get Podman.

Let's start by running our first container.

## 2.1 Working with containers

There are thousands of different container images sitting at container registries. Developers/Administrators/Quality Engineers and general users primarily use the `podman run` command to pull down and run/test/explore these container images. In order to start building out containerized applications, the first thing you need to do is start working with a base image. In our examples you pull and run the `registry.access.redhat.com/ubi8/httpd-24` image to container storage in your home directory and start exploring inside the container.

### 2.1.1 Exploring containers

In this section you will examine a typical podman command, step by step. You will execute the `podman run` command which reaches out to the `registry.access.redhat.com` container registry and begins pulling down the image and storing it locally in your home directory.

```
$ podman run -ti --rm registry.access.redhat.com/ubi8/httpd-24 bash
```

Now I will break down the command that you just executed. By default the `podman run` command executes the containerized command in the foreground until the container exits. In this case you end up at a bash prompt running within the container and showing the `bash-4.4$` prompt. When you exit this bash prompt, Podman stops the container.

In this example you used two options `-t` and `-i`, as `-ti`, which tells Podman to hook up to the terminal. Connecting to the input, output and error stream of the bash process within the container to your screen. This allows you to interact within the container.

```
$ podman run -ti --rm registry.access.redhat.com/ubi8/httpd-24 bash
```

The `--rm` option tells Podman to delete the container as soon as the container exits, freeing up all of the container's storage.

```
$ podman run -ti --rm registry.access.redhat.com/ubi8/httpd-24 bash
```

Next specify the container image, `registry.access.redhat.com/ubi8/httpd-24`, that you are working with. The Podman command reaches out to the container registry at `registry.access.redhat.com` and begins copying down the `ubi8/httpd-24:latest` image.

Podman copies multiple layers (Called blobs) as shown below and stores them in the local container storage. You see the progress as the image layers are pulled down. Some images are rather large and can take a long time while being pulled down. If you later run a different container on the same image, Podman skips the pulling image step since you already have the correct image in local container storage.

```
$ podman run -ti --rm registry.access.redhat.com/ubi8/httpd-24 bash
Trying to pull registry.access.redhat.com/ubi8/httpd-24:latest... #A
Getting image source signatures
Checking if image destination supports signatures
Copying blob 296e14ee2414 skipped: already exists #B
Copying blob 356f18f3a935 skipped: already exists #B
Copying blob 359fed170a21 [=====>-----] 11.8MiB / 16.2MiB #B
Copying blob 226cafc3a0c6 [=====>-----] 10.1MiB / 61.1MiB #B
```

#A Contact with the registry

#B Layer pulling is skipped

Finally you specify the executable to be run within the container, in this case bash.

```
$ podman run -ti --rm registry.access.redhat.com/ubi8/httpd-24 bash
...
bash-4.4$
```

**NOTE** Images almost always have default commands that they execute. You only have to specify a command if you want to override the default application that the image runs with. In the case of `registry.access.redhat.com/ubi8/httpd-24` image, it runs the Apache web server.

While inside the bash shell of container `cat /etc/os-release` file and notice you are probably in a different OS or a different version versus the `/etc/os-release` outside of the container. Explore around in the container and notice how different it is from your host environment.

```
bash-4.4$ grep PRETTY_NAME /etc/os-release
PRETTY_NAME="Red Hat Enterprise Linux 8.4 (Ootpa)"
```

On my host on a different terminal the same command outputs

```
$ grep PRETTY_NAME /etc/os-release
PRETTY_NAME="Fedora Linux 35 (Workstation Edition Prerelease)"
```

Back inside the container you notice that there are a lot fewer commands available.

```
bash-4.4$ ls /usr/bin | wc -l
525
```

Whereas on the host you see:

```
$ ls -l /usr/bin | wc -l
3303
```

Execute the `ps` command to see what processes are running inside of the container.

```
$ ps
PID TTY          TIME CMD
1 pts/0    00:00:00 bash
2 pts/0    00:00:00 ps
```

You only see two processes. The bash script and the ps command. Needless to say, on my host machine, there are hundreds of processes running. (Including these two processes.)

You can further explore the inside of the container, to gain an understanding of what is going on within a container.

When you are done, you exit the bash script and the container shuts down. Since you ran with the `--rm` option, Podman removes all of the container storage and deletes the container. The container image remains in `container/storage`.

Now that you have explored the inner workings of a container, it is time to start working with the default application within the container.

### 2.1.2 Running the containerized application

In the previous example you pulled and ran bash within a containerized application, but you did not run the application that the developer intended us to run. In this next example you are going to run the actual application, by removing the command and running with a couple of new options.

First remove the `-ti` and the `--rm` options, since you want the container to remain running when the podman command exits. You are not a shell running within the container interactively since it is just running the containerized web service.

```
$ podman run -d -p 8080:8080 --name myapp registry.access.redhat.com/ubi8/httpd-24
37a1d2e31dbf4fa311a5ca6453f53106eaae2d8b9b9da264015cc3f8864fac22
```

The first option to notice is the `-d` (`--detach`) option which tells Podman to launch the container and then detach from it. Basically run the container in the background. The Podman command actually exits and leaves the container running. Chapter 6 goes much deeper into what is going on behind the scenes.

```
$ podman run -d -p 8080:8080 --name myapp registry.access.redhat.com/ubi8/httpd-24
```

The `-p` (`--publish`) option tells Podman to publish or bind the container port 8080 to the host port 8080 when the container is running. With the `-p` option, the field before the colon refers to the host port, while the port after the colon refers to the container port. In this case you see that the ports are the same. If you specify only one port, Podman considers this port a container port, and randomly picks a host port on which the container port is bound. You can use the `podman port` command to discover which ports are bound to a container.

```
$ podman port myapp
8080/tcp -> 0.0.0.0:8080 #A
```

**#A** Shows that port 8080/tcp inside of the container is bound to all of the host networks (0.0.0.0) at port 8080.

By default containers are created within their own network namespace, meaning they are not bound to the host network but to their virtualized network. Suppose I execute the container without the `-p` option. In that case the Apache server within the container binds to the network

interface within the container's network namespace, but Apache is not bound to the host network.

Only processes within the container are able to connect to port 8080 to communicate with the webserver. By executing the command with the `-p` option Podman connects the port from inside of the container to the host network at the specified port. The connection allows external processes like a web browser to be able to read from the web service.

The `-p` option can map port numbers inside of the container to different port numbers outside of the container.

**NOTE** If you are running containers in rootless mode, covered in chapter 3, Podman users are by default not permitted to bind to ports < 1024 by the kernel. Some containers want to bind to lower ports like port 80, which is allowed inside of the container but `-p 80:80` fails since 80 is less than 1024. Using `-p 8080:80` causes Podman to bind the host's port 8080 to port 80 within the container. The upstream Podman repo contains troubleshooting information on problems like binding to ports less than 1024 and many others. <https://github.com/containers/podman/blob/main/troubleshooting.md>

```
$ podman run -d -p 8080:8080 --name myapp registry.access.redhat.com/ubi8/httpd-24
```

In the example name the container `myapp` using the `--name myapp` option. Specifying a name makes it easier to find the container and it allows you to specify a name that can then be used for other commands. For example, `podman stop myapp`. If you don't specify a name, Podman automatically generates a unique container name along with a container ID. All of the Podman commands that interact with containers can use either the name or the ID.

```
$ podman run -d --name myapp -p 8080:8080 registry.access.redhat.com/ubi8/httpd-24
```

When the `podman run` command completes, the container is running. Since this container is running in detached mode, Podman prints out the container id and exits, but the container remains running.

```
$ podman run -d -p 8080:8080 --name myapp registry.access.redhat.com/ubi8/httpd-24
37a1d2e31dbf4fa311a5ca6453f53106eaae2d8b9b9da264015cc3f8864fac22
```

Now that the container is running you can launch a web browser to communicate with the web server inside of the container at localhost port 8080.

```
$ web-browser localhost:8080
```

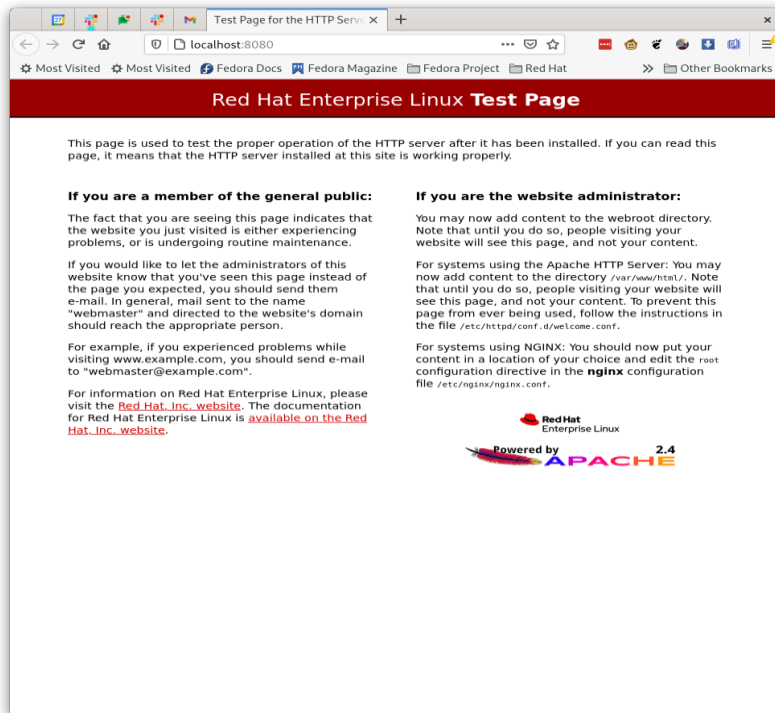


Figure 2.1 Web browser window connecting to the ubi8/httpd-24 container running in Podman.

Congratulations, you have launched your first containerized application.

Now imagine you want to start another container, you can just execute a similar command with a couple of changes.

```
$ podman run -d -p 8081:8080 --name myapp1 registry.access.redhat.com/ubi8/httpd-24
fa41173e4568a8fa588690d3177150a454c63b53bdfa52865b5f8f7e4d7de1e1
```

Notice you need to change the name of the container to **myapp1**, otherwise the podman run command fails with the myapp name because the container previously existed. Also you need to change the -p option to use **8081** for the host port, because the previous container, myapp, is currently running and is bound to port 8080. The second container isn't allowed to bind to port 8080 until the first container exits.

```
$ podman run -d -p 8081:8080 --name myapp1 registry.access.redhat.com/ubi8/httpd-24
```

The `podman create` command is almost identical to the `podman run` command. The create command pulls the image if it is not in container storage and configures the container information to make it ready to run, but never executes the container. It is often used together with the `podman start` command described in section 2.1.4. You might want to create a container and then later use a systemd unit file to start and stop the container.

### FAVORITE PODMAN RUN OPTIONS

- `--user USERNAME` Tells Podman to run the container as a specific user defined in the image. By default Podman will run the container as root, unless the container image specifies a default user.
- `--rm` automatically remove the container when it exits
- `--tty - (t)` allocates a pseudo-tty and attaches it to the standard input of the container.
- `--interactive (-i)` connects stdin to the primary process of the container. These options give you an interactive shell within the container.

Use the `man podman-run` command for information about all options.

**NOTE** There are dozens of podman run options available, allowing you to change security features, namespaces, volumes, and so on. Some of these I use and explain throughout the book. Refer to the `podman-run` man page for a description of all of the options. Most of the `podman create` options defined in table 2.1 are also available for podman run.

You see the container is up and running, now it is time to stop the container to get to the next step.

### 2.1.3 Stopping containers

You have two containers running and have tested them by running a web browser against them. In order to continue the development, by actually adding some content to the web page, you can stop the containers using the `podman stop` command.

```
$ podman stop myapp
```

The stop command stops the container started with the previous podman run command.

When stopping a container, Podman examines the running container and sends a stop signal, usually SIGTERM to the primary process (pid1) of the container and then by default waits ten seconds for the container to stop. The stop signal tells the primary process within the container to exit gracefully. If the container doesn't stop within 10 seconds Podman sends the SIGKILL signal to the process forcing the container to stop. The ten second wait gives the processes in the container time to cleanup and commit changes.

The default stop signal can be changed for a container using the `podman run --stop-signal` option. Sometimes the primary or init process of a container ignores SIGTERM. For example containers which use SystemD as the primary process inside of a container. SystemD ignores SIGTERM and specifies that it be shut down using the SIGRTMIN+3 (signal #37) signal. The stop signal can be embedded in container images as I describe in the `podman build` section 2.3.

Some containers ignore the SIGTERM stop signal, which means you have to wait 10 seconds for the container to exit. If I know the container ignores the default stop signal and I don't care about the container cleaning up, I can just add the option `-t 0` option to `podman stop` to send the SIGKILL signal right away.

```
$ podman stop -t 0 myapp1
myapp1
```

Podman has a similar command `podman kill`, which sends the specified kill signal. The `podman kill` command can be useful when you want to send signals into the container without actually wanting to stop the container.

#### FAVORITE PODMAN STOP OPTIONS

- `--timeout (-t)` sets the timeout, `-t 0` sends the SIGKILL without waiting for the container to stop.
- `--latest (-l)` is a useful option to allow you to stop the last created container rather than having to use the container name or container id. Most Podman commands that require you to specify a container name or id, also accept the `--latest` option. Only available on Linux machines.
- `--all` tells Podman to stop all running containers. Similarly to `--latest`, Podman commands which require a container name or container id parameter, also take the `--all` option.

### 2.1.4 Use the `man podman-stop` command for information about all options.

Eventually, your system has lots of stopped containers, sometimes you need to restart them, for example if the system was rebooted. Another common use case is to first create a container and later start it. The next section explains how to start a container.

### 2.1.5 Starting containers

Now that the container you created was stopped, you might want to start it back up again using the following command:

```
$ podman start myapp
myapp #A
```

**#A** Start command prints the names of the containers that were started.

The `podman start` command starts one or more containers.

This command will output container id, indicating that your container is up and running. You can now reconnect to it with a web browser. One common use case for `podman start` is to start a container after a reboot, to start all of the containers that were stopped during shutdown.

#### FAVORITE PODMAN START OPTIONS

- `--all` starts all of the stopped containers in container storage.
- `--attach` attaches your terminal to the output of the container.
- `--interactive (-i)` attaches the terminal input to the container.

Use the `man podman-start` command for information about all options.

After you've been using Podman for a while, and pulled down and run many different container images, you might want to figure out which containers are running. Or which containers you have in local storage. You need to be able to list these containers.



## 2.1.6 Listing containers

You can list the running containers and all of the containers that were previously created. Use the `podman ps` command to list containers.

```
$ podman ps
CONTAINER ID IMAGE                                COMMAND                                CREATED      STATUS
PORTS          NAMES
b1255e94d084 registry.access.redhat.com/ubi8/httpd-24:latest /usr/bin/run-http... 6 minutes
ago Up 4 minutes ago 0.0.0.0:8080->8080/tcp myapp
```

Notice the `podman ps` command by default lists the running containers. Use the `--all` option to see all of the containers.

```
$ podman ps --all
CONTAINER ID IMAGE                                COMMAND                                CREATED      STATUS
PORTS          NAMES
b1255e94d084 registry.access.redhat.com/ubi8/httpd-24:latest /usr/bin/run-http... 9 minutes
ago Up 8 minutes ago 0.0.0.0:8080->8080/tcp myapp
3efee4d39965 registry.access.redhat.com/ubi8/httpd-24:latest /usr/bin/run-http... 7 minutes
ago Exited (0) 3 minutes ago 0.0.0.0:8081->8080/tcp myapp1
```

### FAVORITE PODMAN PS OPTIONS

- `--all` tells Podman to list all containers rather than just running containers.
- `--quiet` tells Podman to only print the container ids
- `--size` tells Podman to return the amount of disk space currently used for each container other than the images they are based on.

Use the `man podman-ps` command for information about all options.

Now that you know all of the containers you have on the system, you might want to inspect their internals.

## 2.1.7 Inspecting containers

To fully understand a container, sometimes you want to know which image a container was based on, or what environment variables a container gets by default or what are the security settings used for a container. The `podman ps` command gives us some data about the containers, but if you want to really examine information about the container, then you can use the `podman inspect` command.

The `podman inspect` command can also be used to inspect images, networks, volumes and pods. The `podman container inspect` command is also available and specific to containers. But most users just type the shorter `podman inspect` command.

```
$ podman inspect myapp
[
  {
    "Id": "240271ae90480d3836b1477e5c0b49fbd3883846ca474e3f6effdfb271f4ff54",
    "Created": "2021-09-27T05:27:47.163828842-04:00",
    "Path": "container-entrypoint",
    "Args": [
      "/usr/bin/run-httpd"
    ],
    ...
  ]
]
```

As you can see the `podman inspect` command outputs a large json file, 307 lines on my machine. All of this information is eventually handed down the the OCI runtime to launch the container. When using the `inspect` command, it is often better to pipe its output to `less` or `grep` to find particular fields you are interested in. Or you can use the format option.

If you want to to examine the command executed when you start the container, execute:

```
$ podman inspect --format '{{ .Config.Cmd }}' myapp #A
[/usr/bin/run-httpd]
#A Inspect is displaying data from the OCI Image Specification.
```

Or if you wanted to see the stop signal:

```
$ podman inspect --format '{{ .Config.StopSignal }}' myapp
15 #A
#A The default stop signal for all containers is 15 (SIGTERM)
```

#### FAVORITE PODMAN INSPECT OPTIONS

- `--latest (-l)` is handy in that it allows you to quickly inspect the last created container rather than specifying the container name or container id.
- `--format` is useful as you see above to extract particular fields out of the json.
- `--size` adds the amount of disk space the container is using. Gathering this information takes a long time, so it is not done by default.

Use the `man podman-inspect` command for information about all options.

After you inspect a container, you might realize that you no longer need that container taking up storage, so you need to remove it.

### 2.1.8 Removing containers

If you are done using a container, you often want to remove the container to free up disk space, or to reuse the container name. Remember you started a second container called **myapp1**, you no longer need it so you can remove it. Make sure to stop the container (section 2.1.3) before removing it. Then use the `podman rm` command to remove container:

```
$ podman rm myapp1
3efee4d3996532769356ffea23e1f50710019d4efc704d39026c5bffd6aa18be
```

#### FAVORITE PODMAN RM OPTIONS

- `--all` option is useful if you want to remove all of your containers
- `--force` option tells Podman to stop all of the running containers when removing.

Use the `man podman-rm` command for information about all options.

Now that you understand a few commands it is time to start modifying the running container.

### 2.1.9 Execing into a container

Often when a container is running, you might want to start another process within the container to debug/examine what is going on, or in some cases modify some of the content that the container is using.

Imagine you wanted to go into your container and modify the web page that it is showing. You can exec into the container using the `podman exec` command. You use the `--interactive` (`-i`) option to allow us to execute commands within the container. You need to specify the name of the container `myapp` and finally execute the `bash` script while in the container. If you stopped the `myapp` container you need to restart it, since `podman exec` only works on running containers.

In the example below you exec'd a `bash` process into the container to create the `/var/www/html/index.html` file. I write HTML content that causes the containerized website to display the bolded Hello World.

```
$ podman exec -i myapp bash -c 'cat > /var/www/html/index.html' << _EOF
<html>
<head>
</head>
<body>
<h1>Hello World</h1>
</body>
</html>
_EOF
```

Execing back into the container a second time you can see that the file was successfully modified. This shows that modifications to a container via `exec`, are permanent to the container, and will remain even if you stopped and restarted the container. Key difference between `podman run` and `podman exec` is that `run` will create a new container off of an image with processes running inside, while `exec` starts processes inside of existing containers.

```
$ podman exec myapp cat /var/www/html/index.html
<html>
<head>
</head>
<body>
<h1>Hello World</h1>
</body>
</html>
Now let's connect a web browser to the container to see if the content has changed.

$ web-browser localhost:8080
```

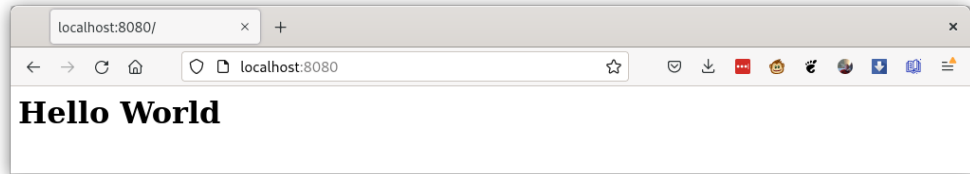


Figure 2.2 Web browser window connecting to the `ubi8/httpd-24` container running in Podman with update Hello World HTML.

#### FAVORITE PODMAN EXEC OPTIONS

- `--tty` connects a tty to the exec session
- `--interactive, -i` option tells Podman to run in interactive mode, meaning you can interact with an exec'd program like a shell.

Use the `man podman-exec` command for information about all options.

Now that you have created an application you might want to share it with others. First you need to commit the container to an image.

#### 2.1.10 Creating an image from a container

Developers often run containers from a base image to create a new container environment. Once they are done, they pack this environment into a container image to be able to share it with other users. Those users can then use Podman to launch the containerized application.

The way you do this with Podman is to commit the container to an OCI image.

First stop or pause the container to make sure nothing gets modified while you are committing it.

```
$ podman stop myapp
```

Now you can execute the `podman commit` command to take your application container, **myapp**, and commit it creating a new image named **myimage**.

```
$ podman commit myapp myimage
Getting image source signatures
Copying blob e39c3abf0df9 skipped: already exists
Copying blob 8f26704f753c skipped: already exists
Copying blob 83310c7c677c skipped: already exists
Copying blob 654b3bf1361e skipped: already exists
Copying blob 9e816183404c done
    Copying config e38084bb8a done
Writing manifest to image destination
Storing signatures
e38084bb8a76104a7cac22b919f67646119aff235bb1cfcba5478cc1fbf1c9eb
```

Now you can continue running the existing container `myapp` by calling `podman start`, or you can create a new container based on `myimage`.

```
$ podman run -d --name myapp1 -p 8080:8080 myimage
0052cb32c8e63b845ac5dfd5ba176b8204535c2c6cafa3277453424de601263f
```

**NOTE** Using the `podman commit` command to create an image, is not commonly used. The entire process of building container images can be scripted and automated using `podman build`. Explained in section 2.3.

#### FAVORITE PODMAN COMMIT OPTIONS

- `--pause` pauses a running container during the commit. Notice I stopped the container before doing the commit. I could have simply paused it. The `podman pause` and `podman unpause` commands allow you to pause and unpause containers directly.
- `--change` option allows you to commit instructions on how to use the image. The instructions are `CMD`, `ENTRYPOINT`, `ENV`, `EXPOSE`, `LABEL`, `ONBUILD`, `STOPSIGNAL`, `USER`, `VOLUME`, `WORKDIR`. These instructions match up with the directives in the Containerfile or Dockerfile.

Use the `man podman-commit` command for information about all options.

**NOTE** You have examined a few of the Podman container commands. There are many more. Use the `podman-container(1)` man pages to explore all of them as well as a full description of commands specified in this section.

```
$ man podman container
```

Now that you have committed your container to an image, it is time to show how Podman can work with images.

**Table 2.1 Podman container commands.**

Command	Man Page	Description
<code>attach</code>	<code>podman-container-attach(1)</code>	Attach to a running container
<code>checkpoint</code> <code>int</code>	<code>podman-container-checkpoint(1)</code>	Checkpoint a container
<code>cleanup</code>	<code>podman-container-cleanup(1)</code>	Cleanup network and mount points of a container
<code>commit</code>	<code>podman-container-commit(1)</code>	Commit a container into an image
<code>cp</code>	<code>podman-container-cp(1)</code>	Copy files/folder into and out of containers
<code>create</code>	<code>podman-container-create(1)</code>	Create a new container
<code>diff</code>	<code>podman-container-diff(1)</code>	Inspect changes in a containers file system
<code>exec</code>	<code>podman-container-exec(1)</code>	Run a process in a container
<code>exists</code>	<code>podman-container-exists(1)</code>	Check if a container exists
<code>export</code>	<code>podman-container-export(1)</code>	Export container's filesystem as a tar archive
<code>init</code>	<code>podman-container-init(1)</code>	Init a container
<code>inspect</code>	<code>podman-container-inspect(1)</code>	Display detailed information on a container
<code>kill</code>	<code>podman-container-kill(1)</code>	Send a signal to containers in container
<code>List</code> <code>(ps)</code>	<code>podman-container-list(1)</code>	List all of the containers

logs	podman-container-logs(1)	Fetch logs for a container
mount	podman-container-mount(1)	Mount a container's root filesystem
pause	podman-container-pause(1)	Pause container
port	podman-container-port(1)	List port mappings for a container
prune	podman-container-prune(1)	Remove all non running containers
rename	podman-container-rename(1)	Rename an existing container
restart	podman-container-restart(1)	Restart a container
restore	podman-container-restore(1)	Restore a checkpointed container
rm	podman-container-rm(1)	Remove a container
run	podman-container-run(1)	Run a command in a new container
runlabel	podman-container-runlabel(1)	Execute the command described by an image label
start	podman-container-start(1)	Start a container
stats	podman-container-stats(1)	Display statistics for a container
stop	podman-container-stop(1)	Stop a container
top	podman-container-top(1)	Display running process in container
unmount	podman-container-unmount(1)	Unmount a container's root filesystem
unpause	podman-container-unpause(1)	Unpause all the containers in a pod
wait	podman-container-wait(1)	Wait for a container to exit

## 2.2 Working with container images

In the previous section you tried basic operations with containers, including inspecting and committing to a container image. In this section you will try working with container images, learn how they differ from containers, and how to share them through container registries.

### 2.2.1 Difference between a container and an image

One of the problems with computer programming is that the same names are constantly used for different purposes. In the “container world” there is no more overused term than “container”. Often container refers to the running processes launched by Podman. But container can also refer to container data as the non-running objects sitting in container storage. As you saw in the previous section `podman ps --all`, shows running and non-running containers.

Another example is the term namespace which is used in many different ways. I often get confused when people talk about namespaces within Kubernetes. Some people hear the term and think of “virtual clusters”, but when I hear it, I think of [Linux namespaces](#) used with the Pods and Containers.

Similarly, `image` can refer to a VM Image , a Container image, OCI Image or a Docker image stored at a container registry.

I think of containers as executing processes within an environment or something that is prepared to be run. In contrast, images are committed `containers`, which are prepared to be shared with others. Other users/systems can use these images to create new containers.

Container images are just committed containers. The OCI defines the format of an image. Podman uses the <https://github.com/containers/image> library for all of its interaction with images. Container images can be stored in different types of storage or transports, as container/image refers to them. These transports can be Container Registries, Docker Archives, OCI Archives, docker-daemon as well as containers/storage. I cover transports later in section 2.2.4.1.

In the context of Podman, I usually refer to images as the content stored locally in a container storage or in container registries like docker.io and quay.io. Podman uses the <https://github.com/containers/storage> library for handling locally stored images. Let's take a closer look at it.

The container/storage library provides the concept of a storage container. Basically storage containers are intermediate storage content that hasn't been committed yet. Think of this as files on disk and some JSON describing the content. Podman has its own datastore of data related to a Podman container. Podman needs to deal with multiple users of its containers at the same time. It relies on file system locking provided by containers/storage to make sure hundreds of Podman executables can reliably share the same datastore.

When you commit a container to storage, Podman copies the container storage to the image storage. Images are stored in a series of layers. Every commit creates a new layer.

I like to think of an image like a wedding cake. In our example above you used the ubi8/httpd-24 image which is two layers, the base layer is ubi8, and then the image provided added the httpd package and a few others to create the ubi8/httpd-24. Now when you commit your container in the previous section, Podman adds another layer on top of the ubi8/httpd-24 image called myimage.

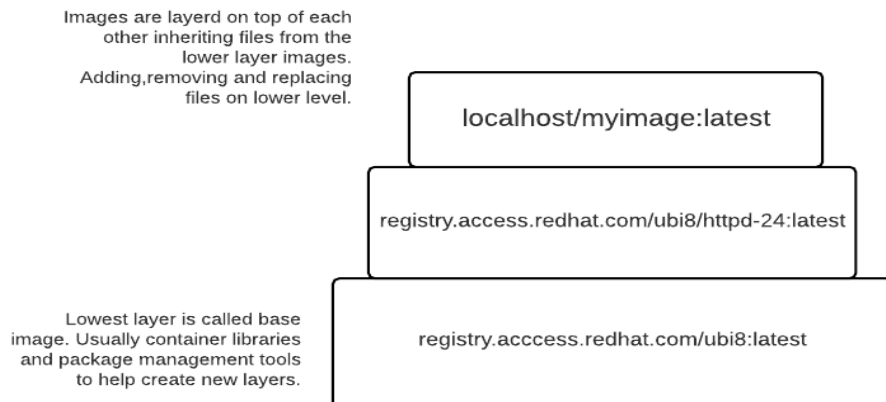


Figure 2.3 Wedding cake display showing the images making up our HelloWorld application.

One handy Podman command for showing the layers of an image is the `podman image tree` command.

```
$ podman image tree myimage
Image ID: 2c7e43d88038
Tags: [localhost/myimage:latest]
Size: 461.7MB
Image Layers
├─ ID: e39c3abf0df9 Size: 233.6MB
├─ ID: 42c81bd2b468 Size: 20.48kB Top Layer of: [registry.access.redhat.com/ubi8:latest]
├─ ID: 51a7beaa0b88 Size: 57.43MB
├─ ID: 519e681b5702 Size: 170.6MB Top Layer of: [registry.access.redhat.com/ubi8/httpd-24:latest]
└─ ID: bc3dcdefdac3 Size: 69.63kB Top Layer of: [localhost/myimage:latest
    localhost/myapp:latest]
```

You can see that the image **myimage** consists of five layers.

Another useful Podman command, `podman image diff`, allows you to see the actual files and directories that have been changed (C), added (A), or deleted (D) compared to another image or the lower layer:



```
$ podman image diff myimage ubi8/httpd-24
C /etc/group
C /etc/httpd/conf
C /etc/httpd/conf/httpd.conf
C /etc/httpd/conf.d
C /etc/httpd/conf.d/ssl.conf
C /etc/httpd/tls
C /etc
C /etc/httpd
A /etc/httpd/tls/localhost.crt
A /etc/httpd/tls/localhost.key
...
```

Images are just TAR diffs of software applied on lower level images. Container content is an uncommitted layer of software. Once a container is committed, you can create other containers on top of your image. You can also share the image with others so that they can create other containers on your image.

Now let's look at all of the images in your container storage.

## 2.2.2 Listing images

In the container section you were working with images and used command `podman images` to list the images in local storage.

```
$ podman images
REPOSITORY                                TAG      IMAGE ID      CREATED      SIZE
localhost/myimage                         latest   2c7e43d88038  46 hours ago  462 MB
registry.access.redhat.com/ubi8/httpd-24 latest   8594be0a0b57  5 weeks ago  462 MB
registry.access.redhat.com/ubi8           latest   ad42391b9b46  5 weeks ago  234 MB
```

Let's look at the different fields in the default output. Table 2.1 describes the different fields and data available with the `podman images` command.

**Table 2.2 Default fields listed by the `podman images` command**

Heading	Description
Repository	Complete name of the image
TAG	Version (tag) of the image. Image tagging is covered in section 2.2.6.
IMAGE ID	Unique identifier of the image. It is generated by Podman as a SHA256 hash of the image's JSON configuration object.
CREATED	Elapsed time since the image was created. Images are sorted by this field by default.
SIZE	The amount of storage being used by the image.

You will use the `podman images` command throughout this section.

**NOTE** Over time the amount of storage used by all of the images you pull grows. I have seen many instances where users have run out of disk space. You should monitor the size of images and containers removing them when you are no longer using them. Use the `man podman-system-prune` command for more information on cleaning up.

#### FAVORITE PODMAN IMAGES OPTIONS

- `--all` option is useful for listing all images. By default, podman images lists only the currently in use images. When an image is replaced by a newer image with the same tag, the previous image is tagged as `<none><none>`. These images are called dangling images. I cover dangling images in the 2.3.1 Podman build.

Use the `man podman-images` command for information about all options.

Similarly to containers, you often want to examine the configuration information associated with an image by inspecting it.

### 2.2.3 Inspecting images

In the previous sections I mentioned a couple of commands to examine images. I used the `podman image diff` to examine files/directories created or deleted between images. I also showed you a way to see the image hierarchy or wedding cake layers of images using the `podman image tree` command.

Sometimes you want to examine the configuration of an image. You use the `podman image inspect` command for this. The `podman inspect` command can also be used to inspect images, but the names can conflict with containers, so I like to use the specific image command.

```
$ podman image inspect myimage
[
  {
    "Id": "3b8fcf9081b4c4e6c16d763b8d02684df0737f3557a1e03ebfe4cc7cd6562135",
    "Digest":
"sha256:ff49aa6253ae47569d5aadbd73d70e7d0431bcf3a2f57b1b56feecdb531029a3",
    "RepoTags": [
      "localhost/myimage:latest"
    ],
    "RepoDigests": [
      "localhost/myimage@sha256:ff49aa6253ae47569d5aadbd73d70e7d0431bcf3a2f57b1b56feecdb531029a3"
    ],
    ...
  ]
]
```

As you can see this command outputs a large json array, 153 lines in the example above. It includes the data used for the OCI Image specification. When you create a container from an image, this information is used as one of the inputs to create the container.

When using the `inspect` command, it is often better to pipe its output to `less` or `grep` to find particular fields you are interested in. Or you can use the `--format` option.

If you want to to examine the default command to be executed from this image, execute:

```
$ podman image inspect --format '{{ .Config.Cmd }}' myimage
[/usr/bin/run-httpd]
```

Or if you wanted to see the stop signal:

```
$ podman image inspect --format '{{.Config.StopSignal}}' myimage
```

As you can see, nothing is output, meaning the developer of the application did not specify a STOPSIGNAL. When you build a container off of this image, the STOPSIGNAL is the default, 15, unless you override it via the command line.

#### FAVORITE PODMAN IMAGE INSPECT OPTION

- `--format` is useful as you see above to extract particular fields out of the json.

Use the `man podman-image-inspect` command for information about the command.

Once you are happy with a container and commit it to an image, the next step is to share it with others or maybe run it on another system. You need to push the image out to other types of container storage, usually a container registry.

## 2.2.4 Pushing images

In Podman, you use the `podman push` command to copy an image and all of its layers out of container storage and push it to other forms of container image storage like a container registry. Podman supports a few different types of container storage, which it calls transports.

#### CONTAINER TRANSPORTS

Podman uses the <https://github.com/containers/image> library for pulling and pushing images. I describe the containers/image project as a library for copying images between different types of container storage. One storage, as you have seen, is containers/storage.

When pushing an image, the [destination] is specified using `transport:ImageName` format. If no transport is specified, the ``docker`` (container registry) transport is used by default.

One of the novel things that Docker did, as I explained above, was invent the Container Registry concept. Basically a web server that contains container images. The `docker.io`, `quay.io`, `Artifactory`, web servers are all examples of container registries. The Docker engineering team defined a protocol for pulling and pushing these images from the container registries, which I refer to as the Container registry or “docker” transport.

When I want to run a container of an image, I can fully specify the image name including the transport like the command below.

```
$ podman run docker://registry.access.redhat.com/ubi8/httpd-24:latest echo hello
hello
```

For Podman, **`docker://`** transport is the default, it can be skipped for convenience:

```
$ podman run registry.access.redhat.com/ubi8/httpd-24:latest echo hello
hello
```

The `myimage` image that you created in the previous section was created locally, which means it doesn't have a registry associated with it. By default, locally created images have the **`localhost`** registry associated with them. You can see the images in the containers/storage using the `podman images` command, explained below.

```
$ podman images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
localhost/myimage	latest	2c7e43d88038	46 hours ago	462 MB
registry.access.redhat.com/ubi8/httpd-24	latest	8594be0a0b57	5 weeks ago	462 MB
registry.access.redhat.com/ubi8	latest	ad42391b9b46	5 weeks ago	234 MB

If the image has a remote registry associated with it - e.g., `registry.access.redhat.com/ubi8`, it can be pushed without specifying the [destination] field. On the contrary, since `localhost/myimage` does not have a registry associated with it, remote registry needs to be specified - e.g., `quay.io/rhatdan`.

```
$ podman push myimage quay.io/rhatdan/myimage
Getting image source signatures
Copying blob 164d51196137 done
Copying blob 8f26704f753c done
Copying blob 83310c7c677c done
Copying blob 654b3bf1361e [=====>-----] 82.0MiB / 162.7MiB
Copying blob e39c3abf0df9 [=====>-----] 100.0MiB / 222.8MiB
```

**NOTE** Before executing the `podman push` command, I logged into the `quay.io/rhatdan` account using `podman login`, which is covered in the next section.

After the push command is finished, the image becomes available for the pull for other users, given they have access to this container registry.

Table 2.2 describes the supported transports for different types of container's storage.

**Table 2.3 Podman supported transports**

Transport	Description
container registry (docker)	Default transport. References a container image stored in a remote container image registry. Container registry is a place for storing and sharing container images. For example, docker.io, quay.io.
oci	References a container image, compliant with the Open Container Image Layout Specification. The manifest and layer tarballs as individual files are located in the local directory.
dir	References a container image, compliant with the Docker image layout. It is very similar to the `oci` transport but stores the files using the legacy "docker" format. As a non-standardized format, primarily useful for debugging or noninvasive container inspection.
docker-archive	References a container image in Docker image layout which is packed into a TAR archive.
oci-archive	References an image compliant with the Open Container Image Layout Specification which is packed into a TAR archive. It is very similar to the `docker-archive` transport, but stores an image in OCI Format.
docker-daemon	References an image stored in the Docker daemon's internal storage. Since the Docker daemon requires root privileges, Podman has to be run by root user.
<b>container-storage</b>	References an image located in a local container storage. It is not a transport, but more of a mechanism for storing images. It can be used to convert other transports into container-storage. Podman defaults to using container-storage for local images.

You want to push your image to a container registry, but if you try to push it, the container registry rejects your push, since you have not provided login authorization information. You need to execute `podman login`, to create the authorization.

### 2.2.5 podman login (Logging into a container registry)

In the previous section, I pushed the image to my container registry by executing:

```
$ podman push myimage quay.io/rhatdan/myimage
```

But I left out a key step - logging into a container registry using correct credentials. This is a necessary step for pushing a container image. It is also required for pulling a container image from a private registry.

For this section to follow along, you need to set up an account at a container registry. There are a lot of container registries available. The <https://quay.io> and <https://docker.io> registries both provide free accounts and storage. Your company might have a private registry where you can also get an account.

For the examples, I continue to use my account rhatdan at quay.io. Login to get your credentials.

```
$ podman login quay.io
Username: rhatdan
Password:
Login Succeeded!
```

Notice the Podman command prompts me for my Username and Password at the registry. The podman login command has options to pass the username/password information on the command line to avoid the prompt, allowing you to automate the login process.

To store authentication information for the user, podman login command creates an auth.json file. By default this is stored in the `/run/user/$UID/containers/auth.json` file.

```
cat /run/user/3267/containers/auth.json
{
  "auths": {
    "quay.io": {
      "auth": "OBSCURED-BASE64-PASSWORD"
    }
  }
}
```

The auth.json file contains your registry password in a base64 encoded string, there is no cryptography involved. Therefore the auth.json file needs to be protected. Podman defaults to storing the file in `/run` because it is a temporary file system and is destroyed when you log out or the system is rebooted. The `/run/user/$UID/containers` directory is not accessible by other users on the system.

It is possible to override the location by specifying option `--auth-file`. Alternatively, you can use the `REGISTRY_AUTH_FILE` environment variable to modify its location. If both are specified the `--auth-file` option is used. All container tools use this file to access the container registry.

It is possible to run the podman login command multiple times to login to multiple registries storing the login information in the same authorization file with a different stanza.

**NOTE** Podman supports other mechanisms to store the password information. These are called credential helpers.

After you are done using the registry, you can log out of a registry executing `podman logout`. This command deletes the cached credentials stored in the auth.json file.

```
$ podman logout quay.io
Removed login credentials for quay.io
```

#### FAVORITE PODMAN LOGIN AND LOGOUT OPTIONS

- `--username (-u)` tells Podman username to use when logging into the registry, otherwise the command prompts.
- `--authfile` tells Podman to store the authorization file in a different location. You can also use the `REGISTRY_AUTH_FILE` environment variable to change the location.
- `--all` allows you to logout of all of the registries.

Use the `man podman-login` and `man podman-logout` commands for information about all options.

Notice when you pushed the image to a container registry, you sort of renamed **myimage** to **quay.io/rhatdan/myimage**.

```
$ podman push myimage quay.io/rhatdan/myimage
```

It'd be nice to just have the local image named `quay.io/rhatdan/myimage`, in which case you could have just executed.

```
$ podman push quay.io/rhatdan/myimage
```

In the next section, you learn how to add names to images.

## 2.2.6 Tagging images

Earlier in this chapter I pointed out that locally created images get created with a `localhost` registry. Images get created with the `localhost` registry when you commit a container to an image or if you use `podman build` to build an image. Podman has a mechanism to add additional names to images, Podman calls these names tags. The command is `podman tag`.

Using the `podman images` command, check to local image name:

```
$ podman images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
localhost/myimage	latest	2c7e43d88038	46 hours ago	462 MB
registry.access.redhat.com/ubi8/httpd-24	latest	8594be0a0b57	5 weeks ago	462 MB
registry.access.redhat.com/ubi8	latest	ad42391b9b46	5 weeks ago	234 MB

You want the final image that you plan on shipping to be referred to as `quay.io/rhatdan/myimage`, you add that name with the following `podman tag` command:

```
$ podman tag myimage quay.io/rhatdan/myimage
```

Now run `podman images` again to examine the images, you see the new name `quay.io/rhatdan/myimage`. Notice that the `localhost/myimage` and `quay.io/rhatdan/myimage` have the same IMAGE ID, `2c7e43d88038`.

```
$ podman images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
localhost/myimage	latest	2c7e43d88038	46 hours ago	462 MB
quay.io/rhatdan/myimage	latest	2c7e43d88038	46 hours ago	462 MB
registry.access.redhat.com/ubi8/httpd-24	latest	8594be0a0b57	5 weeks ago	462 MB
registry.access.redhat.com/ubi8	latest	ad42391b9b46	5 weeks ago	234 MB

Since the images have the same image ID they are the same image with multiple names.

Now you can interact directly with quay.io/rhatdan/myimage. First you need to log back in to quay.io.

```
$ podman login --username rhatdan quay.io
Password:
Login Succeeded!
```

Now push without requiring the destination name.

```
$ podman push quay.io/rhatdan/myimage
Getting image source signatures
...
Storing signatures
```

Much simpler.

Let's tag previously used image with a version **1.0**:

```
$ podman tag quay.io/rhatdan/myimage quay.io/rhatdan/myimage:1.0
```

Once again examine the images, notice that myimage now has three different names/tags. All three have the same IMAGE ID, 2c7e43d88038.

```
$ podman images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
localhost/myimage	latest	2c7e43d88038	46 hours ago	462 MB
quay.io/rhatdan/myimage	1.0	2c7e43d88038	46 hours ago	462 MB
quay.io/rhatdan/myimage	latest	2c7e43d88038	46 hours ago	462 MB
registry.access.redhat.com/ubi8/httpd-24	latest	8594be0a0b57	5 weeks ago	462 MB
registry.access.redhat.com/ubi8	latest	ad42391b9b46	5 weeks ago	234 MB

Now you can push the 1.0 version of the myimage (application) to the registry.

```
$ podman push quay.io/rhatdan/myimage:1.0
Getting image source signatures
Copying blob 8f26704f753c skipped: already exists
Copying blob e39c3abf0df9 skipped: already exists
Copying blob 654b3bf1361e skipped: already exists
Copying blob 83310c7c677c skipped: already exists
Copying blob 164d51196137 [-----] 0.0b / 0.0b
Copying config 2c7e43d880 [-----] 0.0b / 4.0KiB
Writing manifest to image destination
Storing signatures
```

Users can pull either the latest image or the 1.0 version. Later when you build version 2.0 of your application, you can store both images at the registry. You can run both version 1.0 and 2.0 of your application on the host at the same time.

Use a web browser (Firefox, Chrome, Safari, Internet Explorer, Microsoft Edge ...) to look at the images at quay.io. You can see 1.0 and the latest image



```
$ web-browser quay.io/repository/rhatdan/myimage?tab=tags
```

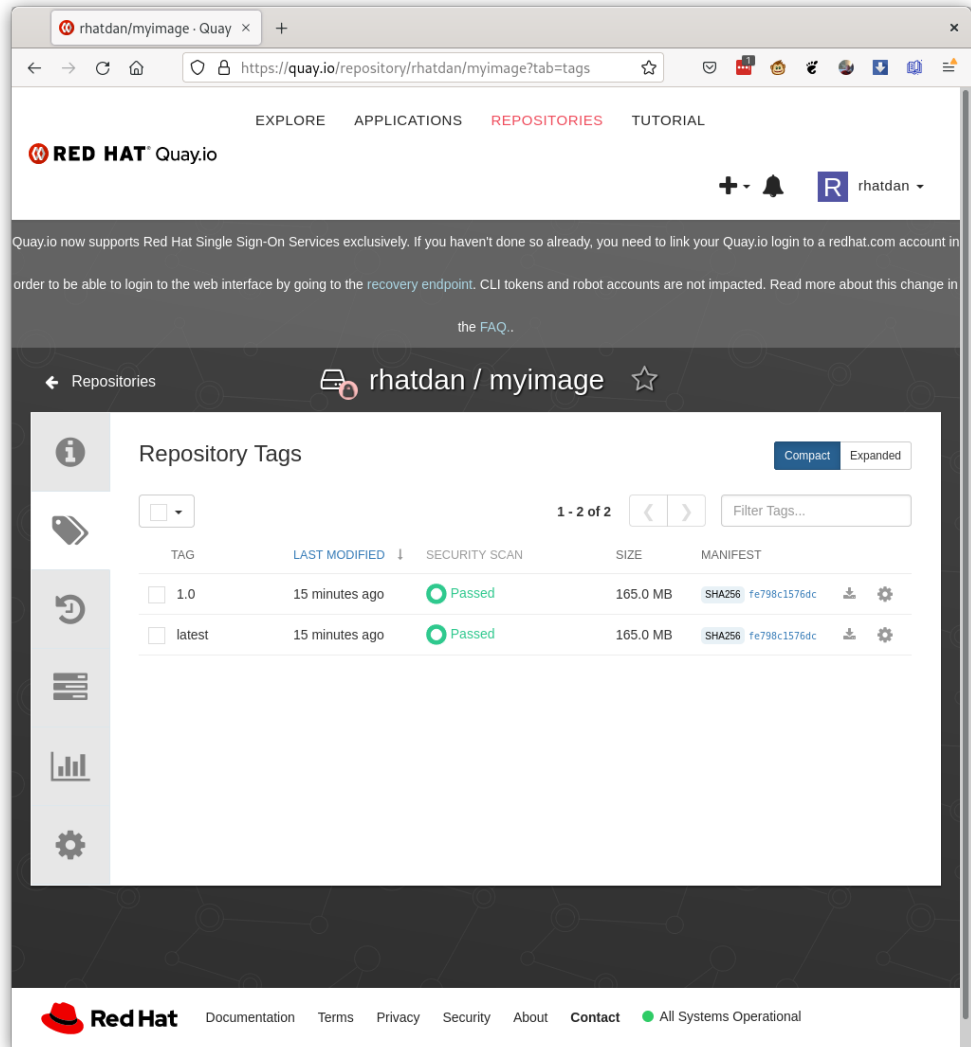


Figure 2.4 List of myimage tags on quay.io. <https://quay.io/repository/rhatdan/myimage?tab=tags>

**NOTE** Contrary to common sense, the tag *latest* does not refer to the most up-to-date image in the repository. It is just another tag with no magic involved. Even worse, because it is being used as a default tag for images pushed without tag, it could refer to any random version of an image. There could be newer images in the container registry than your local containers storage with this tag. Thus, it is always better to refer to the specific version of the image you want to use, rather than relying on the latest.

Now that you have pushed your image to a container registry, you may want to free up storage from your home directory by removing the images.

## 2.2.7 Removing images

Over time, images can take up a lot of disk space. Thus, it will be a good idea to remove no longer used images.

Let's list local images first:

```
$ podman images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
localhost/myimage	1.0	2c7e43d88038	46 hours ago	462 MB
quay.io/rhatdan/myimage	1.0	2c7e43d88038	46 hours ago	462 MB
quay.io/rhatdan/myimage	latest	2c7e43d88038	46 hours ago	462 MB
registry.access.redhat.com/ubi8/httpd-24	latest	8594be0a0b57	5 weeks ago	462 MB
registry.access.redhat.com/ubi8	latest	ad42391b9b46	5 weeks ago	234 MB

You use the `podman rmi` command to remove local images:

```
$ podman rmi localhost/myimage
Untagged: localhost/myimage:latest
```

Listing the local images again, you see that the command didn't actually remove the image but only the localhost tag to the image. Podman still has two references to the same image id, the actual content of the image has not been removed. None of the disk space was freed up.

```
$ podman images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
quay.io/rhatdan/myimage	1.0	2c7e43d88038	46 hours ago	462 MB
quay.io/rhatdan/myimage	latest	2c7e43d88038	46 hours ago	462 MB
registry.access.redhat.com/ubi8/httpd-24	latest	8594be0a0b57	5 weeks ago	462 MB
registry.access.redhat.com/ubi8	latest	ad42391b9b46	5 weeks ago	234 MB

You can remove the other tags using a short name (see 2.2.8.1). Podman uses the short name and finds the first name in local storage that matches the shortname without a registry and removes it, which is why I need to remove it twice, to get rid of both images. Tags other than `latest` need to be specified explicitly.

```
$ podman rmi myimage
Untagged: quay.io/rhatdan/myimage:latest
$ podman rmi myimage:1.0
Untagged: quay.io/rhatdan/myimage:1.0
Deleted: 2c7e43d88038669e8cbbdf324a9f9605d99697215a0d21c360fe8d8fa8471bab
```

It is only when the last tag is removed the actual disk space is reclaimed.

```
$ podman images
REPOSITORY                                TAG      IMAGE ID      CREATED      SIZE
registry.access.redhat.com/ubi8/httpd-24  latest   8594be0a0b57  5 weeks ago  462 MB
registry.access.redhat.com/ubi8           latest   ad42391b9b46  5 weeks ago  234 MB
```

Alternatively you can try removing the images by specifying the image id.

```
$ podman rmi 14119a10abf4
Error: unable to delete image
"2c7e43d88038669e8cdbdff324a9f9605d99697215a0d21c360fe8dfa8471bab" by ID with more
than one tag ([quay.io/rhatdan/myimage:1.0 quay.io/rhatdan/myimage:latest]): please
force removal
```

But that fails, because there are multiple tags for the same image. Adding the `--force` option removes the image and all of its tags:

```
$ podman rmi 14119a10abf4 --force
Untagged: quay.io/rhatdan/myimage:1.0
Untagged: quay.io/rhatdan/myimage:latest
Deleted: 2c7e43d88038669e8cdbdff324a9f9605d99697215a0d21c360fe8dfa8471bab
```

As your images size and quantity grow and more containers are created it becomes harder to figure out which images are no longer needed. Podman has another useful command: `podman image prune` for removing all “dangling” images. Dangling images are images that no longer have a tag associated with them or a container using them. The `prune` command also has the `--all` option which removes all images that are currently not in use by any containers including dangling images.

```
$ podman image prune -a
WARNING! This command removes all images without at least one container associated with
them.
Are you sure you want to continue? [y/N] y
6d633c2626113fb4e5aa75babb2af39268948497893f7bb5b4c2043d7a986ba0
B9097177b416944cabdcfcab0e74a319223ad1acaed38ac57a262b2421732355
```

**NOTE** Having no containers running, `podman image prune` command removes all of the local images. This frees up all of the disk space in the home directory. You can use the `podman system df` command to show all of the storage in your home directory used by podman.

```
$ podman images
REPOSITORY                                TAG      IMAGE ID      CREATED      SIZE
```

#### FAVORITE PODMAN IMAGE PRUNE OPTIONS

- `--all` tells Podman to remove all images, freeing up all storage. Note images which have containers running on them are not removed.
- `--force` tells Podman to stop and remove any containers that are running on them and remove any images that are dependent on the image you are attempting to remove

Use the `man podman-image-prune` command for information about all options.

Images pushed to the registry could also be pulled for various reasons, including but not limited to sharing your applications with others, testing other versions, getting back removed local versions, working on a new version of an image, etc.

## 2.2.8 Pulling images

Although you previously removed all local images, you have to start from scratch and you can use previously pushed at `quay.io/rhatdan/myimage`.

Podman has the `podman pull` command to pull images from container registries (transports) into local container storage:

```
$ podman pull quay.io/rhatdan/myimage
Trying to pull quay.io/rhatdan/myimage:latest...
Getting image source signatures
Copying blob dfd8c625d022 done
Copying blob e21480a19686 done
Copying blob 68e8857e6dcb done
Copying blob 3f412c5136dd done
Copying blob fbfcc23454c6 done
Copying config 2c7e43d880 done
Writing manifest to image destination
Storing signatures
2c7e43d880382561ebae3fa06c7a1442d0da2912786d09ea9baaef87f73c29ae
```

Does the output look familiar? You probably remember similar output of the `podman run` command from the section 2.1.2:

```
$ podman run -d -p 8080:8080 --name myapp registry.access.redhat.com/ubi8/httpd-24
Trying to pull registry.access.redhat.com/ubi8/httpd-24:latest...
Getting image source signatures
Checking if image destination supports signatures
Copying blob 296e14ee2414 skipped: already exists
Copying blob 356f18f3a935 skipped: already exists
Copying blob 359fed170a21 done
Copying blob 226cafc3a0c6 done
Writing manifest to image destination
Storing signatures
37a1d2e31dbf4fa311a5ca6453f53106eaae2d8b9b9da264015cc3f8864fac22
```

Many Podman commands implicitly execute the `podman pull` command if the required image is not present locally.

So, executing `podman images` shows the image back in container storage, ready to be used for containers:

```
$ podman images
REPOSITORY          TAG       IMAGE ID       CREATED        SIZE
quay.io/rhatdan/myimage latest    2c7e43d88038  2 days ago    462 MB
```

Up until now you have been typing the image with the full names as `registry.access.redhat.com/ubi8/httpd-24` or `quay.io/rhatdan/myimage`, but if you are like me and not a great typist, this can be a pain. What you need is a way to refer to the images via short names.

### SHORT NAMES AND CONTAINER REGISTRIES

When Docker first hit the scene they defined an image reference as a combination of the container registry where the image was stored, the repository, image name and a tag or version of the image.

In our examples we have been using `quay.io/rhatdan/myimage`. In table 2.4 you can see this image name breakdown, note that the *latest* tag was used implicitly as the image version wasn't specified.

**Table 2.4 Container image name table**

Registry	Repository	Name	Tag
quay.io	rhatdan	myimage	latest

The Docker command line has internally set `docker.io` registry as the only registry, thus making every short image name refer to images at `docker.io`. There is also a special repository **library**, which is used for certified images.

So, rather than typing

```
# docker pull docker.io/library/alpine:latest
```

You can just execute

```
# docker pull alpine
```

Conversely, if you want to pull an image from a different registry, you need to specify the full name of the image:

```
# docker pull registry.access.redhat.com/ubi8/httpd-24:latest
```

Table 2.4 shows the difference between the image name used in a short name versus the fully specified image name.

**Table 2.5 Short name to container image name table**

Registry	Repo	Name	Tag
		<b>alpine</b>	
docker.io	library	<b>alpine</b>	latest

Since I am lazy and hate to type extra characters, I almost always use short names.

With Podman, the developers did not want to hard code one registry, `docker.io`, into the tool. Podman allows distributions, companies and you to control which registries to use and to be able to configure multiple registries. At the same time, Podman provides support for the easier to use short names.

Podman usually comes with multiple registries defined, controlled by the distribution that packaged Podman.

You can use the `podman info` command to see what registries are defined for your Podman installation:

```
$ podman info
...
registries:
  search:
    - registry.fedoraproject.org
    - registry.access.redhat.com
    - docker.io
    - quay.io
```

The list of registries can be modified in the `registries.conf` file, which is described in chapter 5, section 6.2.1.

Let's discuss security side of things using these commands:

```
$ podman pull rhatdan/myimage
$ podman pull quay.io/rhatdan/myimage
```

From a security perspective, it is always better to specify the full image name when pulling it from a registry. That way Podman guarantees that it pulls from the specified registry. Imagine you are attempting to pull `rhatdan/myimage`. Using the search order above, there is a chance that someone could set up an account on `docker.io/rhatdan`, and trick you into mistakenly pulling `docker.io/rhatdan/myimage`.

To help protect against this, on the first pull of an image Podman prompts you to select an exact image from the list of found images in configured registries.

```
$ podman create -p 8080:8080 ubi8/httpd-24
? Please select an image:
  registry.fedoraproject.org/ubi8/httpd-24:latest
  ▶ registry.access.redhat.com/ubi8/httpd-24:latest
  docker.io/ubi8/httpd-24:latest
  quay.io/ubi8/httpd-24:latest
```

Once you have selected and pulled an image successfully, Podman records the shortname mapping. In the future when you run a container with this short name, Podman uses the short name mapping to pick the correct registry and does not prompt.

Linux distributions also ship mappings of the most commonly used short names as they want you to pull from their supported registries. You can find these short name configuration files in the `/etc/containers/registries.conf.d` directory on the Linux host. Companies can also drop short name alias files in this directory.

```
$ cat /etc/containers/registries.conf.d/000-shortnames.conf
[aliases]
# centos
"centos" = "quay.io/centos/centos"
# containers
"skopeo" = "quay.io/skopeo/stable"
"buildah" = "quay.io/buildah/stable"
"podman" = "quay.io/podman/stable"
...
```

### FAVORITE PODMAN PULL OPTIONS

- `--arch` tells Podman to pull an image for a different architecture. For example on my x86\_64 machine I can pull an arm64 image. By default podman pull pulls images for the native architecture.
- `--quiet (-q)` tells Podman to not print out all of the progress information. It just prints the image id, when it completes.

Use the `man podman-pull` command for information about all options.

I have mentioned a few images up til now in this book, but there are thousands and thousands of images available. You need a mechanism to be able to search through these images for the perfect match.

## 2.2.9 Searching for images

You might not know the name of a particular image that you want to run or use as a base for your own image.

Podman provides a command `podman search`, which allows you to search container registries, for matching names:

```
$ podman search registry.access.redhat.com/httpd
INDEX      NAME                                     DESCRIPTION
redhat.com registry.access.redhat.com/rhsc1/httpd-24-rhel7      Apache HTTP 2.4
Server
redhat.com registry.access.redhat.com/ubi8/httpd-24          Platform for running
Apache httpd 2.4 or bui...
redhat.com registry.access.redhat.com/rhsc1/varnish-6-rhel7    Varnish available as
container is a base pla...
...
```

In this example we are searching for images that include the string *httpd* in their name on the repository `registry.access.redhat.com`.

### FAVORITE PODMAN SEARCH OPTIONS

- `--no-trunc` tells Podman to show the full description of the image.
- `--format` allows you to customize which fields are displayed by Podman.

Use the `man podman-search` command for information about all options.

Up until now you have seen several ways of managing and manipulating container images, including inspecting, pushing, pulling and searching for them. But you have only been able to look at the contents of an image by running it as a container. One way to simplify the process is to mount a container image.

## 2.2.10 Mounting images

Often you might want to examine the contents of a container image, and one way to do this is to launch a shell inside of a running container from the image. The problem with this is that the tools you use to examine the container image might not be available within the container. There is also a security risk that the application in the container is malicious, making use of this container undesirable.

To help with these problems, Podman provides the `podman image mount` command to mount an image's root filesystem in a read-only mode without creating a container from it. Mounted image becomes immediately available on the host system, allowing you to examine its contents.

Let's try mounting the image we pulled previously:

```
$ podman mount quay.io/rhatdan/myimage
Error: cannot run command "podman mount" in rootless mode, must execute `podman unshare`
first
```

The reason for this error is that rootless mode does not allow mounting images. You need to enter a user namespace and separate mount namespace. Chapter 5 explains how most rootless Podman commands enter the user namespace and mount namespace when they execute. For now, it is enough to know that the `podman unshare` command enters the user and mount namespace and will shut down when you execute the `exit` command of your shell..

**NOTE** The name *unshare* comes from the Linux system call `unshare`. (man 2 unshare). Linux also includes an `unshare` tool (man 1 unshare) which allows you to create namespaces by hand. Another low level tool called `nsenter`, namespace enter (man 1 nsenter), allows you to join processes to different namespaces. Podman `unshare` uses the same Kernel features. It simplifies the process of creating and configuring namespaces and inserting processes into the namespaces.

The `podman unshare` command leaves you at a `#` prompt, where you can actually mount an image:

```
$ podman unshare
#
Mount the image and save the location of the mounted file system in an environment
variable:
# mnt=$(podman image mount quay.io/rhatdan/myimage)
```

Now you can actually examine the content of the image. Let's print the contents of a file on the terminal:.

```
# cat $mnt/var/www/html/index.html
<html>
<head>
</head>
<body>
<h1>Hello World</h1>
</body>
</html>
```

When you are done, unmount the image and exit the unshare session:

```
# podman image unmount quay.io/rhatdan/myimage
# exit
```

**NOTE** You have examined about a half of the `podman image` subcommands, arguably the most used ones. Refer to the Podman man pages for full explanation of these and other subcommands of the `podman image` command:



```
$ man podman-image
```

Now you have a better understanding of containers and images. The next important step is updating your image. The main reasons for this are the need to update your application and availability of new versions for the base image you use. You can build scripts to manually run the commands to build the image, but luckily Podman optimized the experience.

## 2.3 Building images

So far you have been working with images, which were already created and uploaded to a container registry. The process of creating a container image is called building.

When building container images, you manage not only your application, but also the image content used by this application. In the days prior to containers, you shipped applications as an RPM or DEB package, and then it was up to the distribution to make sure the other parts of the OS were kept up to date and secure. But in the container world the container image includes the application along with a subset of the OS. It is the developers responsibility to keep all of the image contents up to date and secure.

A co-worker of mine, [Scott McCarty](#) (@fatherlinux), has a saying that

“container images don’t age like wine, but more like cheese.

As the image gets older it gets stinky.”

What it means is that if the developer doesn’t keep up with the security updates ,the number of vulnerabilities in the image will grow at an alarming rate.

Luckily for developers, Podman has a special mechanism for helping you with image building for your applications. The `podman build` command uses the <https://github.com/quay.io/buildah> tool as a library to build container images, Buildah is covered in appendix A.

The `podman build` uses a special text document called Containerfile or Dockerfile to automate the building of container images. This document lists the commands in order you would call on the command line to assemble an image.

**NOTE** The concept of a Dockerfile and its syntax, was originally created for the Docker tool, developed by Docker, inc. Podman defaults to using Containerfile for the name, which uses the exact same syntax. Dockerfile is supported as well for legacy purposes. The `Docker build` command does not support Containerfile by default, but can use the Containerfile. You can specify the `-f` option. `# docker build -f Containerfile .`

### 2.3.1 Format of a Containerfile/Dockerfile

Containerfiles take many directives, I break these down into two categories, adding content to the container image and describing and documenting how to use the image.

#### ADDING CONTENT TO AN IMAGE

Recall back in section 1.1.2 that I described a container image as a directory on disk that looks like `root` on a linux system. This directory is called a `rootfs`. Several of the directives in a Container job is to add content to this `rootfs`. This `rootfs` eventually contains all of the content used to create your container image.

Every Containerfile must include a `FROM` line. The `FROM` line specifies the image that the new image is based off of, often called a base image. The `podman build` command supports a special image name ``scratch`` which means to start your image with no content. When Podman sees the `FROM scratch` directive, it just allocates space in containers/storage for an empty rootfs, then `COPY` can be used to populate the rootfs. More often the `FROM` directive uses an existing image. For example `FROM registry.access.redhat.com/ubi8`, causes Podman to pull the `ubi8` image from the `registry.access.redhat.com` container registry and copy it to container storage. This `podman build` pulls the image the same as the `podman pull` command which you learned about in section 2.2.8. When the image is pulled Podman uses container storage to mount the image on the rootfs directory, using a copy on write file system like overlays, where the other directives can start to add content. This image becomes the base layer of the rootfs.

The `COPY` directive is often used to copy files, directories or tar balls off of the local host into the newly created rootfs.

The `RUN` directive is one of the most commonly used Containerfile directives. `RUN` tells Podman to actually run a container on the image. Package management tools like `dnf/yum` and `apt-get` are run to install packages from distributions onto your new image. The `RUN` directive runs any command within the container image as a container. The `podman build` command runs the commands with the same security constraints as the `podman run` command.

As an example, imagine you want to add the `ps` command to a container image. Create a directive like the following. The `RUN` command executes a container which updates all of the packages from the base image, then installs the `procps-ng` package, which includes the `ps` command. Finally the containerized command executes `yum` to clean up after itself, so cruft is removed from the container image.

```
RUN yum -y update; yum -y install procps-ng; yum -y clean all
```

Adding content to the container image is only half of what you need to do when creating a container image, you also need to describe and document how the image will be used when other users download and run your image.

#### DOCUMENTING HOW TO USE THE IMAGE

Recall back in section 1.1.2 I also described the JSON file that included the image specification, this specification describes how the container image is to be run, the command, which user to run it with and other requirements of the image. The Containerfile also supports many directives which tells Podman how to run containers, these include:

The `ENTRYPOINT` and `CMD` directives instrument the image with the default command to be executed when users execute the image with Podman run. `CMD` is the actual command to run, `ENTRYPOINT` can cause the entire image to execute as a single command.

The `ENV` directive sets up the default environment variables to run when podman runs a container on the image.

The `EXPOSE` directive records the network ports for podman to expose in containers based on the image. If you execute `podman run --publish-all ...`, Podman looks inside of the image for the `EXPOSE` network ports and connects them to the host.

Table 2.5 explains the directives used in a Containerfile to add content to a container image.

**Table 2.6 Containerfile directives that update the image**

Directive Examples	Explanation
<b>FROM</b> quay.io/rhatdan/myimage	Sets the base image for subsequent instructions. Containerfiles must have <b>FROM</b> as its first instruction. The <b>FROM</b> may appear multiple times within a single Containerfile in order to create multiple build stages.
<b>ADD</b> start.sh /usr/bin/start.sh	Copies new files, directories or remote file URLs to the filesystem of the container at a specified path.
<b>COPY</b> start.sh /usr/bin/start.sh	Copies files to the filesystem of the container at a specified path.
<b>RUN</b> dnf -y update	Executes commands in a new layer on top of the current image and commits the results. The committed image is used for the next step in the Containerfile.
<b>VOLUME</b> /var/lib/mydata	Creates a mount point with the specified name and marks it as holding externally-mounted volumes from the native host or from other containers. For more on volumes, check Chapter 3.

Table 2.6 explains the directives used in a Containerfile to populate the OCI Runtime specifications with information that tells container engines like Podman information about the image how to run the image.

**Table 2.7 Containerfile directives that define the OCI Runtime Specification**

Directive Examples	Explanation
<b>CMD</b> /usr/bin/start.sh	Specifies the default command to run when launching a container off this image. If CMD is not specified the parent image's CMD is inherited. Note: RUN and CMD are very different. RUN runs the commands during the build process, while CMD is only used when a user launches the image without specifying a command.
<b>ENTRYPOINT</b> "/bin/sh -c"	Allows you to configure a container to run as an executable. The ENTRYPOINT instruction is not overwritten when arguments are passed to podman run. This allows arguments to be passed to the entrypoint, for instance, podman run <image> -d passes the -d argument to the ENTRYPOINT.
<b>ENV</b> foo="bar"	Adds an environment variable to be used during both the image build and container execution.
<b>EXPOSE</b> 8080	Announces the port that containerized applications will be exposing. Does not actually map or open any ports.
<b>LABEL</b> Description="Web browser which displays Hello World"	Adds metadata to an image.
<b>MAINTAINER</b> Daniel Walsh	Sets the Author field for the generated images.
<b>STOPSIGNAL</b> SIGTERM	Sets the default stop signal sent to the container to exit. The signal can be a valid unsigned number or a signal name in the format SIGNAME.
<b>USER</b> apache	Sets the user name (or UID) and group name (or GID) to use for any of RUN, CMD and ENTRYPOINT specified after it.
<b>ONBUILD</b>	Adds a trigger instruction to the image to be executed at a later time, when the image is used as the base for another build.
<b>WORKDIR</b> /var/www/html	Sets the working directory for any following RUN, CMD, ENTRYPOINT and COPY directives. Directory will be created if it doesn't exist.

There is much more information on Containerfiles in the containerfile(5) man page.

### COMMITTING THE IMAGE

When `podman build` completes processing of the Containerfile, it commits the image. This is using the same code as `podman commit` which you learned about in section 2.1.9. Basically Podman TARs up all of the differences between the new content in the rootfs and the base image, pulled down by the FROM directive. Podman also commits the JSON file and saves this as an image in container storage.

**TIP** Use the `--tag` option to name the new image that you are creating with `podman build`. This tells Podman to add the specified tag or name to the image in container storage in the same way as the `podman tag` command.

Now you want to take the steps used to build out containerized applications and automate them using a Containerfile and Podman build.

### 2.3.2 Automating the building of our application

First you create a directory to put your Containerfile in and any other content for the container image. The directory is called a context directory.

```
mkdir myapp
```

Next you create the `index.html` file you plan to use in the containerized application in the `myapp` directory.

```
$ cat > myapp/index.html << _EOF
<html>
<head>
</head>
<body>
<h1>Hello World</h1>
</body>
</html>
_EOF
```

Next you create a simple Containerfile to build your application in the `myapp` directory.

The first line of the Containerfile is the FROM directive to pull the `ubi8/httpd-24` image that you are treating as your base image. Then you add a COPY command to copy the `index.html` file into the image. The COPY directive tells Podman to copy the `index.html` file out of the context directory (`./myapp`) and copy it to the `/var/www/html/index.html` file within the image.

```
$ cat > myapp/Containerfile << _EOF
FROM ubi8/httpd-24
COPY index.html /var/www/html/index.html
_EOF
```

Finally, you use `podman build` to build my containerized application. You specify the `--tag` (`-t`) to name the image `quay.io/rhatdan/myimage`. You also need to specify the context directory `./myapp`.

```
$ podman build -t quay.io/rhatdan/myimage ./myapp
STEP 1/2: FROM ubi8/httpd-24
STEP 2/2: COPY index.html /var/www/html/index.html
COMMIT quay.io/rhatdan/myimage
--> f81b8ace4f1
Successfully tagged quay.io/rhatdan/myimage:latest
F81b8ace4f134d08cedb20a9156ae727444ae4d4ec1ceb3b12d3aff23d18128b
```

When the `podman build` command completes it commits the image and tags (-t) it with the `quay.io/rhatdan/myimage` name. Which is now ready to be pushed to the container registry using the `podman push` command.

Now you can set up a CI/CD system or even a simple cron job to regularly build and replace myapplication:

```
$ cat > myapp/automate.sh << _EOF
#!/bin/bash
podman build -t quay.io/rhatdan/myimage ./myapp
podman push quay.io/rhatdan/myimage
_EOF
$ chmod +x myapp/automate.sh
```

Add some test scripts as well to make sure your application works the way it was designed, before replacing the previous version.

Let's take a look at the images that were built:

```
$ podman images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
quay.io/rhatdan/myimage	latest	f81b8ace4f13	2 minutes ago	462 MB
<none>	<none>	<b>2c7e43d88038</b>	2 days ago	462 MB
registry.access.redhat.com/ubi8/httpd-24	latest	8594be0a0b57	5 weeks ago	462 MB

Notice the old version of `quay.io/rhatdan/myimage`, IMAGE ID `2c7e43d88038`, still exists in container storage but now has a REPOSITORY and TAG of `<none> <none>`. Images like these are called *dangling images*. Since I have created a new version of `quay.io/rhatdan/myimage` with the `podman build` command, the previous image loses that name. You can still use the Podman commands with the image id, or if the new image doesn't work, simply use `podman tag` to rename the old image back to `quay.io/rhatdan/myimage`. If the new image works correctly then you can remove the old image with `podman rmi`. These `<none><none>` images tend to build up over time wasting space, but you can periodically use the `podman image prune` command to remove them.

The `podman build` could really use a chapter or even a book to itself. People build images in thousands of different ways using the commands briefly described above.

#### FAVORITE PODMAN BUILD OPTION

- `--tag` specifies the image tag or name for the image. Remember that you can always add additional names after you create the image with the `podman tag` command you used in section 2.2.6.

Use the `man podman-build` command for information about all options.

**Table 2.8 Podman image commands.**

Command	Man Page	Description
build	podman-image-build(1)	Build an image using instructions from Containerfiles
diff	podman-image-diff(1)	Inspect changes in image's file system
exists	podman-image-exists(1)	Check if a image exists
history	podman-image-history(1)	Show history of a specified image
import	podman-image-import(1)	Import a tarball to create a filesystem image
inspect	podman-image-inspect(1)	Display the configuration of an image
list	podman-image-list(1)	List all of the images
load	podman-image-load(1)	Load image(s) from a tarball
mount	podman-image-mount(1)	Mount an image's root filesystem
prune	podman-image-prune(1)	Remove unused images
pull	podman-image-pull(1)	Pull an image from a registry
push	podman-image-push(1)	Push an image to a registry
rm	podman-image-rm(1)	Remove an image
save	podman-image-save(1)	Save image(s) to an archive
scp	podman-image-scp(1)	Securely copy images to other containers/storage
search	podman-image-search(1)	Search registry from an image
sign	podman-image-sign(1)	Sign an image
tag	podman-image-tag(1)	Add an additional name to a local image
tree	podman-image-tree(1)	Prints layer hierarchy of an image in a tree format
trust	podman-image-trust(1)	Manage container image trust policy
unmount	podman-image-unmount(1)	Unmount a image's root filesystem
untag	podman-image-untag(1)	Remove a name from a local image

## 2.4 Summary

- Podman's simple command line interface makes working with containers easy
- Podman run, stop, start, ps, inspect, rm, and commit are all commands for working with containers
- Podman pull, push, login, rmi are tools for working with images and sharing them via container registries
- Podman build is a great command for automating the build of container images
- Podman's command line is based on the Docker CLI and supports them exactly, allowing us to tell people to just alias docker=podman
- Podman has additional commands and options to support more advanced concepts like `podman image mount`

# 3

## Volumes

### This chapter covers

- Using volumes to isolate data from the containerized application.
- Sharing content from your host into containers via volumes
- Using volumes with the user namespace and SELinux
- Embedding volumes into container images
- Exploring different types of volumes and the `volumeman` volume commands

Up until now the containers you have been working with include all of their content within the container image. As I described in chapter 1, the only thing required to be shared with traditional containers is the Linux kernel. There are several reasons why you need to isolate application data from the application, including the following:

- Avoid embedding actual data for applications such as databases.
- Use the same container image to run multiple different environments.
- Reduce overhead and improve storage read/write performance, since volumes write directly to the file system, while containers use the overlay or fuse-overlayfs file system to mount their layers. Overlay is a layered file system, meaning that the kernel needs to copy the previous layer entirely in order to create a new layer. And fuse-overlay switches each read and write from kernel space to user space and back. All of this creates quite an overhead.
- Share content available via network storage.

**NOTE** Bind mounts remount parts of the file hierarchy in a different location on the file system. The files and directories in the bind mount are the same as the original. Bind mounts are explained in the `man mount` command. A bind mount allows the same content to be accessible in two places, without any additional overhead. It is important to understand that "bind" does not copy the data or create new data.



Supporting volumes also adds complexity, especially around security. A lot of the security features of containers prevent the container processes from gaining access to the file system outside of the container image. In this chapter you will discover the ways that Podman allows you to work around these obstacles.

### 3.1 Using volumes with containers

Let's go back to your containerized application. Up until now you have simply embedded the web application data into your container file system directly. Recall in chapter 2 (section 2.1.8), you used the `podman exec` command to modify the "Hello World" `index.html` data within the container:

```
$ podman exec -i myapp bash -c 'cat > /var/www/html/index.html' << _EOF
<html>
<head>
</head>
<body>
<h1>Hello World</h1>
</body>
</html>
_EOF
```

You have made the containerized image more flexible by allowing users to supply their own content for the web service or perhaps to update the web service on the fly. At the same time, while this method is possible, it is error-prone and not scalable. It is where volumes come in handy.

Podman allows you to mount host file system content into containers using the `podman run` command via the `--volume (-v)` option.

The `--volume HOST-DIR:CONTAINER-DIR` option tells Podman to bind mount `HOST-DIR` in the host to `CONTAINER-DIR` in the container. Podman supports other kinds of volumes as well, but in this section I am going to focus on bind mount volumes.

It is possible to mount both files or directories in a single option. Changes of the content on the host will be seen inside the container. Similarly, if container processes change the content inside the container, then the changes are seen on the host.

Let's look at an example. Create a directory, "html," in your home directory and then create a new `html/index.html` file in it:

```
$ mkdir html
$ cat > html/index.html << _EOF
<html>
<head>
</head>
<body>
<h1>Goodbye World</h1>
</body>
</html>
_EOF
```

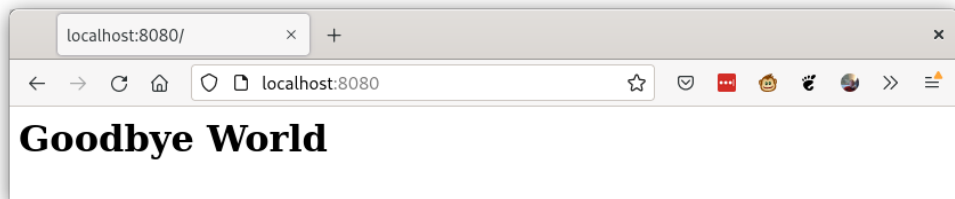
Now launch a container with the option `-v ./html:/var/www/html`:

```
$ podman run -d -v ./html:/var/www/html:ro,z -p 8080:8080 quay.io/rhatdan/myimage
94c21a3d8fda740857abc571469aaaa181f4db27a464ceb6743c4a37fb875772
```

Now, notice extra fields **:ro,z** in the `--volume` option. The **ro** option tells Podman to mount the volume in read-only mode. The read only mount means processes within the container cannot modify any content under `/var/www/html`, While processes on the host are still able to modify the content. Podman defaults all volume mounts to read-write mode. The **z** option tells Podman to re-label the content to a shared label for use by SELinux, more on this in section 3.1.2.2

Now that you have launched the container, open a web browser and navigate to `localhost:8080` to make sure the changes have taken place.

```
$ web-browser localhost:8080
```



**Figure 3.1** Web browser window connecting to the myimage Podman container with volume mounted Goodbye World HTML.

Now you can shut down and remove the container you just created. Removing the container does not affect the content at all. The following command removes the latest (**--latest**) container, yours. The **--force** option tells Podman to stop the container and then remove it.

```
$ podman rm --latest --force
```

Finally, remove the content with this command:

```
$ rm -rf html
```

**NOTE** The `--latest` option is not available on MAC and Windows. You must specify the container name or ID. Remote mode will be explained in chapter 9. Podman on MAC And Windows is explained in appendix E and F.

### 3.1.1 Named volumes

In the first volume example, you created a directory on disk and then mounted it into the container. Similarly, you can take any existing file or directory and mount it into a container, as long as you have read access to it.

Another mechanism for persisting Podman containers data is named volume. You can create one of these with the `podman volume create` command. In the following example you create a volume named `webdata`:

```
$ podman volume create webdata
webdata
```

Podman defaults to create local named volumes, with storage allocated in the container storage directories. You can inspect the volume and look for its mount point using the following command:

```
$ podman volume inspect webdata
[
  {
    "Name": "webdata",
    "Driver": "local",
    "Mountpoint": "/home/dwalsh/.local/share/containers/storage/volumes/webdata/_data",
    "CreatedAt": "2021-10-11T14:10:48.741367132-04:00",
    "Labels": {},
    "Scope": "local",
    "Options": {}
  }
]
```

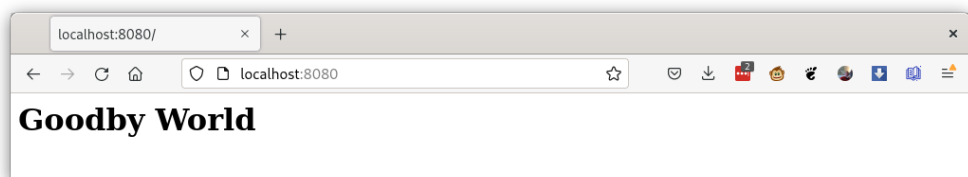
Podman actually creates a directory in your local container storage, **/home/dwalsh/.local/share/containers/storage/volumes/webdata/\_data**, to store the content of the volume. You can create content from the host in this directory.

```
$ cat > /home/dwalsh/.local/share/containers/storage/volumes/webdata/_data/index.html <<
_EOL
<html>
<head>
</head>
<body>
<h1>Goodbye World</h1>
</body>
</html>
_EOL
```

Now you can run the myimage application using this volume.

```
$ podman run -d -v webdata:/var/www/html:ro,z -p 8080:8080 quay.io/rhatdan/myimage
0c8eb612831f8fe22438d73d801e5bb664ec3b1d524c5c10759ee0049061cb6b
```

Now refresh the web browser to ensure the file created in the host directory is displaying "Goodby World".



**Figure 3.2** Web browser window connecting to the myimage Podman container with the named volume mounted.

Named volumes can be used for more than one container at a time. And they will stay around even after the container is removed.

If you are done with the named volume and container, you can first stop the container without waiting for the processes to finish:

```
$ podman stop -t 0 0c8eb61283
```

Then remove the volume with the `podman volume rm` command. Note the **--force** option which tells Podman to remove the volume and all containers that rely on the volume:

```
$ podman volume rm --force webdata
```

Now you can make sure the volume is gone by executing the volume list command.

```
$ podman volume list
```

If a named volume doesn't exist prior to executing the `podman run` command, it will be created automatically. In the following example you specify `webdata1` for the name of the named volume, then list the volumes:

```
$ podman run -d -v webdata1:/var/www/html:ro,z -p 8080:8080 quay.io/rhatdan/myimage
58ccaf37958496322e34cd933cd4dd5a61ab06c5ba678beb28fdc29cfb81f407
$ podman volume list
DRIVER    VOLUME NAME
local     webdata1
```

Of course, this volume is empty. Remove the `webdata1` volume and container:

```
$ podman volume rm --force webdata1
```

Podman also supports other types of volumes. It uses the concept of volume plugins so third-parties can provide volumes; check out the `podman-volume-create` man pages for more information.

Podman has other interesting volume features. The `podman volume export` command exports all of the content of a volume into an external tar archive. This archive can be copied to other machines used to recreate the volume on another machine with the `podman volume import` command.

Now that you understand the handling of volumes, it is time to dig deeper into volume options.

### 3.1.2 Volume mount options

You have been using volume mount options throughout this chapter. The `ro` option tells Podman to mount the volume read-only and the lowercase `z` option tells Podman to re-label the content in a way that multiple containers can read and write it.

```
$ podman run -d -v ./html:/var/www/html:ro,z -p 8080:8080 quay.io/rhatdan/myimage
```

Podman supports some other interesting volume options.

### THE U VOLUME OPTION

Sometimes when you run a rootless container, you need a volume to be owned by the user of the container. Imagine if your application needed to allow the webserver to write to the volume. In your container, Apache Web Server process (httpd) runs by user `apache` (UID==60). The `html` directory in your home directory is owned by you, so it is owned by root inside the container. The kernel does not allow a process running as UID==60 inside the container to make changes to a directory owned by root. You need to set the ownership of the volume to UID==60.

In rootless containers the UIDs of the container are actually offset by the user namespace. My user namespace mapping looks like this:

```
$ podman unshare cat /proc/self/uid_map
    0           3267           1
    1          100000         65536
```

The UID==0 inside the container is actually my UID 3267, and UID 1==100000, UID 2==100001... UID60==100059, meaning I need to set the ownership of the `html` directory to 100059.

I can do this fairly simply. With the `podman unshare` command, as follows:

```
$ podman unshare chown 60:60 ./html
```

Now everything works. One problem with this is that I need to do some mental gymnastics in order to figure out which UID the container will run with.

Many container images exist with the default UID defined in them. The `mariadb` image is another example of this, it runs with the `mysql` user, UID=999.

```
$ podman run docker.io/mariadb grep mysql /etc/passwd
mysql:x:999:999:./home/mysql:/bin/sh
```

If you created a volume to be used for the database, you need to figure out what UID=999 mapped to within the user namespace. On my system this is UID=100998.

Podman supplies the `u` command option for just this exact situation. The `u` option tells Podman to recursively change ownership (`chown`) the source volume to match the default UID that the container executes with.

Try it out by first creating the directory for the database. Notice the directory in the home directory is owned by your user.

```
$ mkdir mariadb
$ ls -ld mariadb/
drwxrwxr-x. 1 dwalsh dwalsh 0 Oct 23 06:55 mariadb/
```

Now run the `mariadb` container with the `--user mysql` and bind mount the `./mariadb` directory to `/var/lib/mariadb` with the `:U` option. Notice that the directory is now owned by the `mysql` user:

```
$ podman run --user mysql -v ./mariadb:/var/lib/mariadb:U docker.io/mariadb ls -ld
/var/lib/mariadb
drwxrwxr-x. 1 mysql mysql 0 Oct 23 10:55 /var/lib/mariadb
```

If you look at the mariadb directory on the host again, you will see that it is now owned by UID 100998, Or whatever UID 999 maps to within your user namespace.

```
$ ls -ld mariadb/
drwxrwxr-x. 1 100998 100998 0 Oct 23 06:55 mariadb/
```

User namespace is not the only security mechanism you need to work around with rootless containers. SELinux, while great for container security, can cause some problems when working with volumes.

### THE SELINUX VOLUME OPTIONS

In my opinion SELinux is the best mechanism for protecting the file system from hostile container processes. Over the years, several container escapes have been thwarted by SELinux. SELinux is covered in detail in chapter 10, Section 10.8.

As I explained previously, volumes leak files from the OS into the container, and from an SELinux point of view these files and directories must be labeled correctly, or the kernel blocks access.

The lowercase `z` command option that you have been using in this chapter tells Podman to recursively re-label all content in the source directory with a label that can be read and written by all containers from an SELinux point of view.

If the volume is not going to be used by more than one container, re-labeling with the lowercase `z` option isn't what you want. If a different hostile container escapes confinement, it might be able to get access to this data and read-write it. Podman provides an uppercase `Z` option that tells Podman to recursively relabel the content in such a way that only the processes within the container are able to read-write the content.

In both previous cases you re-labeled the content of the directory. Relabeling works great as long as the directory is specified for use by containers. Sometimes you might want to use a container to examine content in a system-specific directory. Say you wanted to run a container that examined all of the logs in `/var/log` or examined all of your home directories `/home/dwalsh`.

**NOTE** Using this option on a home directory can have disastrous effects on the system, because it recursively re-labels all content in the in directory as if the data was private to a container. Other confined domains would be prevented from using the mislabeled data.

For these cases, you need to disable SELinux enforcement for container separation to allow the containers to use the volume. Podman provides the command option, `--security-opt label=disable`, to disable SELinux support for a single container, basically running the container with an "unconfined" label from an SELinux perspective.

```
$ podman run --security-opt label=disable -v /home/dwalsh:/home/dwalsh -p 8080:8080
quay.io/rhatdan/myimage
```

Table 3.1 lists and describes all of the mount options available in Podman.

**Table 3.1 Volume mount options**

Volume option	Description
<code>nodev</code>	Prevent container processes from using character or block devices on the volume.
<code>noexec</code>	Prevent container processes from direct execution of any binaries on the volume.
<code>nosuid</code>	Prevent SUID applications from changing their privilege on the volume.
<code>O</code>	Mount the directory from the host as a temporary storage using the overlay file system. Modifications to the mount point are destroyed when the container finishes executing. This option is useful for sharing the package cache from the host into the container to allow speeding up builds.
<code>[r]shared </code> <code>[r]slave </code> <code>[r]private </code> <code>[r]unbindable</code>	Specify mount propagation mode. Mount propagation controls how changes to mounts are propagated across mount boundaries.  private (default) - any mounts done inside container will not be visible on host and vice versa shared - mounts done under that volume inside container will be visible on host and vice versa. slave - mounts done on host under that volume will be visible inside container but not the other way around.  unbindable -is an unbindable version of private mode. Prefix r stands for recursive. Meaning that any mounts underneath the mount point will also be treated the same way.
<code>rw ro</code>	Mount a volume in read-only (ro) or read-write (rw) mode. By default, read-write is implied.
<code>U</code>	Use the correct host UID and GID based on the UID and GID within the container. Use with caution because this will modify the host filesystem.
<code>z Z</code>	Relabel file objects on the shared volumes. Choose the z option to label volume content as shared among multiple containers. Choose the Z option to label content as unshared and private.

For more information, see man pages for `mount` and `mount_namespaces(7)`.

Most of the time the simple `--volume` option is powerful enough for mounting volumes into containers. Over time the requests for new mount options grew too complex, so a new option called `--mount` was added.

### 3.1.3 podman run --mount command option

The `podman run --mount` option is a much closer option to the underlying Linux mount command. It allows you to specify all of the mount options that the mount command understands; Podman passes them down directly to the kernel.

The only mount types currently supported are bind, volume, image, tmpfs, and devpts. (For more information, see the `podman-mount(1)` man page for more information.)

Volumes and Mounts are excellent ways to keep data separate from the container image. In most cases the Container image should be treated as read-only, and any data that needs to be written or is not specific to the application should be stored outside of the container image via volumes. In a lot of cases you will get much better performance keeping your data separate, because read and writes will not have the overhead of the copy-on-write file system. These mounts also make it easier to use the same container images with different data.

**Table 3.2 Podman volume commands.**

Command	Man Page	Description
<code>create</code>	<code>podman-volume-create(1)</code>	Create a new volume
<code>exists</code>	<code>podman-volume-exists(1)</code>	Check if a volume exists
<code>export</code>	<code>podman-volume-export(1)</code>	Export the contents of a volume into a tar ball
<code>import</code>	<code>podman-volume-import(1)</code>	Untar a tarball into a volume
<code>inspect</code>	<code>podman-volume-inspect(1)</code>	Display detailed information on a volume
<code>list</code>	<code>podman-volume-list(1)</code>	List all of the volumes
<code>prune</code>	<code>podman-volume-prune(1)</code>	Remove all unused volumes
<code>rm</code>	<code>podman-volume-rm(1)</code>	Remove one or more volumes

## 3.2 Summary

- Volumes are useful for separating the data used by a container from the application inside an image
- Volumes mount parts of the file system into a container's environment, which means security concerns like SELinux and user namespace need to be modified to allow access



# 4

## Pods

### This chapter covers

- An introduction to pods
- Managing multiple containers within a Pod
- Using Volumes with Pods

Podman stands for Pod Manager. Pod is a concept popularized by the Kubernetes project. It is a group of one or more containers working together for a common purpose and sharing the same namespaces and cgroups (resource constraints). Additionally, Podman ensures that on SELinux machines all container processes within a pod share the same SELinux labels, this means that they can all work together from an SELinux point of view.

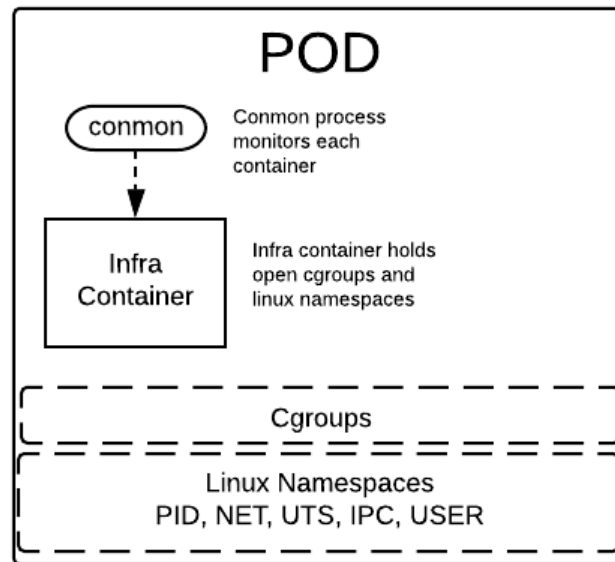
### 4.1 Running Pods

Podman pods, just like Kubernetes pods, always include a container called the *infra container*, sometimes called the pause container (not to be confused with the rootless pause container, mentioned in section 5.2). The infra container only holds open the namespaces and cgroups from the kernel, allowing containers to come and go within the pod. When Podman adds a container to a pod, it adds the container process to the cgroups and namespaces. Notice that the infra container has a container monitor process `conmon` monitoring it. Every container within a pod has its own `conmon`.

Conmon is a lightweight C program that monitors the container until it exits, allowing the Podman executable to exit and reconnect to the container.

Conmon does the following when monitoring the container:

1. Common executes the OCI runtime, handing it the path to the OCI spec file as well as pointing to the container layer mount point in containers/storage. The mount point is called the rootfs.
2. Common monitors the container until it exits and reports its exit code back.
3. Common handles when the user attaches to the container, providing a socket to stream the container's STDOUT and STDERR.
4. The STDOUT and STDERR are also logged to a file for podman logs.



**Figure 4.1** Podman pod launches common with the infra container, which will hold cgroups and linux namespaces.

**NOTE** The infra container (pause container) is similar to the rootless pause container while its only purpose is to hold open the namespaces and cgroups while containers come and go. But each Pod will have a different infra container.

Podman pods also support *init containers*. These containers run before the primary containers in the pods are executed. An example of an init container might be a database initialization on a volume; then the primary container can use the database. Podman supports the following two classes of init containers:

- Once: Only runs the first time the pod is created
- Always: Runs every time the pod is started.

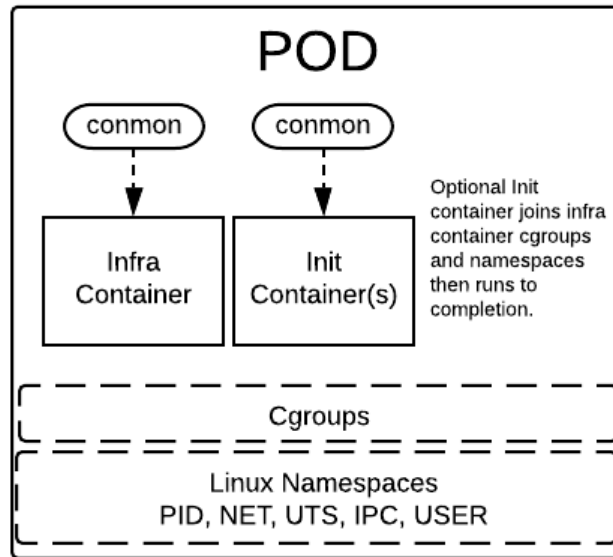
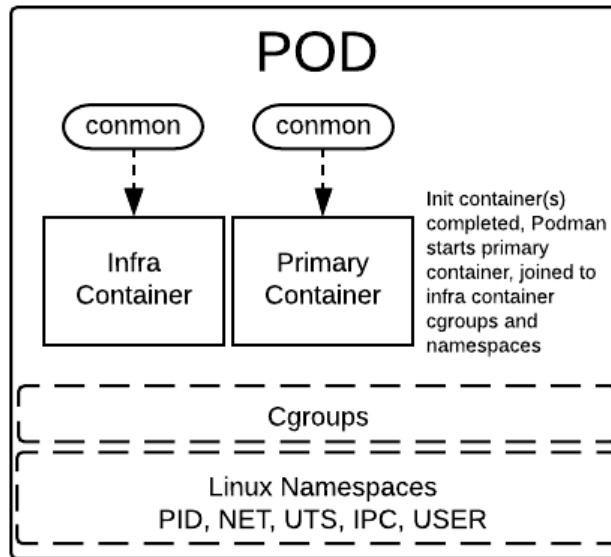


Figure 4.2 Podman next launches any init containers with common. The init containers examine the infra container and join its cgroups and namespaces.

The Primary container runs the application.



**Figure 4.3** Podman waits until the init containers complete before launching the primary containers with their common into the pod.

Pods also support additional containers, these are often called sidecar containers. Sidecar containers often monitor the primary container, or modify the environment where the primary container runs.

The Kubernetes documentation, <https://kubernetes.io/docs/concepts/workloads/pods/>, describes pods with sidecar containers as:

*A Pod can encapsulate an application composed of multiple co-located containers that are tightly coupled and need to share resources. These co-located containers form a single cohesive unit of service—for example, one container serving data stored in a shared volume to the public, while a separate sidecar container refreshes or updates those files. The Pod wraps these containers, storage resources, and an ephemeral network identity together as a single unit.*

If you want to dive deeper into sidecar containers there are good articles at this website <https://www.magali.com/blog/the-sidecar-pattern>.

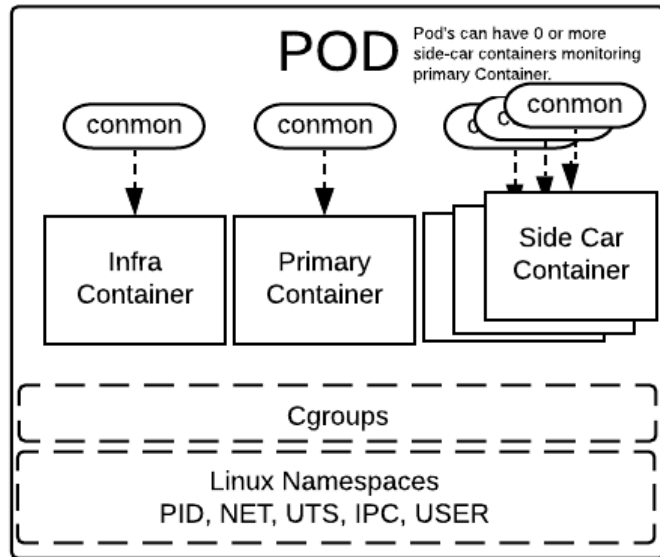


Figure 4.4 Podman can launch additional containers called sidecar containers.

**NOTE** While pods can support more than one sidecar container, I recommend that you only use one. There is a real temptation for people to abuse this capability especially in Kubernetes. But it can use up more resources and become unwieldy.

A big advantage of pods is that you can manage them as discrete units. Starting a pod starts all of the containers within it, and stopping the pod stops all of the containers.

#### 4.1.1 Creating a pod

In this section, you are going to create a pod where you have the myimage application as the primary container within the pod. You will also add a second container to the pod, a sidecar container which will update the web content that is used by your application, to show two containers working together within a pod.

You can create a pod named **mypod** using the `podman pod create` command, as demonstrated in the following command:

```
$ podman pod create -p 8080:8080 --name mypod --volume ./html:/var/www/html:z
790fefe97b280e5f67c526e3a421e9c9f958cf5a98f3709373ef1afd91965955
```

The podman pod create command has many of the same options as the podman container create command. When you create a container within a Pod, the container inherits these options as their default.

Notice that similar to the previous examples, you are binding the pod to port **-p 8080:8080**.

```
$ podman pod create -p 8080:8080 --name mypod --volume ./html:/var/www/html:z
```

Because the containers within the pod share the same network namespace, this port binding is shared by all of the containers. The kernel allows only one process to listen on port 8080. Lastly, notice that the directory `./html` was volume mounted, **--volume ./html:/var/www/html:z**, into the pod.

```
$ podman pod create -p 8080:8080 --name mypod --volume ./html:/var/www/html:z
```

The `:z` parameter causes Podman to relabel the content of the directory. Podman will automatically volume mount this directory into every container that joins the pod. Containers in Pods share the same SELinux label, which means they can share the same volumes.

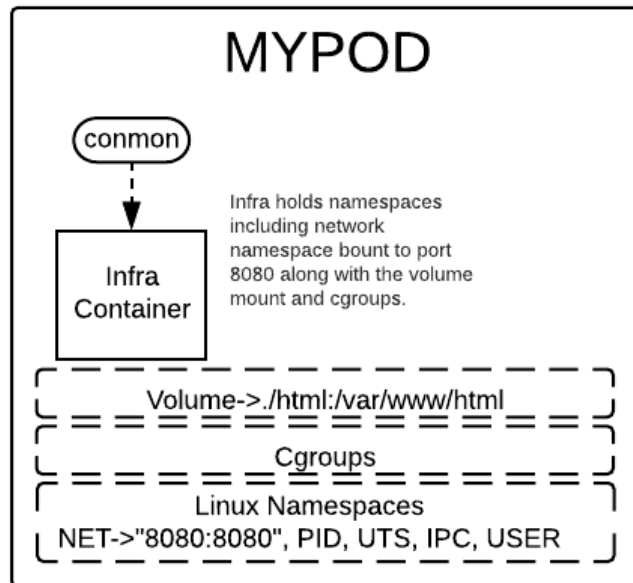
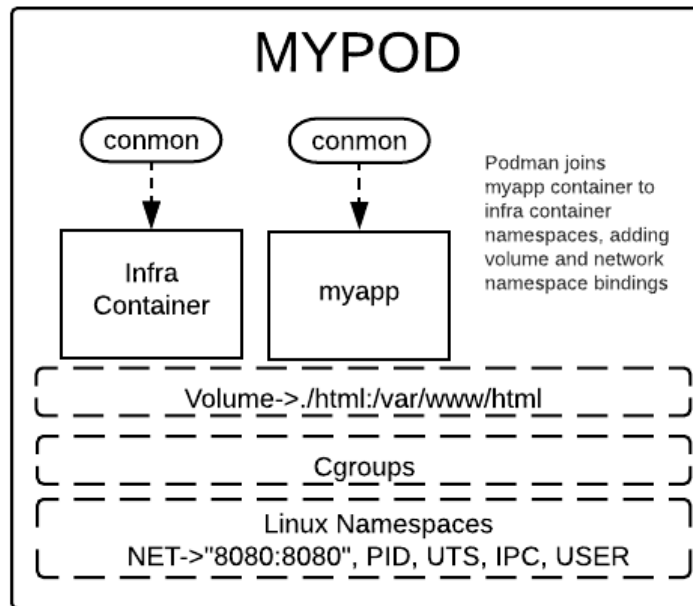


Figure 4.5 Podman creates network namespace and binds port 8080 within container to port 8080 on the host, using `slirp4netns`. Podman creates the infra container with the `/var/www/html` directory from the host into the container, and joins the `cgroups` and network namespace.

### 4.1.2 Adding a container to a pod

You can create a container within a pod using the podman create command. You add the quay.io/rhatdan/myimage container to the pod with the **--pod mypod** option.

```
$ podman create --pod mypod --name myapp quay.io/rhatdan/myimage
Cec045acb1c2be4a6e4e88e21275076fb1de5519a25fb5a55f192da70708a640
```



**Figure 4.6** because the pod does not have any init containers, the first container myapp is launched into the pod.

When you add the first container to the pod, Podman reads the information associated with the infra container and adds the volume mount to the myapp container and then joins it to the namespaces held by the infra container.

The next step you will add the sidecar container to the pod. The sidecar container will be updating the index.html file in the /var/www/html volume, adding a new time stamp every second.

Create a simple bash script to update the index.html used by the myapp container called html/time.sh. You can create it in the ./html directory so that it will be available to processes within the pod.

```
$ cat > html/time.sh << _EOL
#!/bin/sh
data() {
    echo "<html><head></head><body><h1>"; date;echo "Hello World</h1></body></html>"
    sleep 1
}
while true; do
    data > index.html
done
_EOL
```

Make sure that it is executable. You can do this on Linux with the `chmod` command.

```
$ chmod +x html/time.sh
```

Now create the second container named `time` (**--name time**), this time using a different image, **ubi8**. Containers within pods can use totally different images, even images from different distributions. Recall that container images only share the host kernel by default.

```
$ podman create --pod mypod --name time --workdir /var/www/html ubi8 ./time.sh
Resolved "ubi8" as an alias (/etc/containers/registries.conf.d/000-shortnames.conf)
Trying to pull registry.access.redhat.com/ubi8:latest...
...
1be0b2fae53029d518e75def71c0d6961b662d0e8b4a1082ede5589d1353af3
```

Remember the concept of short names was covered in chapter 2. You can type the long name, `registry.access.redhat.com/ubi8`, but that is a lot of typing. Luckily for us, the short name `ubi8` already had an alias map to its long name, meaning you do not need to select it from the list of registries. Podman shows you where it found the alias for the long name in the output.

```
$ podman create --pod mypod --name time --workdir /var/www/html ubi8 ./time.sh
Resolved "ubi8" as an alias (/etc/containers/registries.conf.d/000-shortnames.conf)
```

You also used the **--workdir** command option to set the default directory for the container to `/var/www/html`. When the container starts, the `./time.sh` it will run in the `workdir` and is actually `/var/www/html/time.sh`.

```
$ podman create --pod mypod --name time --workdir /var/www/html ubi8 ./time.sh
```

Because this container is going to be run within the `mypod` pod, it will inherit the **-v ./html:/var/www/html** option from the pod, meaning the `./html/time.sh` command in the host directory is available to every container within the pod.



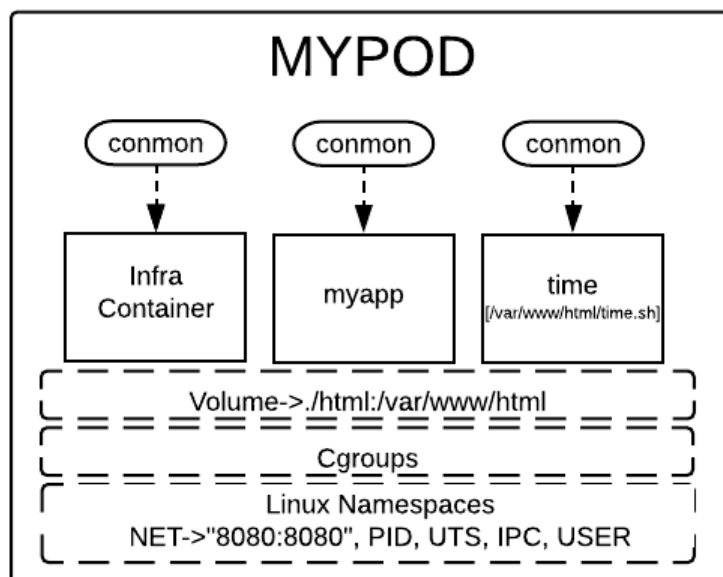


Figure 4.7 Finally Podman launches the sidecar container named time.

Podman examines the infra container and mounts the `/var/www/html` volume and joins the namespaces, when it launches the sidecar container.

Now it is time to start the pod, and see what happens.

### 4.1.3 Starting a pod

You can start the pod with the `podman pod start mypod` command.

```
$ podman pod start mypod
790fefe97b280e5f67c526e3a421e9c9f958cf5a98f3709373ef1afd91965955
```

Use the `podman ps` command to see which containers the pod started.

```
$ podman ps
CONTAINER ID  IMAGE                                COMMAND                                CREATED
STATUS       PORTS                                NAMES
b9536ea4a8ab  localhost/podman-pause:4.0.3-1648837314  8920b1ccd8b0-infra  14 minutes ago Up
5 seconds ago 0.0.0.0:8080->8080/tcp
a978e0005273  quay.io/rhatdan/myimage:latest         /usr/bin/run-http... 14 minutes ago
Up 5 seconds ago 0.0.0.0:8080->8080/tcp myapp
be86937986e9  registry.access.redhat.com/ubi8:latest  ./time.sh           13 minutes
ago Up 5 seconds ago 0.0.0.0:8080->8080/tcp time
```

Notice now that three containers have started. The infra container is based on the `k8s.gcr.io/pause` image, our application based on `quay.io/rhatdan/myimage:latest`, and the update container based on the `registry.access.redhat.com/ubi8:latest` image.

When the `ubi8` sidecar container starts it begins modifying the `index.html` via the `time.sh` script. Since the `myapp` container shares the volume mount `/var/www/html` it can see the changes in `/var/www/html/index.html` file.

Launch your favorite web browser and navigate to `http://localhost:8080` to verify the application is working.



**Figure 4.8** The web-browser communicates with `myapp` running in a pod.

A couple of seconds later hit the refresh button.



**Figure 4.9** The web-browser shows that the content in `myapp` has been changed by the second container running in the pod.

Notice the date changes, indicating that the sidecar container is running and updating the data used by the `myapp` web server running within the primary container.

#### **FAVORITE PODMAN POD START OPTIONS**

- `--all` tells Podman to start all pods.
- `--latest, -l` tells Podman to start the last pod created. (Not available on MAC and Windows)

Now that you demonstrated the application running within a pod, you might want to stop the application.

#### 4.1.4 Stopping a pod

Now that you see the application ran successfully, you can stop the pod with the *podman pod stop* command, as follows:

```
$ podman pod stop mypod
790fefe97b280e5f67c526e3a421e9c9f958cf5a98f3709373ef1afd91965955
```

Use the *podman ps* command to make sure that Podman stopped all of the containers within the pod.

```
$ podman ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES

##### FAVORITE PODMAN POD STOP OPTIONS

- `--all` tells Podman to stop all pods.
- `--latest, -l` tells Podman to stop the latest started pod.
- `--timeout, -t` tells Podman to set the timeout when attempting to stop the containers within a pod.

Now that you have created, run, and stopped the pod, let's start to examine it. First you can list all of the pods on your system.

#### 4.1.5 Listing pods

You can list Pods with the *podman pod list* command:

```
$ podman pod list
```

POD ID	NAME	STATUS	CREATED	INFRA ID	# OF CONTAINERS
790fefe97b28	mypod	Exited	22 minutes ago	b9536ea4a8ab	3

##### FAVORITE PODMAN POD LIST OPTIONS

- `--ctr*` tells Podman to list container information within pods.
- `--format` tells podman to change the output of pods.

Now that you are done with the demonstration time to cleanup the pods and containers.

#### 4.1.6 Removing pods

In a later chapter, I will show you how you can generate Kubernetes YAML files to allow you to launch your pod on other systems using Podman or within Kubernetes. But for now, you can remove a pod with the *podman pod rm* command.

Before you do this, list **--all** the containers on the system. Using the `--format` option to show only the ID, Image and Pod ID, you will see three containers that make up your pod.

```
$ podman ps --all --format "{{.ID}} {{.Image}} {{.Pod}}"
b9536ea4a8ab k8s.gcr.io/pause:3.5 790fefe97b28
a978e0005273 quay.io/rhatdan/myimage:latest 790fefe97b28
be86937986e9 registry.access.redhat.com/ubi8:latest 790fefe97b28
```

Now you can remove the pod with the following command:

```
$ podman pod rm mypod
790fefe97b280e5f67c526e3a421e9c9f958cf5a98f3709373ef1afd91965955
```

Make sure it is gone:.

```
$ podman pod ls
POD ID    NAME      STATUS    CREATED    INFRA ID # OF CONTAINERS
```

Good, it looks like your pod is gone. Verify that Podman removed all of the containers by running the following command to verify:

```
$ podman ps -a --format "{{.ID}} {{.Image}}"
```

The system is fully cleaned up.

#### **FAVORITE PODMAN POD RM OPTIONS**

- `--all` tells Podman to remove all the pods.
- `--force` tells Podman to first stop all running containers before attempting to remove them. Otherwise Podman will only remove non running pods.

**Table 4.1 Podman pod commands.**

Command	Man Page	Description
<code>create</code>	<code>podman-pod-create(1)</code>	Create a new pod
<code>exists</code>	<code>podman-pod-exists(1)</code>	Check if a pod exists
<code>inspect</code>	<code>podman-pod-inspect(1)</code>	Display detailed information on a pod
<code>kill</code>	<code>podman-pod-kill(1)</code>	Send a signal to containers in pod
<code>list</code>	<code>podman-pod-list(1)</code>	List all of the pods
<code>logs</code>	<code>podman-pod-logs(1)</code>	Fetch logs for pod with one or more containers
<code>pause</code>	<code>podman-pod-pause(1)</code>	Pause all the containers in a pod
<code>prune</code>	<code>podman-pod-prune(1)</code>	Remove all stopped pods and their containers
<code>restart</code>	<code>podman-pod-restart(1)</code>	Restart a pod
<code>rm</code>	<code>podman-pod-rm(1)</code>	Remove one or more pods
<code>stats</code>	<code>podman-pod-stats(1)</code>	Display statistics for the containers in a pods
<code>start</code>	<code>podman-pod-start(1)</code>	Start a pod
<code>stop</code>	<code>podman-pod-stop(1)</code>	Stop a pod
<code>top</code>	<code>podman-pod-top(1)</code>	Display running process in the pod
<code>unpause</code>	<code>podman-pod-unpause(1)</code>	Unpause all the containers in a pod

## 4.2 Summary

- Pods are a way to group containers together into more complex applications, sharing namespace and resource constraints
- Pods share most of the options that containers use and when you add a container to a Pod, it shares these options with all containers in the Pod

# 5

## *Customization and configuration files*

### **This chapter covers**

- Using Podman configuration files based on libraries used
- Configuring storage.conf file
- Using registries.conf and policy.json files for configuration
- Using containers.conf file to configure other defaults
- Using system configuration files to allow non-root users namespace access

Container engines like Podman have dozens of hard-coded defaults built into them. These defaults determine many aspects of the functional and non-functional behavior of Podman, such as network and security settings. Podman developers try to pick the maximum amount of security but still allow most containers to run successfully. Similarly I want as much isolation from the host as possible.

The security defaults include which Linux capabilities to use, which SELinux labels to set, the set of syscalls available to the containers. There are defaults for resource constraints like memory usage and maximum processes allowed within a container. Other defaults include local path for storing images, list of container registries, and even system configuration to allow rootless mode to work. The Podman developers wanted to allow users to have ultimate control over these defaults, so the container engine configuration files provide a mechanism to customize the way Podman and other container engines run.

The problem with defaults is that these are best-guess estimates from developers. While most users run Podman in default configuration, sometimes there is a need to change the configuration. Every environment does not have the same configuration, and you might want to default certain machines to different levels of security, different registry configurations. Even rootless users might need different configurations than rootful users.

In this chapter, I show you how to customize different parts of Podman and explain where to find more information about all of the different knobs available to you.

As you have learned in previous chapters, Podman uses multiple libraries to perform different tasks when working with containers. Table 5.1 describes the different libraries that Podman uses.

**Table 5.1 Container libraries used by Podman**

Library	Description
containers/storage	Defines the storage of container images and other basic storage used by container engines.
containers/image	Defines the mechanisms used to move container images from different types of storage. Usually used between container registries and local container storage.
containers/common	Defines all of the default configuration options for container engines, not defined in containers/storage or containers/image.
containers/buildah	As explained in chapter 2, it is used for building container images into local storage using rules defined in a Containerfile or Dockerfile. For more information on buildah see appendix B

Each of these libraries have separate configuration files that are used to set the default settings for the particular library, with the exception of Buildah. The container engines, Podman and Buildah share the containers/common configuration file `containers.conf`, described in section 5.3.

### Note

All of the non-system configuration files used by Podman use the TOML format. TOML's syntax consists of name = "value" pairs, [section names], and # comments. The format of TOML can be simplified to:

```
[table]
option = value
[table.subtable1]
option = value
[table.subtable2]
option = value
```

# See <https://toml.io> for a more complete explanation of the TOML language.

When configuring Podman, usually one of the first concerns is about where you are going to store your containers and images.

## 5.1 Configuration files for storage

Podman uses the `github.com/containers/storage` library, which provides methods for storing file system layers, container images, and containers. Configuration of this library is done using the `storage.conf` configuration file, which can be stored in multiple different directories.

Linux distributions often provide a `/usr/share/containers/storage.conf` file, which can be overridden by creating `/etc/containers/storage.conf` file. Rootless users can store their configuration in the `$XDG_CONFIG_HOME/containers/storage.conf` file, if the `$XDG_CONFIG_HOME` environment variable is not set then the file `$HOME/.config/containers/storage.conf` is used. Most users will never change the `storage.conf` file, but in a few situations, advanced users need to do some customizations. The most common reason for changes is to relocate the container's storage.

**NOTE** When using Podman in remote mode, for example on a MAC or Windows box, the `podman` service uses the `storage.conf` files located in the Linux box. To modify them, you need to enter the VM. When using Podman machine, execute the `podman machine ssh` command to enter the VM. See appendix E and F for more information.

Podman reads only one `storage.conf` and ignores all subsequent ones. Podman first attempts to use the `storage.conf` from your home directory, next goes the `/etc/storage/storage.conf`, and finally, if both files do not exist, Podman reads the `/usr/share/containers/storage.conf` file.

You can see the `storage.conf` file that your Podman command is using with the `podman info` command.

```
$ podman info --format '{{ .Store.ConfigFile }}'
/home/dwalsh/.config/containers/storage.conf
```

### 5.1.1 Storage location

By default rootless Podman is configured to store your images in the `$HOME/.local/share/containers/storage` directory. Default rootful storage location is `/var/lib/containers/storage`.

Sometimes you need to change this default location. Perhaps you don't have enough disk space in `/var` or in the user's home directory, so you want to store your images on a different disk. The `storage.conf` file calls the storage location the `graphroot` and it can be overridden in `/etc/containers/storage.conf` for rootful containers.

In this section you modify the location of the graph driver to `/var/mystorage`.

First become root and make sure the `/etc/containers/storage.conf` file exists. If it does not exist, just copy the `/usr/share/containers/storage.conf` file into it.

```
$ sudo cp /usr/share/containers/storage.conf /etc/containers/storage.conf
```



**NOTE** Some distributions just ship the `/etc/containers/storage.conf`.

Now make a backup and open `/etc/containers/storage.conf` file for editing.

```
$ sudo cp /etc/containers/storage.conf /etc/containers/storage.conf.orig
$ sudo vi /etc/containers/storage.conf
```

Set the `graphdriver` variable `graphroot = "/var/lib/containers/storage"` to `graphroot = "/var/mystorage"`, and save the file.

You `storage.conf` file should include this:

```
$ grep -B 1 graph /etc/containers/storage.conf
# Primary Read/Write location of container storage
graphroot = "/var/mystorage"
```

Execute `podman info` to see if the change took place.

```
$ sudo podman info
...
Store:
  configFile: /etc/containers/storage.conf
...
graphDriverName: overlay
graphOptions:
  overlay.mountopt: nodev,metacopy=on
graphRoot: /var/mystorage
...
volumePath: /var/mystorage/volumes
```

Notice in the storage section, that the `graphRoot` is now **`/var/mystorage`**, all images and containers will be stored in this directory.

Now run the `podman info` command in rootless mode, the storage location does not change. It is still **`/home/dwalsh/.local/share/containers/storage`**.

```
$ podman info
store:
  configFile: /home/dwalsh/.config/containers/storage.conf
  containerStore:
    number: 27
    paused: 0
    running: 0
    stopped: 27
  graphDriverName: overlay
  graphOptions: {}
  graphRoot: /home/dwalsh/.local/share/containers/storage
```

You can create a `$HOME/.config/containers/storage.conf` and change it there, but this does not scale well for systems with multiple users. The key `rootless_storage_path` allows you to change the location for all users on your system.

This time uncomment and modify the `rootless_storage_path` line:

```
$ sudo vi /etc/containers/storage.conf
```

Modify the `rootless_storage_path` line in `storage.conf` from:

```
# rootless_storage_path = "$HOME/.local/share/containers/storage"
```

To:

```
rootless_storage_path = "/var/tmp/$UID/var/mystorage"
```

Save the storage.conf file. When you are done it should look like this:

```
$ grep -B 3 rootless_storage_path /etc/containers/storage.conf
# Storage path for rootless users
#
rootless_storage_path = "/var/tmp/$UID/var/mystorage"
```

Now run `podman info` to see the changes. Notice that the `graphRoot` now points at the **`/var/tmp/3267/var/mystorage`** directory.

```
$ podman info
...
store:
  configFile: /home/dwalsh/.config/containers/storage.conf
  ...
  graphOptions: {}
  graphRoot: /var/tmp/3267/var/mystorage
```

Container/storage supports expanding of environment variables `$HOME` and `$UID` for this path.

To revert changes, copy and restore the original storage.conf file.

```
$ sudo cp /etc/containers/storage.conf.orig /etc/containers/storage.conf
```

### Note

If you are running on an SELinux system and change the default location of storage. You need to inform SELinux about it, using the `semanage` command below to tell SELinux to label the new location as if it was in the old location. And then change the labeling on disk using the `restorecon` command.. You can do this with the following commands:

```
sudo semanage fcontext -a -e /var/lib/containers/storage /var/mystorage
sudo restorecon -R -v /var/mystorage
```

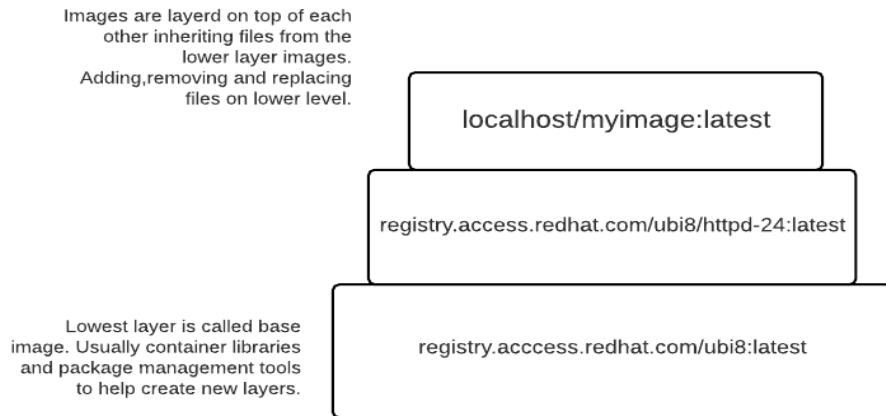
In rootless mode you need to do the following:

```
sudo semanage fcontext -a -e $HOME/.local/share/containers/storage /var/tmp/3267/var/mystorage
sudo restorecon -R -v /var/tmp/3267/var/mystorage
```

Sometimes you might want to change the storage driver, or more likely the configuration of the storage driver.

### 5.1.2 Storage drivers

Recall the wedding cake image from chapter 2.2. This image shows that images are often made of multiple layers. These layers are stored on disk by the container/storage library but when you are running a container on them, each layer needs to be mounted on the previous layer.



**Figure 5.1** Layered images stacked on one another are reassembled and mounted using container/storage.

Container/storage uses a Linux Kernel file system concept called a *layered file system* to do this. Podman, using container/storage, supports multiple different types of layered file systems. In Linux these file systems are called *Copy-On-Write (COW)* file systems. In containers/storage these different file system types are called *drivers*. By default Podman uses the *overlay* storage driver.

**NOTE** Docker supports two types of overlay drivers, overlay and overlay2. Overlay2 was an improvement over overlay, and no one uses the original overlay driver any more. In contrast, Podman uses the newer overlay2 driver and just calls it overlay. You can select the overlay2 driver in Podman, but this is just an alias for overlay.

Table 5.2 lists all of the storage drivers that Podman and containers/storage supports. I recommend that you just stick to the overlay driver, since this is the driver the vast majority of the world uses.

**Table 5.2 Container storage drivers**

Storage Drivers	Description
overlay (overlay2)	Default driver, and I heavily recommend its use. Based on the Linux kernel overlay file system. Overlay and overlay2 are exactly the same. It is the most tested driver which the overwhelming majority of users use.
vfs	Simplest driver, it creates full copies of each lower layer up onto the next layer. It works everywhere but it is slow and very disk intensive.
devmapper	Was heavily used when Docker first became popular, before the overlay driver was available. reallocates the size of each layer at a maximum size. Not recommended any longer.
aufs	Was never merged into the upstream kernel,so it is only available on a few Linux distributions.
btrfs	Allows storage on btrfs snapshots based on the btrfs file system. Some users have had success using this file system.
zfs	Uses the zfs file system which is a proprietary file system, and not available on most distributions.

**OVERLAY STORAGE OPTIONS**

The overlay driver has some interesting customization options. These options are located in the `storage.conf` [storage.options.overlay] table.

There are several advanced options available for configuring the overlay driver. I just want to quickly mention a few to describe use cases.

The `mount_program` option allows you to specify an executable to use instead of the kernel overlay driver. Podman usually ships with the `fuse-overlayfs` executable, which provides a Fuse (User Space) overlay driver. Podman automatically fails over to the `fuse-overlayfs` `mount_program` if it is installed on systems where rootless native overlay is not supported. Most kernels support native overlay, however there are use cases where you might want to configure the `mount_program`. The `fuse-overlayfs` has advanced features that are not currently supported in the native overlay.

Podman is quickly being adopted by the High Performance Computing (HPC) community. The HPC community does not allow rootful containers, and in a lot of cases allows workloads

to run only with a single UID. This means that some HPC systems do not allow users namespaces with multiple UIDs. Since many images come with multiple UIDs, Podman added an `ignore_chown_errors` option to containers/storage to allow images with files with different UIDs to be flattened into a single UID. Table 5.3 lists all of the current storage options supported by container storage.

**Table 5.3 Container storage drivers**

Storage Drivers	Description
<code>ignore_chown_errors</code>	Ignore chowning file UIDs for rootless containers with a single UID. No entry in <code>/etc/subuid</code> .
<code>mount_program</code>	Path to an helper program to use for mounting the file system instead of using kernel overlay to mount it. Older kernels did not support rootless overlay.
<code>mountopt</code>	Comma separated list of mount options to be passed to the kernel. Defaults: <code>"nodev,metacopy=on"</code>
<code>skip_mount_home</code>	Do not create a not create PRIVATE bind mounts on the storage home directory.
<code>inode</code>	Maximum number of inodes in a container image.
<code>size</code>	Maximum size of a container image.
<code>force_mask</code>	<p>Permissions mask for new files and directories in an image.</p> <p>Values:</p> <p><code>"private"</code>: sets all file system objects to 0700. No other users on the system can access the files.</p> <p><code>"shared"</code>: it is equivalent to 0755. Everyone on the system read, access and execute files in image. Useful for sharing container storage with other users.</p> <p>NOTE: All files within the image are made readable and executable by any user on the system. Even <code>/etc/shadow</code> within your image is now readable by any user.</p> <p>When <code>"force_mask"</code> is set the original permission mask is stored in <code>xattr</code>s and the <code>"mount_program"</code> like <code>/usr/bin/fuse-overlayfs</code> presents the <code>xattr</code> permissions to processes within containers.</p>

**Note**

You have examined a few of the `storage.conf` fields. There are many more. Use the `containers-storage.conf` man page to explore all of them. <https://github.com/containers/storage/blob/main/docs/containers-storage.conf.5.md>

```
$ man containers-storage.conf
```

Now that you know about configuring the container storage, the next configuration to look at is container registry access.

## 5.2 Configuration files for registries

Podman uses the `github.com/containers/image` library for pulling and pushing container images, usually from container registries. Podman uses the `registries.conf` configuration file to specify registries and the `policy.json` file for signature verification of images. As with the container storage `storage.conf`, most users never modify these files and just use the distribution defaults.

### 5.2.1 registries.conf

The `registries.conf` configuration file is a system-wide configuration file for container image registries. Podman uses the `$HOME/.config/containers/registries.conf` if it exists; otherwise it uses `/etc/containers/registries.conf`.

**NOTE** When using Podman in remote mode, for example on a MAC or Windows box, `registries.conf` files are stored in the Linux box on the server side. You need to ssh into the linux box to make the changes. With Podman machine, you can execute `podman machine ssh`. See appendix E and F for more information.

The main key value to use with the `registries.conf` file is `unqualified-search-registries`.

This field specifies an array of `host[:port]` registries to try when pulling via short-names, in order.

If you specify only one registry in the `unqualified-search-registries` option, Podman will work similarly to Docker and force a single registry on the user.

In this exercise you are going to modify the default search registries to be used by Podman.

First you need to make a backup of `/etc/containers/registries.conf` file and then remove `docker.io` and add `example.com`.

```
$ sudo cp /etc/containers/registries.conf /etc/containers/registries.conf.orig
$ sudo vi /etc/containers/registries.conf
```

Modify the following line:

```
unqualified-search-registries = ["registry.fedoraproject.org",
    "registry.access.redhat.com", "docker.io", "quay.io"]
```

To:

```
unqualified-search-registries = ["registry.fedoraproject.org",
                                "registry.access.redhat.com", "example.com", "quay.io"]
```

Save the file. And execute `podman info` to verify the changes.

```
$ podman info
registries:
  search:
    - registry.fedoraproject.org
    - registry.access.redhat.com
    - example.com
    - quay.io
```

Now if you attempt to pull via a unknown shortname you should see the following prompt

```
$ podman pull foobar
? Please select an image:
  ▸ registry.fedoraproject.org/foobar:latest
    registry.access.redhat.com/foobar:latest
    example.com/foobar:latest
    quay.io/foobar:latest
```

Now copy back the original to `registries.conf` file.

```
$ sudo cp /etc/containers/registries.conf.orig /etc/containers/registries.conf
```

Table 5.4 describes all of the options available in `registries.conf` files.

**Table 5.4 Container registries.conf global fields**

Fields	Description
unqualified-search-registries	An array of host[:port] registries to try when pulling an unqualified image, in order.
short-name-mode	Determines how Podman should handle short-names. Values: enforcing: If one unqualified-search registry, use it. If 2 or more Podman running in a terminal, prompt the user to select one of the search registries otherwise error. permissive: Behaves as enforcing but does not lead to an error if no terminal just uses each entry in unqualified-search registries until success. disabled: Use all unqualified-search registries without prompting.
credential-helpers	An array of default credential helpers used as external credential stores. Note that "containers-auth.json" is a reserved value to use auth files as specified in containers-auth.json(5). The credential helpers are set to ["containers-auth.json"] if none are specified.

Another interesting thing you can configure in registries.conf is the ability to block users from pulling from a container registry.

#### **BLOCKING PULLING FROM CONTAINER REGISTRIES.**

In this next example, you configure registries.conf to block pulls from docker.io. The registries.conf file has a specific `[[registry]]` table entry that can specify how to handle individual container registries. You can add this table multiple times, once per registry.

```
$ sudo vi /etc/containers/registries.conf
```

Add

```
[[registry]]
Location = "docker.io"
blocked=true
```

Save the file. Examine the settings using podman info.



```
$ podman info
...
registries:
  Docker.io:
    Blocked: true
    Insecure: false
    Location: docker.io
    MirrorByDigestOnly: false
    Mirrors: null
    Prefix: docker.io
  search:
    - registry.fedoraproject.org
    - registry.access.redhat.com
    - docker.io
    - quay.io
```

Now attempt to pull an image from docker.io.

```
$ podman pull docker.io/ubuntu
Trying to pull docker.io/library/ubuntu:latest...
Error: initializing source docker://ubuntu:latest: registry docker.io is blocked in
/etc/containers/registries.conf or /home/dwalsh/.config/containers/registries.conf.d
```

This demonstrates that administrators have the ability to block content from specific registries.

#### NOTE

Copy back the original `registries.conf` in order to pull from `docker.io` for the rest of this book.

```
$ sudo cp /etc/containers/registries.conf.orig /etc/containers/registries.conf
```

Table 5.5 describes the sub options available for the `[[registry]]` table in the `registries.conf` file.

**Table 5.5** `[[registry]]` table fields.

Fields	Description
<code>location</code>	Name of the registry/repository to apply the filters on.
<code>prefix</code>	Select the specified configuration when attempting to pull an image that is matched by the specific prefix.
<code>insecure</code>	If true, unencrypted HTTP as well as TLS connections with untrusted certificates are allowed.
<code>blocked</code>	If true, pulling images with matching names is forbidden.

Some users work on systems that are fully isolated from the internet, but still need to use applications that rely on images from the internet. For example if you have an application that expects to use `registry.access.redhat.com/ubi8/httpd-24:latest`, but has no access to `registry.access.redhat.com` on the internet. You can download the image and put it onto an internal registry and then configure `registries.conf` with a mirror registry. If you configure an entry in `registries.conf` that looks like this.

```
[[registry]]
location="registry.access.redhat.com"
[[registry.mirror]]
location="mirror-1.com"
```

Then your users can use the `podman pull` command

```
$ podman pull registry.access.redhat.com/ubi8/httpd-24:latest
```

Podman actually pulls `mirror-1.com/ubi8/httpd-24:latest`, but they will not notice the difference.

### Note

You have examined a few of the `registries.conf` fields. There are many more. Use the `containers-registries.conf(5)` man page to explore all of them.

```
$ man containers-registries.conf
```

<https://github.com/containers/image/blob/main/docs/containers-registries.conf.5.md>

Now that you know how to configure storage and registries, it is time to look at how to configure all of the options that are central to Podman.

### 5.3 Configuration files for engines

Podman and other container engines use the [github.com/containers/common](https://github.com/containers/common) library for handling the default settings not related to container storage or container registries. These configuration settings come from the `containers.conf` file

Podman reads the following files if they exist:

**Table 5.6 containers.conf files read by both rootful and rootless podman**

File	Description
<code>/usr/share/containers/containers.conf</code>	Usually shipped with the distribution defaults
<code>/etc/containers/containers.conf</code>	System administrator can use this file to set and modify different defaults
<code>/etc/containers/containers.conf.d/*.conf</code>	Some package tools might drop additional default files into this directory, sorted numerically.

When running in rootless mode, Podman also reads these files if they exist:

**Table 5.7 containers.conf files read by rootless podman**

File	Description
<code>\$HOME/.config/containers/containers.conf</code>	Users can create this file, to override system defaults.
<code>\$HOME/.config/containers/containers.conf.d/*.conf</code>	Users can also drop files here, if they want, sorted numerically.

Unlike `storage.conf` and `registries.conf`, `containers.conf` files are merged together, they do not fully override previous versions. Individual fields can override the same field in the higher level `containers.conf` file. Podman does not require any `containers.conf` file to exist, since it has built-in defaults. Most systems come with only the distribution default overrides in `/usr/share/containers/containers.conf`

**NOTE** Podman supports the `CONTAINERS_CONF` environment variable, which forces Podman to use the target of the `$CONTAINER_CONF`. All other `containers.conf` files are ignored. This is useful for testing environments or to make sure that no one has customized the Podman defaults.

Containers.conf currently supports five different tables, as shown in table 5.5. You need to be careful that you modify the options within the correct table.

**Table 5.8 Containers.conf tables**

table	Description
[containers]	Configuration on how to run individual containers. Examples are the namespaces to stick containers in, whether or not SELinux is enabled, default environment variables for containers.
[engine]	Default configurations for Podman to use. Examples are default logging system, paths to OCI Runtimes to use, and the location of common.
[service_destinations]	Remote connection data for use with podman --remote. Remote service is covered in chapter 9.
[secrets]	Information about the secrets plugin driver to use for containers.
[network]	Special configuration for network configuration. Default network name, location of cni plugins and default subnets.

Many users of Podman want to change the default ways that it launches containers in an environment. I previously explained how the HPC community wants to use Podman to run their workloads, but they are very specific about the volumes that get added to containers, which environment variables are added, and which namespaces are enabled.

Perhaps you want all of your containers to have the same environment variables set. Let's try an example. Run podman to show the default environment in the ubi8 image.

```
$ podman run --rm ubi8 printenv
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
TERM=xterm
container=oci
HOME=/root
HOSTNAME=ba4acf180386
```

**NOTE** When using Podman in remote mode, for example on a MAC or Windows box. Most of the settings of the containers.conf files are used from the Linux box on the server side. A containers.conf file in the users home directory is used for storing connection data, which is covered in chapter 9. Remote Clients MAC and Windows Appendix E and F.

Now lets create an env.conf file in the home directory with the **env="[foo=bar]"** set.

```
$ mkdir -p $HOME/.config/containers/containers.conf.d
$ cat << _EOF > $HOME/.config/containers/containers.conf.d/env.conf
[containers]
env=[ "foo=bar" ]
_EOF
```

Run any container and you see the **foo=bar** environment set.

```
$ podman run --rm ubi8 printenv
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
TERM=xterm
container=oci
foo=bar
HOME=/root
HOSTNAME=406fc182d44b
```

I use containers.conf when configuring Podman to run within a container. Many users want to run Podman within a container for CI/CD systems or for just testing out newer versions of Podman then their distribution enables. When lots of people were opening issues because they were having a hard time running it, I decided to try to set up a default image [quay.io/podman/stable](https://quay.io/podman/stable) to help them get it done. While creating that image, I realized that a lot of the Podman defaults did not work well when running it within a container so I used containers.conf to change those settings. You can see my containers.conf file at this link: <https://github.com/containers/podman/blob/main/contrib/podmanimage/stable/containers.conf>

You can see the contains.conf by actually running the image.

```
$ podman run quay.io/podman/stable cat /etc/containers/containers.conf
[containers]
netns="host"
userns="host"
ipcns="host"
utsns="host"
cgroupns="host"
cgroups="disabled"
log_driver = "k8s-file"
[engine]
cgroup_manager = "cgroupfs"
events_logger="file"
runtime="crun"
```

Here was what I was thinking while writing this file. First I decided that since Podman is running inside of a container, I disable all of the cgroups and namespaces other than the mount and user namespace. If users set cgroups or configured namespaces, then the container run by Podman in a container follows the parents Podman's rules.

```
[containers]
netns="host"
userns="host"
ipcns="host"
utsns="host"
cgroupns="host"
cgroups="disabled"
```

The default `log_driver`, event logger and `cgroup` manager on a lots of distributions is `journald` and `systemd` respectively, but inside of the container, `systemd` and `journald` are not running, so the container engine needs to use the file system

```
[containers]
log_driver = "k8s-file"
[engine]
cgroup_manager = "cgroupfs"
events_logger="file"
```

Finally use the OCI runtime `crun` rather than `runc`, mainly because `crun` is a lot smaller than `runc`.

```
[engine]
runtime="crun"
```

Now attempt to run a container within a container. A trick needed to make this work is to run the `podman/stable` image with `--user podman`, this causes the Podman inside of the container to run in rootless mode. Since the `podman/stable` image uses the `fuse-overlay` driver within the container you also need to add the `/dev/fuse` device.

```
$ podman run --device /dev/fuse --user podman quay.io/podman/stable podman run ubi8-micro
echo hi
Resolved "ubi8" as an alias (/etc/containers/registries.conf.d/000-shortnames.conf)
Trying to pull registry.access.redhat.com/ubi8:latest...
Getting image source signatures
Copying blob sha256:5368f457acd16b337e2b150741f727c46f886c69eea1a4d56d0114c88029ed87
...
hi
```

### Note

You examined a few of the `containers.conf` fields. There are many more. Use the `container.conf(5)` man page to explore all of them.

```
$ man containers.conf
```

<https://github.com/containers/common/blob/main/docs/containers.conf.5.md>

I have finished covering the configuration tools specific to container tools like Podman, but there are still some system configuration files that Podman needs.

## 5.4 System configuration files

When you run rootless Podman, you are using the `/etc/subuid` and `/etc/subgid` files to specify the UID ranges for your containers. As I explained in section 3.1.2.1, (User namespace), Podman reads the `/etc/subuid` and `/etc/subgid` files for UID and GID ranges allocated for your user account. Podman then launches `/usr/bin/newuidmap` and `/usr/bin/newgidmap` which verifies the range of UIDs and GIDs that Podman specified are actually allocated to you. In certain cases you need to modify these files to add UIDs. Tools like `useradd` automatically

update the `/etc/subuid` and `/etc/subgid` when you add new users to your system. For example when I installed my laptop `useradd` set up my user account to use UID 3267 and added the mapping `dwalsh:100000:65536` to `/etc/subuid` and `/etc/subgid`. Figure 5.2 shows what containers based on this mapping look like on my system.

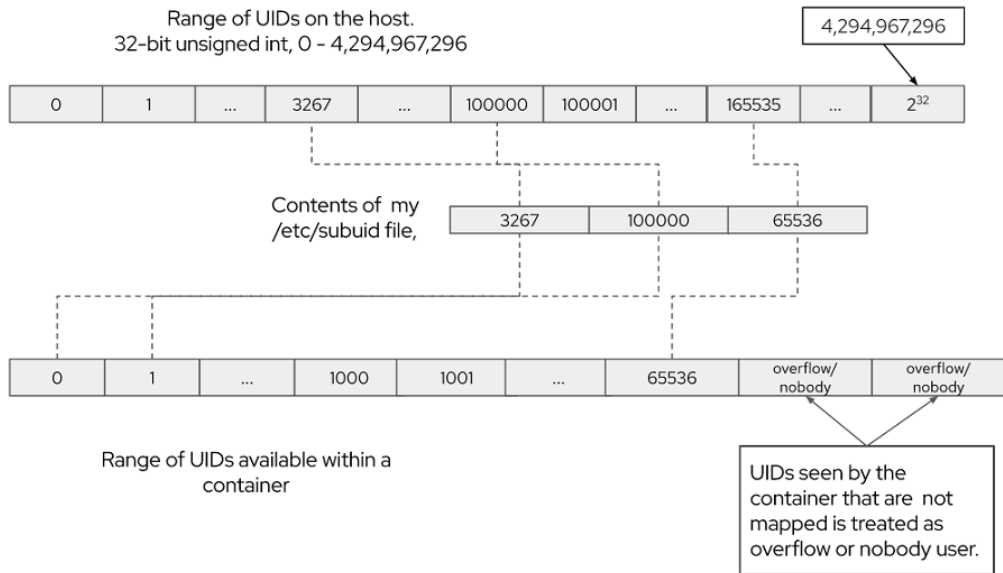


Figure 5.2 User namespace mapping for containers.

**NOTE** You want to keep the ranges of UIDs unique for each user and not to overlap with any system UIDs. Podman and the system do not verify that there is no overlap. If two different users had the same UIDs in their range, the processes in the containers are allowed to attack each other from the User Namespace perspective. This is a manual process to verify. The `useradd` tool automatically selects unique ranges.

As the `subuid(5)` and `subgid(5)` man pages explain, each line in `/etc/subuid` and `/etc/subgid` contains a user name and a range of subordinate user ids, or gids respectively, that the user is allowed to use. The entry is specified with three fields delimited by colons (":"). These fields are:

- Login name or UID
- Numerical subordinate user ID or group ID
- Numerical subordinate user ID or group ID count

Newer versions of the operating system, specifically the packages that ship `/usr/bin/newuidmap` and `/usr/bin/newgidmap`, are gaining the ability to share the contents of these files via the network from an LDAP Server. On Fedora, these executables are shipped in the `shadow-utils` package. Versions 4.9 or later have this feature.

---

**Tip: Changes to `/etc/subuid` and `/etc/subgid` may not be immediately reflected in the login users account.**

This is a common issue from users who modify these files after they have already run Podman. But remember when Podman first runs it launches the `podman pause` process in the user namespace and then all other containers join this Podman processes user namespace. In order to get the new user namespace to take effect you must execute the `podman system migrate` command, which stops the `podman pause` process, and recreates the user namespace.

---

## 5.5 Summary

- Podman has multiple configuration files based on the libraries that it uses
- Configuration files are shared between the rootful and the rootless environments
- The `storage.conf` is used to configure containers/storage. Configuring storage driver as well as the location where containers and their images are to be stored
- The `registries.conf` and `policy.json` files are used to configure the container/image library, primarily the access to container registries, short names, and mirror sights
- The `containers.conf` file used to configure all of the other defaults used within Podman
- System configuration files `"/etc/subuid"`, `"/etc/subgid"` are used to configure the user namespace required for running rootless Podman



# 6

## *Rootless containers*

### **This chapter covers**

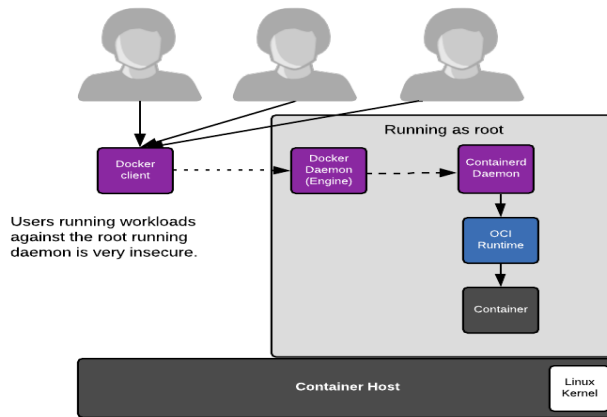
- **Why rootless mode is more secure**
- **How Podman works with the user and mount namespace**
- **The architecture of Podman running in rootless mode**

I take a deep dive into what is going on when running Podman in a rootless mode. I believe it is helpful to understand what is happening when you run rootless containers and learn about the issues that running in rootless mode can cause.

With the introduction of containerized applications over the last few years, certain highly secure environments were not able to take advantage of the new technology.

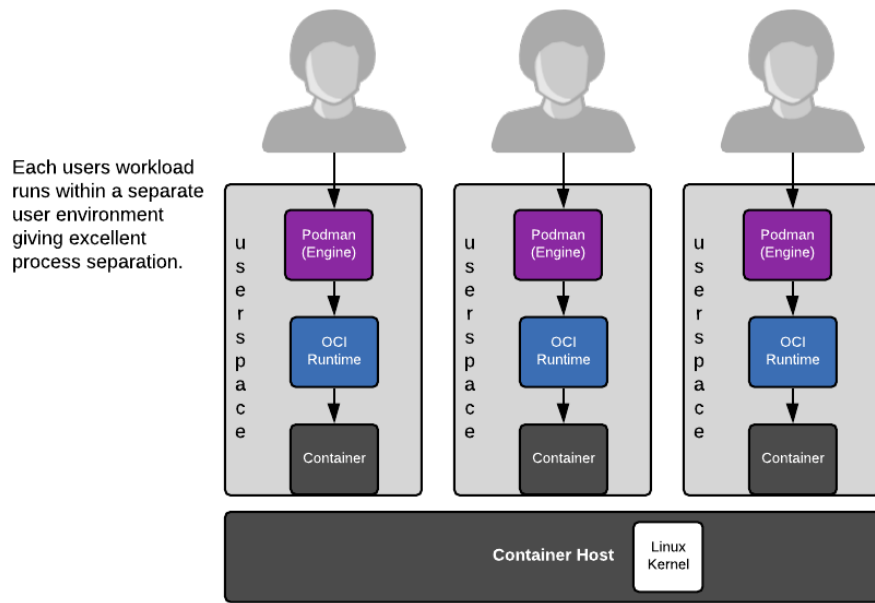
High performance computing (HPC) systems run the fastest computers in the world. These tend to be national labs and universities and deal with high security information. They handle some of the most secure data in the world and expressly forbid the use of rootful containers. HPC systems deal with huge data sets, like artificial intelligence, nuclear weapons, global weather patterns, medical research, etc. These systems tend to have thousands of shared computers. These systems need to be locked down, because of their multi-user shared environments. HPC computing believes that root running daemons are too insecure. If a rogue container process breaks out of confinement and gains root access, it can access highly sensitive data. Administrators of HPC environments couldn't use open container initiative (OCI) containers until Podman came along. The HPC community is now working to move to rootless Podman.

Similarly large financial company administrators do not allow users and developers access to *root* on their shared computer systems, out of concern for the financial data involved. The largest financial firms in the world were having difficulty fully adopting OCI containers.



**Figure 6.1** Multiple users' workloads sharing the same root running daemon is inherently insecure.

Bottom line: allowing users on a shared computing system to run container workloads accessing the same root running daemon, is too insecure. Running each user's containers in rootless mode under different users accounts is more secure.



**Figure 6.2 Each workload running within their unique user space is more secure.**

Linux was designed from the ground up with a separation between privileged mode and unprivileged mode, root, and rootless. In Linux almost all tasks run without being privileged. Privileged operations are only required for modifications to the core operating system. Almost all applications that run in containers, web servers, databases, and user tools run without requiring root. The applications do not modify core parts of the system.

Sadly most of the images that you find on container registries are built to require root privileges or at least start as root and then drop privileges.

In the corporate world, administrators are very reluctant to give out root access to their users. If you receive a corporate laptop from your employer, usually you are not granted any root access. Administrators need to control what is installed on their systems, because of scale. Administrators need to be able update hundreds to thousands of machines at the same time, so controlling what is in the OS is critical. If someone else is administering your machine, they need to control who gets root access.

As a security person I still flinch a little when I see sudo without a password. When I first started working with Docker, I was shocked that it was encouraging the use of the docker group, giving users full root access on the host, without password. The holy grail of hackers is to get a root exploit; this means that they gain full control over the system.

Bottom line is that if you have a container escape, as bad as that is, you are better off in rootless mode: The hackers have control over only non-privileged processes, as opposed to a

root exploit where they have full control over the system and all of the data (ignoring other security mechanisms like SELinux).

Podman's design goals include the ability to run as many workloads as possible without being root and to push the core OS to make it easier for you to run in this more secure mode.

## 6.1 How does rootless Podman work?

Have you ever wondered what happens behind the scenes of a rootless Podman container? In chapter 2 all of the podman examples were running in rootless mode. Let's take a look at what happens under the hood of rootless Podman containers. I'll explain each component and then break down all of the steps involved.

**NOTE** Some of this section is copied and rewritten from the:

"What happens behind the scenes of a rootless Podman container?"

Blog (<https://www.redhat.com/sysadmin/behind-scenes-podman>), written by myself and coworkers Mathew Heon and Giuseppe Scrivano.

First, let's first clear out all storage so you can get a fresh environment, and then run a container on quay.io/rhatdan/myimage. (Remember: the `podman rmi --all --force` command removes all images and containers from storage.)

```
$ podman rmi --all --force
Untagged: registry.access.redhat.com/ubi8/httpd-24:latest
Untagged: registry.access.redhat.com/ubi8-init:latest
Untagged: localhost/myimage:latest
Untagged: quay.io/rhatdan/myimage:latest
Deleted: d2244a4379d6f1981189d35154beaf4f9a17666ae3b9fba680ddb014eac72adc
Deleted: 82eb390304938f16dd707f32abaa8464af8d4a25959ab342e25696a540ec56b5
Deleted: 8773554aad01d4b8443d979cdd509e7b8fa88ddbc966987fe91690d05614c961
```

Now that you have a clean system, you need to retrieve the application image, quay.io/rhatdan/myimage, from the container registry that you pushed it to in chapter 2. In the following command re-create the application on your machine. The following Podman command pulls the image back from the container registry and starts the myapp container on your host.

```
$ podman run -d -p 8080:8080 --name myapp quay.io/rhatdan/myimage
Trying to pull quay.io/rhatdan/myimage:latest...
...
2f111737752dcbf1a1c7e15e807fb48f55362b67356fc10c2ade24964e99fa09
```

Now, let's dig deep into what just happened when you ran a rootless Podman container.

The first thing that happens when you run a rootless container is that Podman needs to set up the user namespace. In the next section, I explain why, and how it works.

### 6.1.1 Images contain content owned by multiple user identifiers (UIDs)

In Linux user identifiers (UIDs) and group identifiers (GIDs) are assigned to processes and stored on file system objects. The file system objects also have permission values assigned to

the file system objects. Linux controls the processes' access to the file system based on these UIDs and GIDs. This access is called discretionary access control (DAC). When you log in to a Linux machine, your rootless user processes run with a single UID, say 1000, but container images usually come with multiple different UIDs in their image layers. Let's examine the UIDs needed to run our image. In this example you examine all of the UIDs defined within the container image, by running another container.

In the command below, you launch a container with the **quay.io/rhatdan/myimage** image. You need to run the container as root (**--user=root**) inside the container to examine every file within the image.

```
$ podman run --user=root --rm quay.io/rhatdan/myimage -- bash -c "find / -mount -printf \"%U=%u\n\" | sort -un" 2>/dev/null
```

Since this is only a temporary container, you use the **--rm** option to make sure the container is removed when it finishes running. The container runs a bash script which finds all of the UIDs and USERS associated with every file/directory in the container. The script pipes the output to sort to show unique entries and redirects stderr to /dev/null to eliminate any errors.

```
$ podman run --user=root --rm quay.io/rhatdan/myimage -- bash -c "find / -mount -printf \"%U=%u\n\" | sort -un" 2>/dev/null
0=root
48=apache
1001=default
65534=nobody
```

As you can see from the output our container image uses four different UIDs shown in table 6.1.

**Table 6.1 Unique UIDs required to run the container image**

UID	Name	Description
0	root	Owns most of the content within the container image.
48	apache	Owns all of the Apache content.
1001	default	Default user which the container runs as.
65634	nobody	Assigned to any UID that is not mapped into the container.

In order for you to pull a container image to your homedir, Podman needs to store at least three different UIDs: 0, 48, 1001. Since the Linux kernel prevents non-privileged accounts from using more than a single UID, you are prevented from creating files with different UIDs.

You need to take advantage of the user namespace.

## USER NAMESPACE

Linux supports the concept of user namespaces, which is a mapping of UID/GIDs from the host to different UIDs and GIDs inside the namespace. Here is how the man page describes it:

```
$ man user namespaces
```

```
...
User namespaces isolate security-related identifiers and attributes, in
particular, user IDs and group IDs (see credentials(7)), the root directory, keys(see
keyrings(7)), and capabilities (see capabilities(7)). A process's user and group IDs
can be different inside and outside a user namespace. In particular, a process can have
a normal unprivileged user ID outside a user namespace while at the same time having a
user ID of 0 inside the namespace; in other words, the process has full privileges for
operations inside the user namespace, but is unprivileged for operations outside the
namespace.
```

Since your container requires more than one UID, the Podman process first creates and enters a user namespace where it has access to more UIDs. Podman must also mount several file systems in order to run a container. These mount commands are not allowed outside a user namespace (along with a mount namespace).

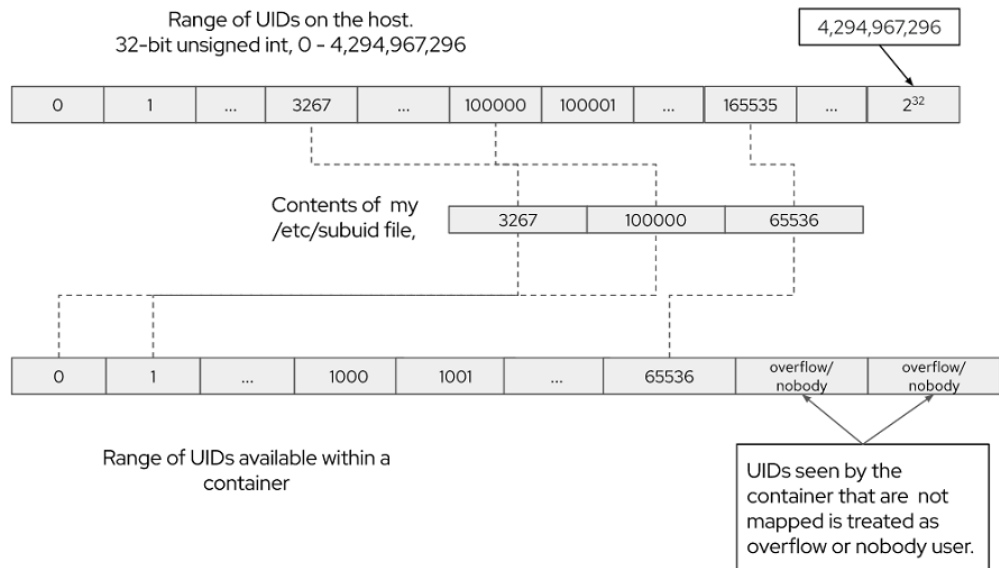


Figure 6.3 User namespace mapping for containers.

When I created my system, I used the `useradd` program to create my account. `Useradd` assigned me, 3267 as my UID and GID, defined in `/etc/passwd` and `/etc/group`. `Useradd` also allocated UID 100000-1065535 additional UIDs and GIDs for me defined in `/etc/subuid` and `/etc/subgid`. Let's see the content of these files:

```
$ cat /etc/subuid
dwalsh:100000:65536
Testuser:165536:65536
$ cat /etc/subgid
dwalsh:100000:65536
Testuser:165536:65536
```

Cat these files on your system you'll see something similar. On my system I also have a testuser account and you see that useradd also added UIDs/GIDs for that user, starting right after my allocation.

Within a user namespace, I have access to UIDs 3267 (my UID) as well as UIDs 100000,100001,100002,...,165535, for a total of 65537 UIDs. A root user can modify the /etc/subuid and /etc/subgid files to increase or decrease this number.

The useradd command starts at UID 100000 in order to allow you to have around 99000 regular users plus 1000 UIDs reserved for system services on a Linux system. The kernel supports more than 4 billion UIDs ( $2^{32}=4294967296$ ). Since useradd allocates 65537 per user, Linux can support more than 60000 users. The 65536 ( $2^{16}$ ) number was picked because up until the Linux kernel 2.4, this was the maximum number of users on a Linux system. Let's look deeper into the User Namespace.

Every process on a Linux system is in namespaces, including the init process, systemd. These are the host namespaces. Therefore every process is in a user namespace. You can see the user namespace mapping for your process by examining the /proc file system. The /proc/PID/uid\_map and /proc/PID/gid\_map contains the user namespace mappings for each process on the OS. /proc/self/uid\_map contains the UID map of the current process.

```
$ cat /proc/self/uid_map
0      0 4294967295
```

The mapping means that UIDs starting at UID 0 are mapped to UID 0 for a range of 4294967295 UIDs.

Another way of looking at this mapping is:

UID 0->0, 1->1,...3267->3267,...,4294967294->4294967294.

Basically there is no mapping, so root is root. And my UID 3267 is mapped to 3267--itself.

Now let's enter the user namespace and see what is mapped. Podman has a special command `podman unshare` which allows you to enter a user namespace without launching a container. It allows you to examine what is going on within the user namespace, while still running as a regular process on your system.

In the following command, I run `podman unshare` to launch the `cat /proc/self/uid_map` within the default user namespace for my account:

```
$ podman unshare cat /proc/self/uid_map
0      3267      1
1      100000   65536
```

The mappings show that UID 0 is mapped to UID 3267 (my UID) for a range of 1. Then UID 1 is mapped to UID 100000 for a range of 65536 UIDs.

Any UID that is not mapped to the user namespace is reported within the user namespace as the `nobody` user. You saw this earlier when you searched for the UIDs within the container image.

```
$ podman run --user=root --rm quay.io/rhatdan/myimage -- bash -c "find / -mount -exec stat
-c %u=%U {} \; | sort -un" 2>/dev/null
0=root
48=apache
1001=default
65534=nobody
If you look at / on the host, you see it is owned by the real root.
$ ls -l -ld /
dr-xr-xr-x. 18 root root 242 Sep 21 22:32 /
```

If you examine the same directory within the user namespace, you see it as owned by the `nobody` user.

```
$ podman unshare ls -ld /
dr-xr-xr-x. 18 nobody nobody 242 Sep 21 22:32 /
```

Since the host's UID 0 is not mapped into the user namespace, the kernel reports the UID as the `nobody` user. Processes within the user namespace have access to only `nobody` files based on only the "other" or "world" permissions. In the example that follows, you launch a bash script that shows the user is root within the user namespace but sees `/etc/passwd` as owned by the user `nobody`. You can read the file with the `grep` command because `/etc/passwd` is world readable. But the `touch` command fails because even root cannot modify files owned by UIDs not mapped to the user namespace.

```
$ podman unshare bash -c "id ; ls -l /etc/passwd; grep dwalsh /etc/passwd; touch
/etc/passwd"
uid=0(root) gid=0(root) groups=0(root),65534(nobody)
-rw-r--r--. 1 nobody nobody 2942 Sep 28 07:08 /etc/passwd
dwalsh:x:3267:3267:Dan Walsh:/home/dwalsh:/bin/bash
touch: cannot touch '/etc/passwd': Permission denied
```

Looking at your home directory on the host versus inside of the user namespace, you see that the same files are reported as being owned by your UID.

```
$ ls -ld /home/dwalsh
drwx-----. 365 dwalsh dwalsh 24576 Sep 28 07:30 /home/dwalsh
```

Within the user namespace, they are owned by root.

```
$ podman unshare ls -ld /home/dwalsh
drwx-----. 365 root root 24576 Sep 28 07:30 /home/dwalsh
```

Podman by default maps your UID to root within the user namespace. Podman defaults to root because, as I specified at the beginning of this chapter, the majority of container images assume they start with root.

One last example. Create a directory and a file within the directory while in the user namespace and use the `chown` command to change the contents UIDs to 1:1..

```
$ podman unshare bash -c "mkdir test;touch test/testfile; chown -R 1:1 test"
```



Outside the user namespace, you see the testfile is owned by UID 100000:

```
$ ls -l test
total 0
-rw-r--r--. 1 100000 100000 0 Sep 28 07:53 testfile
```

When you create the testfile and chown it to UID/GID 1:1 within the user namespace, the on-disk owner is actually UID 100000/100000. Remember within the user namespace, UID 1 is mapped to UID 100000, so when you create a UID 1 file within the user namespace, the OS actually creates UID 100000.

If you attempt to remove the file outside of the user namespace, you get an error:

```
$ rm -rf test
rm: cannot remove 'test/testfile': Permission denied
```

Outside the user namespace, you have access to only your UID, you don't have access to the additional UIDs.

**NOTE** In chapter 3 section 3.1.2.1, I showed how this can be problematic with container volumes, and discuss ways that you can handle it.

Re-entering the user namespace, you can remove the file.

```
$ podman unshare rm -rf test
```

Hopefully you are starting to get a feel for the user namespace; the `podman unshare` command makes it easy to explore your system within the user namespace and understand what is happening in rootless containers.

When running a rootless container, Podman needs more than just to run as root, it also needs access to some of the special powers of root called Linux capabilities.

In Linux, the root processes actually are not all equally powerful. Linux breaks root privileges into a series of Linux capabilities. A root process with all Linux capabilities is all-powerful. A root process without Linux capabilities is not allowed to manipulate a lot of the system; for example, it cannot read non-root files unless those files have permission flags which allow all UIDs on the system to read (world-readable).

Let's see how capabilities work with the user namespace:

```
$ man capabilities
...
DESCRIPTION
For the purpose of performing permission checks, traditional UNIX implementations
distinguish two categories of processes: privileged processes (whose effective user
ID is 0, referred to as superuser or root), and unprivileged processes (whose
effective UID is nonzero). Privileged processes bypass all kernel permission checks,
while unprivileged processes are subject to full permission checking based on the
process's credentials (usually: effective UID, effective GID, and supplementary
group list).
Starting with kernel 2.2, Linux divides the privileges traditionally associated with
superuser into distinct units, known as capabilities, which can be independently
enabled and disabled. Capabilities are a per-thread attribute.
```

Linux currently has around 40 capabilities. Examples are `CAP_SETUID` and `CAP_SETGID`, which allow processes to change their UIDs and GIDs. `CAP_NET_ADMIN` allows you to manage the network stack.

Another capability called `CAP_CHOWN` allows processes to change the UID/GID of files on disk. In the preceding example, where you chowned the test directory to 1:1, you used the `CAP_CHOWN` capability within the user namespace.

```
$ podman unshare bash -c "mkdir test;touch test/testfile; chown -R 1:1 test"
```

When you run within a user namespace you are using namespaced capabilities. The root user within your user namespace has these capabilities beyond the UIDs/GIDs defined within the namespace. Processes with the namespaced capability, `CAP_CHOWN`, are allowed to chown files owned within your user namespace to UIDs that are also within the user namespace. If a process within a user namespace attempts to chown a file not mapped to the user namespace, owned by the `nobody` user, the process is denied permission. Likewise, a process attempting to chown a file with a UID not defined within the user namespace also gets denied.

Similarly the `CAP_SETUID` capability only allows processes to change UIDs to UIDs defined within the user namespace.

When Podman runs a container it needs to mount several file systems for the container. In Linux the `CAP_SYS_ADMIN` capability is required for mounting file systems. From a security point of view, mounting file systems can be a dangerous thing to do on Linux. The kernel adds additional controls on which types of file systems can be mounted and requires your user namespaced processes to also be in a unique mount namespace. In a later chapter, you see how Podman limits the number of Linux capabilities available to the namespaced root within a container.

## MOUNT NAMESPACE

Mount namespaces allow processes within them to mount file systems where the mount points are not seen by processes outside the mount namespace. Inside a mount namespace you can mount a tmpfs on `/tmp`, which blocks the processes within the namespaces view of `/tmp`. Outside the mount namespace, processes still see the original mount and files within `/tmp`, they do not see your mount.

In rootless containers Podman needs to mount the content in the container images, as well as `/proc`, `/sys`, devices from `/dev`, and some tmpfs file systems. For that, Podman needs to create a mount namespace.

```
$ man mount namespaces
```

```
...
Mount namespaces provide isolation of the list of mount points seen by the processes in
each namespace instance. Thus, the processes in each of the mount namespace instances
see distinct single-directory hierarchies.
```

When you execute the `podman unshare` command you are actually entering a different mount namespace as well as a different user namespace.

You can examine a process's namespaces by listing the `/proc/self/ns/` directory, as follows:

```
$ ls -l /proc/self/ns/user /proc/self/ns/mnt
lrwxrwxrwx. 1 dwalsh dwalsh 0 Sep 28 09:17 /proc/self/ns/mnt -> 'mnt:[4026531840]'
lrwxrwxrwx. 1 dwalsh dwalsh 0 Sep 28 09:17 /proc/self/ns/user -> 'user:[4026531837]'
```

Notice that when I enter the user namespace and mount namespace the identifiers change:

```
$ podman unshare ls -l /proc/self/ns/user /proc/self/ns/mnt
lrwxrwxrwx. 1 root root 0 Sep 28 09:17 /proc/self/ns/mnt -> 'mnt:[4026533087]'
lrwxrwxrwx. 1 root root 0 Sep 28 09:17 /proc/self/ns/user -> 'user:[4026533086]'
```

In the following test, you can create a file on /tmp and then attempt to bind mount it onto /etc/shadow. Outside the namespaces, the kernel rightly prevents me from mounting the file, as you can see in the following output:

```
$ echo hello > /tmp/testfile
$ mount --bind /tmp/testfile /etc/shadow
mount: /etc/shadow: must be superuser to use mount.
```

Once you enter the user namespace and mount namespace, your namespaced process can successfully mount over the /etc/shadow file. You can see when you run the following command that /etc/shadow is actually modified:

```
$ podman unshare bash -c "mount -o bind /tmp/testfile /etc/shadow; cat /etc/shadow"
hello
```

Once you exit the unshare, everything is back to normal.

#### **USER NAMESPACE + MOUNT NAMESPACE**

As you saw above, when you over-mounted the /etc/shadow file, you might trick some setuid applications like /bin/su or /bin/sudo into giving you full root. The reason rootless users are not allowed to mount file systems, was to prevent this type of attack.

As you have seen, the separate mount namespace prevents you from affecting the host's view of the system, anything you mount is seen only within the mount namespace. Within the user namespace, the container already has a namespaced root. Attacks on your mount points can be escalated to root only within the user namespace, not real root on the host. Containerized processes can not gain setuid to real root or any other UID not mapped into the user namespace.

Even with the namespaces, the Linux kernel only allows you to mount certain file systems types. Many file system types are too dangerous to allow for rootless users, because they gain access to sensitive parts of the kernel. I work with file system kernel engineers to see if there are ways to lock down other file system types which could be allowed to be mounted in rootless mode, without affecting the security of the system.

As of kernel 5.13, the kernel engineers added native overlay mounts to the list of allowed mounts. The file system types currently allowed are listed in table 6.2.

**Table 6.2 File system mounts currently supported in rootless mode**

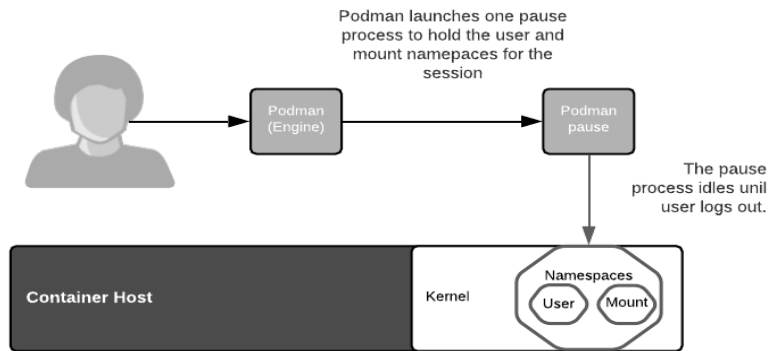
Mount type	Description
bind	Used heavily in rootless containers. Because rootless users are not allowed to create devices, Podman bind mounts <code>/dev</code> on the host into the container. Podman also uses bind mounts to obscure content within the host file system from containers. Bind mounting <code>/dev/null</code> over files in <code>/proc</code> and <code>/sys</code> . Volume mounts, described in chapter 3, also use bind mounts.
binderfs	Filesystem for the Android binder IPC mechanism. Not supported by Podman.
devpts	Virtual filesystem mounted at <code>/dev/pts</code> . Contains device files used for terminal emulators
cgroupfs	Kernel file system used to manipulate cgroups, rootless containers can use cgroupfs to manipulate cgroups in cgroups V1. On v1 this is not supported. Mounted at <code>/sys/fs/cgroups</code> .
FUSE	Used to mount container images using the fuse-overlayfs in rootless mode. Prior to kernel 5.13 this was the only way to use an overlay file system in rootless mode.
procfs	Mounted at <code>/proc</code> within the container. You can examine processes within the container.
mqueue	Implements the POSIX message queues API. Podman mounts this file system at <code>/dev/mqueue</code> .
overlayfs	Used for mounting the image. Performs better in than fuse-overlayfs file system. In certain use cases it provides benefits over native overlay like nfs home directories.
ramfs	Dynamically resizable ram-based Linux filesystem, currently not used with Podman.
sysfs	Mounted at <code>/sys</code> .
tmpfs	Used to obscure kernel file system directories from containers in <code>/proc</code> and <code>/sys</code> .

## 6.2 Rootless Podman under the covers

Now that you have some understanding of how the user namespace and mount namespace work and why they are needed, let's dig deeper into what Podman does when it runs a container.

The first time you run a Podman container after logging in, Podman reads the `/etc/subuid` and `/etc/subgid` files, looking for your username or UID. Once Podman finds the entry, it uses the contents as well as your current UID/GID to generate a user namespace for you.

Podman then launches the `podman pause` process to hold open the user and mount namespace (figure 6.4).



**Figure 6.4** Podman launches the pause process to hold open the user and mount namespaces.

Users commonly report that after they run Podman containers, they see a podman process still running when they run the following command:

```
$ ps -e | grep podman
2541 ?          00:00:00 podman pause
```

Subsequent running of the Podman commands joins the namespaces of the podman pause process. Podman does this to avoid race conditions when user namespaces are coming up and going down. The pause process remains running until you log out. You can also execute the `podman system migrate` command to remove it. The pause process's role is to keep the user namespace alive, as all rootless containers must be run in the same user namespace. If they were not, sharing content and other namespaces (like sharing the network namespace from another container) is impossible.

**NOTE** I often have users report that when changing the `/etc/subuid` and `/etc/subgid` file, their containers don't reflect the changes right away. Since the pause process was launched with the previous user namespace settings, it needs to be removed. Executing the `podman system migrate` command restarts the pause process within the user namespace.

You can kill the pause process at any time, but Podman recreates it on the next run.

By default each rootless user has their own user namespace, and all of their containers run within the same user namespace. You can subdivide the user namespace and run containers with different user namespaces, but realize, by default, you only have 65k UIDs to work with.

Running multiple containers in different User namespaces is much easier to do when running rootful containers.

Now that the user namespace and mount namespace are created, Podman creates storage for the container's image and sets up a mount point to start storing the image.

### 6.2.1 Pulling the image

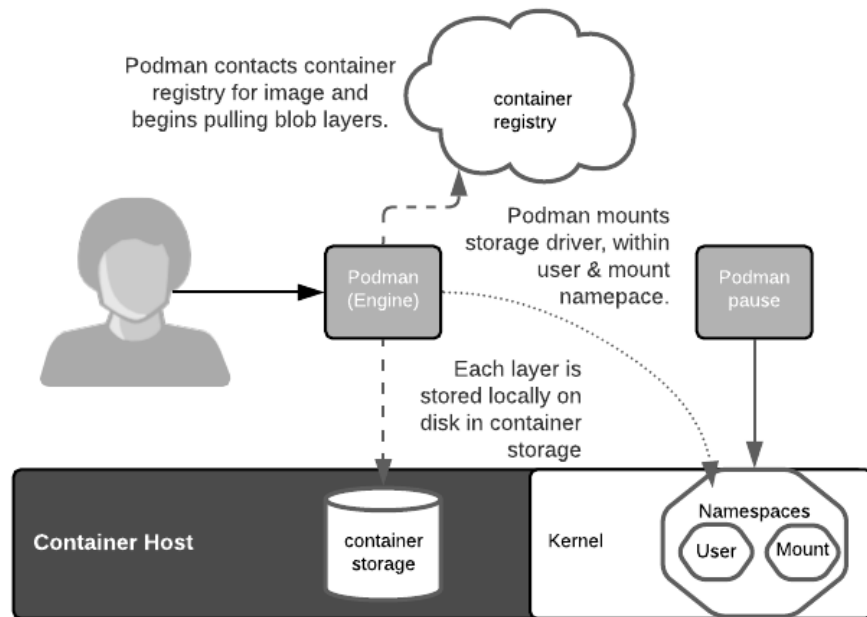


Figure 6.5 Podman pulls an image off of a container registry and stores it in the container storage.

When pulling the image, Podman checks if the container image `quay.io/rhatdan/myimage` exists in local container storage. If it does, then Podman sets up the container network (more about this in the next section). However, if the container image does not exist, Podman uses the `containers/image` library to pull the image. Here are the steps Podman does while pulling the image:

1. Resolve IP address for the registry, quay.io.
2. Connect to the IP address via the HTTPS port (443).
3. Begin pulling the manifest, all layers and the config of the image using the HTTP protocol.
4. Find the multiple layers or blobs of quay.io/rhatdan/myimage
5. Copy all layers simultaneously from the container registry to the host.

As each layer is copied to the host, Podman uses the containers/storage library to reassemble the layers in order, creating an overlay mount point for each of them on top of the previous one in `~/.local/share/containers/storage`. If there is no previous layer, it creates the initial layer.

Next, containers/storage untars the contents of the layer into the new storage layer. As the layers are untarred, containers/storage chowns the UID/GIDs of files in the tarball into the home directory. Podman takes advantage of the user namespace CAP\_CHOWN as explained in previous sections. Remember that Podman fails to create content if the UID or GID specified in the tar file was not mapped into the user namespace.

## 6.2.2 Creating a container

Once the containers/storage library finishes downloading the image and creating the storage, Podman creates a new container based on the image. Podman adds the container to Podman's internal database. Podman tells containers/storage to create writable space on disk and use the default storage driver, usually overlayfs, to mount this space as a new container layer. The new container layer acts as the final read/write layer and is mounted on top of the image.

**NOTE** Rootful containers default to using native Linux overlay mounts. In rootless mode, on kernel versions newer than 5.13 or a kernel with the rootless overlay feature backported (RHEL 8.5 kernels or later also have this feature), use the native overlay mounts. On older kernels Podman uses the fuse-overlayfs executable to create the layer. In Podman overlay and overlay2 are the same drivers.

At this point Podman needs to configure the network inside of the network namespace.





**NOTE** Linux TAP devices create a user space network bridge. In user space TAP Devices can simulate network devices inside of a network namespace. Processes within the namespace interact with the network device. Packets read/written from the network device are routed via the TUN/TAP device to the user space program, `slirp4netns`.

Now that the storage and network is configured, Podman is ready to finally start the container process.

#### 6.2.4 Starting the container monitor - common

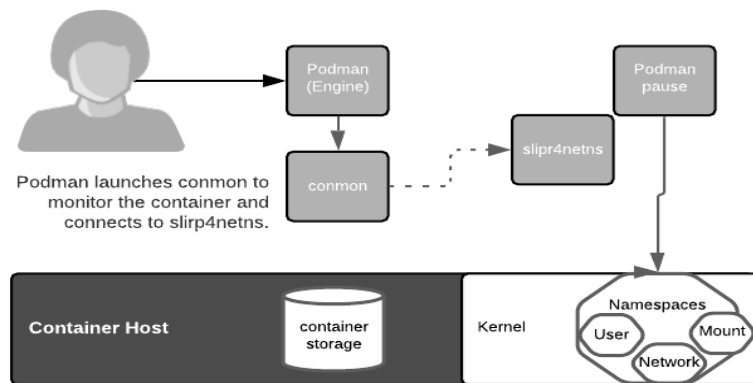


Figure 6.7 Podman launches the container monitor which launches the OCI Runtime.

Podman now executes `common` (container monitor) for the container, telling it to use its configured OCI runtime, usually, `crun` or `runc`. It also executes the `podman container cleanup $CTRID` command when the container exits.

`Common` is described in chapter 4 section 1.

## 6.2.5 Launching the OCI runtime

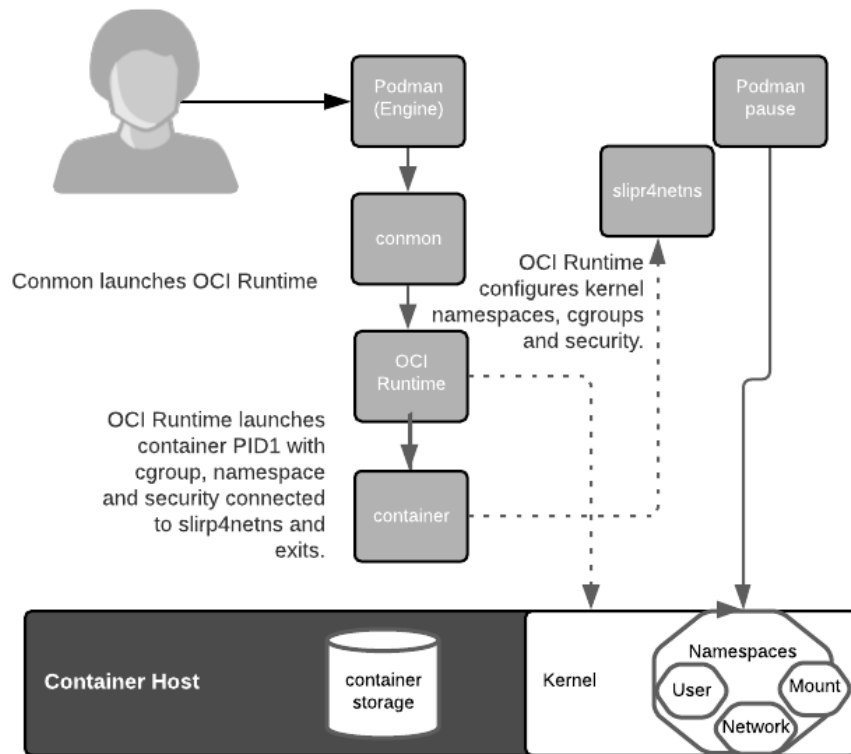
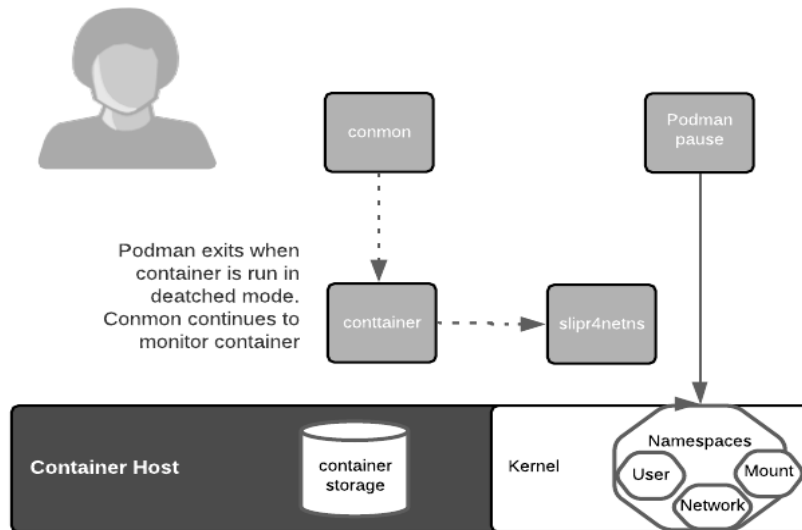


Figure 6.8 common launches the OCI Runtime which configures the kernel

The OCI runtime reads the OCI spec file and configures the kernel to run the container. OCI Runtimes do the following:

1. Set up the additional namespaces for the container.
  2. Configure cgroups V2 (cgroups V1 is not supported for rootless containers).
  3. Set up the SELinux label for running the container.
  4. Load the `/usr/share/containers/seccomp.json` seccomp rules into the kernel.
  5. Set the environment variables for the container.
  6. Bind mounts any volumes onto the paths in the rootfs.
  7. Switch the current `/` to the rootfs `/`.
  8. Fork the container process.
  9. Execute any OCI hook programs, passing them the rootfs as well as the container's PID 1.
  10. Execute the command specified by the image.
  11. Exit the OCI runtime, leaving common to monitor the container.
- And finally, common reports the success back to Podman.



**Figure 6.9 Podman and OCI Runtime exit leaving container running with common monitoring it, and slirp4netns providing the network**

The Podman command now exits because it ran in `--detach (-d)` mode.

```
$ podman run -d -p 8080:8080 --name myapp registry.access.redhat.com/ubi8/httpd-24
```

**NOTE** If later you want Podman to interact with the detached container you use the `podman attach` command which connects to the common socket. Common allows Podman to interact with the container process through the `STDIN`, `STDOUT`, and `STDERR` file descriptors which common has been monitoring.

### 6.2.6 The containerized application runs until completion

The application process can exit on its own or you can stop the container by executing the `podman stop` command.

```
$ podman stop myapp
```

When the container process exits, the kernel sends a `SIGCHLD` to the common process. In turn, common does the following:

1. Records the container's exit code.
2. Closes the container's logfile.
3. Closes the Podman command's `STDOUT/STDERR`.
4. Executes the podman container cleanup `$CTRID` command.
5. Exits itself.

The `podman container cleanup` command takes down the `slirp4netns` network and unmounts all of the container mount points. If you specify the `--rm` option, the container is entirely removed - layers are removed from `containers/storage`, and the container definition removed from the DB.

## 6.3 Summary

- Running rootless containers is more secure than running rootful containers
- The user namespace gives ordinary users the ability to manipulate more than one UID and is key to running containers
- The mount namespace allows Podman to mount file systems within the user namespace
- Podman uses `slirp4netns` for providing network access to containers
- Podman launches common process to monitor the container

# *Appendix A*

## *Podman-related container tools*

This appendix describes the three tools that use containers/storage and containers/image libraries.

These tools address following functionalities:

1. Moving container images between different container registries and storage
2. Building container images
3. Testing, developing, and running containers in production on a single node
4. Running containers in production at scale

As the original creator of Podman, I recognized the need for specialized tools, each performing specific functionality rather than a one-size-fits-all monolithic solution.

From a security perspective, each of these four categories requires different security constraints. Containers running in production need to be run in a more secure environment than ones running in development and testing. Moving container images between registries requires no privileged access to the host you are running the command on, only remote access to the registries. You get the least secure system with a monolithic daemon. If my containers need more access during builds, then in production they get the same access as during builds.

Another critical problem with a monolithic daemon is that it prevents experimentation with the tools and doesn't allow them to go their own way. One example of this is when we proposed a change to the Docker daemon to allow users to pull different types of OCI content off of container registries, and this change was denied as it had little to do with docker containers.

Similarly when the monolithic daemon is modified for one product, it can negatively impact features of another one which is using this daemon. It could cause performance degradation or down right breakage. This happened when Kubernetes was being developed, since it relied on the Docker daemon as the container engine. But since Docker is monolithic and being developed for many other projects a lot of its changes affected Kubernetes, leading to instability of it. It was obvious that Kubernetes needed a dedicated container engine for its workloads and in December 2020 it was announced that Kubernetes will eventually use the newly developed standard the container runtime interface, CRI

(<https://kubernetes.io/blog/2016/12/container-runtime-interface-cri-in-kubernetes/>) to improve interaction between orchestrators and different container runtimes.

I wrote another coloring book (<https://red.ht/3gfVIHF>) illustrated by Máirín Duffy (@marin) describing the container tools talked about in this appendix based on superheroes.

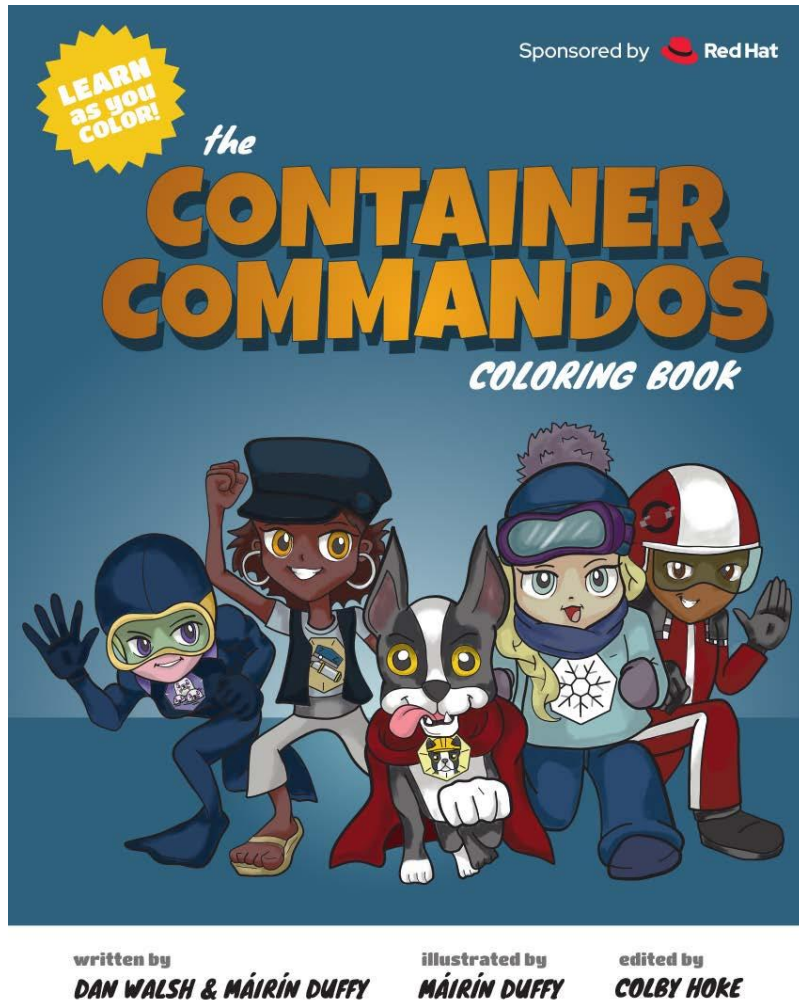


Figure A.1 The container Coloring Book ([https://docs.fedoraproject.org/en-US/fedora-silverblue/\\_attachments/container-commandos.pdf](https://docs.fedoraproject.org/en-US/fedora-silverblue/_attachments/container-commandos.pdf))

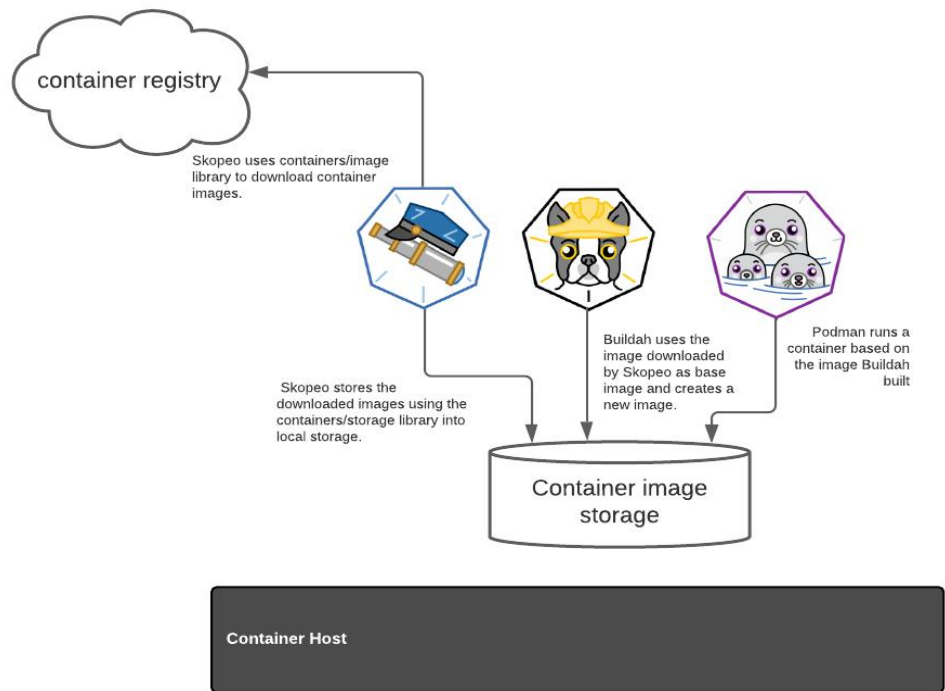
Finally, sometimes there are conflicting interests or release schedules in play. Having separate, independent tools allows releases to be deployed independently from all the others at their own pace to guarantee new features to their customers.

Four projects were created for the distinct functions described in table A.1.

**Table A.1 Primary container tools based on containers/storage and containers/image.**

Command	Description
Skopeo	Performs various operations on container images and image repositories. ( <a href="https://github.com/containers/skopeo">https://github.com/containers/skopeo</a> )
Buildah	Facilitates a wide range of operations on container images. ( <a href="https://github.com/containers/buildah">https://github.com/containers/buildah</a> )
Podman	All-in-one management tool for pods, containers, and images. ( <a href="https://github.com/containers/podman">https://github.com/containers/podman</a> )
CRI-O	OCI-based implementation of the Kubernetes Container Runtime Interface. ( <a href="https://github.com/cri-o/cri-o">https://github.com/cri-o/cri-o</a> )

As you have already learned a great deal about Podman, you know now why it is included in this list. Podman is an excellent tool for understanding and developing containers as well as pods and images. It encapsulates everything that Docker CLI does but without locking everything under one central daemon. Because Podman works without a daemon and uses the operating system for sharing data, other tools can work with the same data stores and libraries.



**Figure A.2** Skopeo, Buildah and Podman work together by sharing the same containers/storage images and containers/image library for pulling and pushing images.

The rest of this appendix describes the rest of the tools, starting with Skopeo.

## A.1 Skopeo

While using container engines like Docker or Podman, if you want to inspect a container image in a registry, you are required to pull this image from the registry to your local storage. Only then can you examine it. The issue is that this image can be huge, and after inspecting it, you might realize that it wasn't what you expected and you wasted time pulling it. Because the protocol used to pull the image and inspect it is just a web protocol, a simple tool, Skopeo, was created to just pull the image's detailed information and display it on the screen. Skopeo is the Greek word for *remote viewing*.





Execute the following `skopeo inspect` command to examine the image detailed information in JSON form:

```
$ skopeo inspect docker://quay.io/rhatdan/myimage
{
  "Name": "quay.io/rhatdan/myimage",
  "Digest":
    "sha256:fe798c1576dc7b70d7de3b3ab7c72cd22300b061921f052279d88729708092d8",
  "RepoTags": [
    "Latest",
    "1.0"
  ],
  ...
}
```

Skopeo was extended to also copy images off of registries. Eventually Skopeo became the tool for copying images between different types of storage (transports). These types of storage became the transports defined in table A.2.

**Table A.2 Podman supported transports**

Transport	Description
container registry (docker)	Default transport. References a container image stored in a remote container image registry. Container registry is a place for storing and sharing container images. For example, docker.io, quay.io.
oci	References a container image, compliant with the Open Container Image Layout Specification. The manifest and layer tarballs as individual files are located in the local directory.
dir	References a container image, compliant with the Docker image layout. It is very similar to the `oci` transport but stores the files using the legacy "docker" format. As a non-standardized format, primarily useful for debugging or noninvasive container inspection.
docker-archive	References a container image in Docker image layout which is packed into a TAR archive.
oci-archive	References an image compliant with the Open Container Image Layout Specification which is packed into a TAR archive. It is very similar to the `docker-archive` transport, but stores an image in OCI Format.
docker-daemon	References an image stored in the Docker daemon's internal storage. Since the Docker daemon requires root privileges, Podman has to be run by root user.
container-storage	References an image located in a local container storage. It is not a transport, but more of a mechanism for storing images. It can be used to convert other transports into container-storage. Podman defaults to using container-storage for local images.

Other container engines and tools wanted to use the functionality developed in Skopeo to copy images, so Skopeo was split in two. The command line, Skopeo, and the underlying library `containers/image`. Splitting functionality into a separate library made it possible to build other container tools, including Podman.

The `skopeo copy` command is very popular for copying images between different types of container storage. One difference compared to Podman and Buildah, as you'll see in section E2, is that Skopeo forces users to specify the transport for the source and destination. Podman and Buildah default to using the `docker` or `containers-storage` transport, depending on the context and command. In the following example, you are copying an image from a container

registry using the docker transport and storing the image locally using the container-storage transport.

```
$ skopeo copy docker://quay.io/rhatdan/myimage containers-storage:quay.io/rhatdan/myimage
Getting image source signatures
Copying blob dfd8c625d022 done
Copying blob 68e8857e6dcb done
Copying blob e21480a19686 done
Copying blob fbfcc23454c6 done
Copying blob 3f412c5136dd done
Copying config 2c7e43d880 done
Writing manifest to image destination
Storing signatures
```

Another command many Skopeo users use is `skopeo sync`, which lets you synchronize images between container registries and local storage.

Skopeo is mainly used for infrastructure projects to help provision multiple container registries, for example copying images from a public registry to a private one. Table A.2 describes the most popular commands used with skopeo.

**Table A.2 Primary Skopeo commands and their description.**

Command	Description
<code>skopeo copy</code>	Copy an image (manifest, filesystem layers, signatures) from one location to another.
<code>skopeo delete</code>	Mark the image name for later deletion by the registry's garbage collector.
<code>skopeo inspect</code>	Return low-level information about image-name in a registry.
<code>skopeo list-tags</code>	List tags in the transport-specific image repository.
<code>skopeo login</code>	Login to a container registry (Same as Podman Login).
<code>skopeo logout</code>	Logout of a container registry (Same as Podman Logout).
<code>skopeo manifest digest</code>	Compute a manifest digest for a manifest-file and write it to standard output.
<code>skopeo sync</code>	Synchronize images between container registries and local directories.

One of the first tools to take advantage of the containers/image library was Buildah.

## A.2 Buildah

As you learned in section chapter 1, section 1.1.2, creating a container image means creating a directory on disk and adding content to it to make it look like the root, `/`, directory on a Linux machine, called a rootfs. Originally the only way to do this was with `docker build` using a `Dockerfile`. While Dockerfiles and Containerfiles are excellent ways to create recipes for your container images, a low-level building block tool which allowed other ways to build container images was needed. One which allows breaking the image build process into individual commands, letting you to use other more powerful scripting tools and languages other than Containerfile to build images. We created a tool called `Buildah` (<https://buildah.io>) to serve this purpose.

Buildah was designed to be that simple tool to build container images. It's built on top of the container/storage and container/image libraries, just like Podman and Skopeo. It has a lot of similar functionality to Podman. You can pull images, push images, commit images, even run containers on images. What mainly differs Podman and Buildah is the underlying concept of a container. Podman container is a long-lived one, "running" container, while Buildah container is just a temporary one, "working" container, which will be used to create an OCI image.



**NOTE** Buildah is a Linux only tool, not available on Mac or Windows. However Podman embeds Buildah in the `podman build` command. Podman on Mac and Windows uses the buildah code on the server side, allowing those platforms to build using Containerfiles and Dockerfiles. See appendix E and F for more information.

Buildah was designed to take the steps that were defined in a Dockerfile and make them available at the command line. Buildah wanted to simplify building a container image by allowing you to use all of the tools available within the OS to populate the image. You can add data to this directory via standard Linux tools, like `cp`, `make`, `yum install`, etc. Then commit the rootfs into a tarball, add some JSON to describe what the creator of the image wanted the image to do, and finally push this to a container registry. Basically, Buildah breaks down the steps you learned about in a Containerfile into individual commands that you can execute from a shell.

**NOTE** The name *Buildah* is a play on the way I pronounce *builder*. If you ever heard me speak, you'd notice I have a strong Boston accent. When the core team asked what I wanted to call the tool, I said I don't care, just call it *builder*. And they heard *Buildah*. :^)

The first step when building a new container image is to pull a base image. In a Containerfile this is done with the `FROM` instruction.

### A.2.1 Creating a working container from a base image

The first command to look at is `buildah from`. It is equivalent to the Containerfiles `FROM` instruction. When executing `buildah from IMAGE`, it pulls the specified image from the container registry, saves it in a local container storage and creates a working container based on this image. As mentioned previously, this container is similar to a Podman container, except that it exists temporarily only to become a container image. In the following example, a working container is created based on an `ubi8-init` image:

```
$ buildah from ubi8-init
Resolved "ubi8-init" as an alias (/etc/containers/registries.conf.d/000-shortnames.conf)
Trying to pull registry.access.redhat.com/ubi8-init:latest... #A
Getting image source signatures
Checking if image destination supports signatures
Copying blob adffa6963146 done
Copying blob 29250971c1d2 done
Copying blob 26f1167feaf7 done
Copying config 4b85030f92 done
Writing manifest to image destination
Storing signatures
ubi8-init-working-container #B
```

#A pulls image from container registry

#B outputs new container name

Notice that the `buildah from` output looks the same as the `podman pull` output, except for the last line, which outputs the container name, `ubi8-init-working-container`. If you run the `buildah from` command again, you get a second container name.

```
$ buildah from ubi8-init
ubi8-init-working-container-1
```

Buildah keeps track of its containers and generates each one by incrementing a counter. Of course you can override the container name with the `--name` option.

Next, you will add content to this container image.

### A.2.2 Adding data to a working container

Buildah has two commands `buildah copy` and `buildah add` for copying the contents of a file, URL, or directory into the container's working directory. They map to the same functionality as the Containerfile's `COPY` and `ADD` instructions.

**NOTE** It is somewhat confusing to have two commands that do almost the same thing. In most cases, I recommend that you just use `buildah copy` and `COPY` inside of a Containerfile. The main difference between the two is that `copy` only copies local files and directories off of the host into the container image. The `add` command supports the use of URLs to pull remote content and insert it into your container. The `add` command also supports taking tar and zip files and expanding them when copied into the container image.

The syntax of the `buildah copy` command requires you to specify the name of the container previously created by the `buildah from` command, followed by the source and optionally, destination. If destination is not provided, source data will be copied into the container's working directory. Destination directory will be created if it doesn't exist yet.

Following example copies local file `html/index.html` (created previously in Section 3.1 of this book) into the `/var/lib/www/html` directory in the container:

```
$ buildah copy ubi8-init-working-container html/index.html /var/lib/www/html/
```

If you would like to use more advanced tools like package managers to add content to your containers, Buildah supports running commands inside the containers.

### A.2.3 Running commands in a working container

To run a command inside of the working container, you need to execute `buildah run`. Under the hood, this command works exactly the same as `RUN` instruction - starts a new container on top of the current one, executes specified command and commits the result back to the working container. The syntax of `buildah run` requires you to specify the name of the working container followed by the command. In the following example you install the `httpd` service within the container:

```
$ buildah run ubi8-init-working-container dnf -y install httpd
Updating Subscription Management repositories.
Unable to read consumer identity
This system is not registered with an entitlement server. You can use subscription-manager
to register.
...
Complete!
```

To make sure you will have a running web server once running container is created, next command enables the Apache HTTP Server service:

```
$ buildah run ubi8-init-working-container systemctl enable httpd.service
Created symlink /etc/systemd/system/multi-user.target.wants/httpd.service →
/usr/lib/systemd/system/httpd.service.
```

Table A.3 shows the relationship between Containerfile instructions and Buildah commands.

**Table A.3 Containerfile instructions mapped to Buildah commands**

Instruction	Command	Description
ADD	<code>buildah add</code>	Add the contents of a file, URL, or a directory to the container.
COPY	<code>buildah copy</code>	Copies the contents of a file, URL, or directory into a container's working directory.
FROM	<code>buildah from</code>	Creates a new working container, either from scratch or using a specified image as a starting point.
RUN	<code>buildah run</code>	Run a command inside of the container.

### A.2.4 Adding content to a working container directly from the host

Up until now, you've seen how Buildah can perform the same commands that you execute within a Containerfile, but one of Buildah's goals is to expose the container image rootfs directly to the host. This allows you to use commands available on your host machine to add content to the container image without requiring the commands to be present inside of the container image.

The `buildah mount` command allows you to mount a working container's root filesystem directly on your system and then use tools like `cp`, `make`, `dnf`, or even an editor to manipulate the contents of the container's rootfs.

If you run `buildah` as root, you can simply execute the `buildah mount` command. But in rootless mode, this isn't allowed. Recall in chapter 2, section 2.2.10, where you learned about the `podman mount` command, you must first enter the user namespace. Similarly, `buildah unshare` command creates a shell running in the user namespace. Once you are in the user namespace, you can mount the container. In the following example, using what you have learned so far, you are going to use your host's command to add content to the container:

```
$ buildah unshare
# mnt=$(buildah mount ubi8-init-working-container)
# echo $mnt
/home/dwalsh/.local/share/containers/storage/overlay/133e1728eac26589b07984e3bdf31b5e318159
940c866d9e0493a1d08e1d2f6a/merged
# grep dwalsh /etc/passwd >> $mnt/etc/passwd
# exit
```

Now you can check if your changes were actually applied inside working container:

```
$ buildah run ubi8-init-working-container grep dwalsh /etc/passwd
dwalsh:x:3267:3267:Daniel J Walsh:/home/dwalsh:/bin/bash
```

After you are done with populating the content of the working container, it's time to specify other instructions from the Containerfile, which describe your intentions as the container image creator.

## A.2.5 Configuring a working container

You probably noticed in table A.3 that there are a lot of Containerfile instructions that are missing. Containerfile instructions like LABEL, EXPOSE, WORKDIR, CMD, and ENTRYPOINT are used to populate the OCI Image Specification.

Now, using the `buildah config` command, you can add a port to expose (EXPOSE) and mark a location inside container rootfs as a volume (VOLUME), which will be used as website root directory:

```
$ buildah config --port=80 --volume=/var/lib/www/html ubi8-init-working-container
```

You can inspect the corresponding OCI Image Specification fields using the `buildah inspect` command:

```
$ buildah inspect --format '{{ .OCIv1.Config.ExposedPorts }}' {{ .OCIv1.Config.Volumes }}'
ubi8-init-working-container
map[80:{}] map[/var/lib/www/html:{}]
```

Table A.4 shows the relationship between Containerfile instructions and Buildah config options. You can also refer to Table 2.5 for the extra information on these instructions.



**Table A.4 Containerfile instructions mapped to Buildah config options.**

Instruction	Command	Description
MAINTAINER	<code>--author</code>	Sets contact information of the image author.
CMD	<code>--cmd</code>	Sets a default command to run within a container.
ENTRYPOINT	<code>--entrypoint</code>	Sets a command for a container that will run as an executable. .
ENV	<code>--env</code>	Sets the environment variable for all subsequent instructions.
HEALTHCHECK	<code>--healthcheck</code>	Specifies a command to check if a container is still running.
LABEL	<code>--label</code>	Adds a key-value metadata.
ONBUILD	<code>--onbuild</code>	Sets a command to be run when the image is used as the base for another image.
EXPOSE	<code>--port</code>	Specifies a port that container will listen on at a runtime.
STOPSIGNAL	<code>--stop-signal</code>	Sets the stop signal to be sent when the container is stopped.
USER	<code>--user</code>	Sets the user to be used when running the container and for all subsequent RUN, CMD and ENTRYPOINT instructions.
VOLUME	<code>--volume</code>	Adds a mount point and marks it as a volume for external data.
WORKDIR	<code>--workingdir</code>	Sets the working directory for all subsequent RUN, CMD, ENTRYPOINT, COPY and ADD instructions.

Once you have completed adding content to the Buildah container image and adding configuration to the OCI Image Specification, you need to create an image from the working container.

### A.2.6 Creating an image from a working container

The working container you've been building so far can be used to create the OCI-compliant image using the `buildah commit` command. This command works in the same way as the `podman commit` command you learned about in chapter 2, section 2.1.9. Inputs for this command are the working container name and an optional image tag, if tag is not specified, the image will have no name..

```
$ buildah commit ubi8-init-working-container quay.io/rhatdan/myimage2
Getting image source signatures
Copying blob 352ba846236b skipped: already exists
Copying blob 3ba8c926eef9 skipped: already exists
Copying blob 421971707f97 skipped: already exists
Copying blob 9ff25f020d5a done
Copying config 5e47dbd9b7 done
Writing manifest to image destination
Storing signatures
5e47dbd9b7b7a43dd29f3e8a477cce355e42c019bb63626c0a8feffae56fcbf9
```

You can see the image using `buildah images`.

```
$ buildah images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
quay.io/rhatdan/myimage2	latest	5e47dbd9b7b7	2 minutes ago	293 MB
registry.access.redhat.com/ubi8-init	latest	4b85030f924b	5 weeks ago	253 MB

Because Podman and Buildah share the same container image storage, you can see the same images with `podman images`.

```
$ podman images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
quay.io/rhatdan/myimage2	latest	5e47dbd9b7b7	4 minutes ago	293 MB
registry.access.redhat.com/ubi8-init	latest	4b85030f924b	5 weeks ago	253 MB

You can even run a Podman container on the image.

```
$ podman run quay.io/rhatdan/myimage2 grep dwalsh /etc/passwd
dwalsh:x:3267:3267:Daniel J Walsh:/home/dwalsh:/bin/bash
```

## A.2.7 Pushing an image to a container registry

Similarly to Podman, Buildah has commands `buildah login` and `buildah push`, which allow you to push images to container registries, shown in the following example.

**NOTE** You can also use `podman login` and `podman push` or even `skopeo login` and `skopeo copy` to accomplish the same task.

```
$ buildah login quay.io
Username: rhatdan
Password:
Login Succeeded!
$ buildah push quay.io/rhatdan/myimage2
Getting image source signatures
Copying blob 3ba8c926eef9 done
Copying blob 421971707f97 done
Copying blob 9ff25f020d5a done
Copying blob 352ba846236b done
Copying config 5e47dbd9b7 done
Writing manifest to image destination
Copying config 5e47dbd9b7 done
Writing manifest to image destination
Storing signatures
```

Congratulations, you have successfully built an OCI-compliant container image manually by using simple shell commands rather than using a Containerfile.

Additionally, if you want to create an image using an existing Containerfile or Dockerfile, you can use the `buildah build` command.

### A.2.8 Building an image from Containerfiles

You can use the `buildah build` command to build an OCI-compliant image from a Containerfile or a Dockerfile. Buildah includes a parser that understands the Containerfile format and can perform all tasks using previously described commands automatically..

Let's use the Containerfile from section 2.3.2, chapter 2:

```
$ cat myapp/Containerfile
FROM ubi8/httpd-24
COPY index.html /var/www/html/index.html
```

You can build your container image using this Containerfile by executing the following command:

```
$ buildah build ./myapp
STEP 1/2: FROM ubi8/httpd-24
Resolved "ubi8/httpd-24" as an alias (/home/dwalsh/.cache/containers/short-name-
aliases.conf)
Trying to pull registry.access.redhat.com/ubi8/httpd-24:latest
...
Getting image source signatures
Checking if image destination supports signatures
Copying blob adffa6963146 skipped: already exists
...
STEP 2/2: COPY html/index.html /var/www/html/index.html
COMMIT
Getting image source signatures
Copying blob 352ba846236b skipped: already exists
...
bbfcf76c994c738f8496c1f274bd009ddbc960334b59a74953691fff00442417
```

You probably notice that this output matches precisely the output of the `podman build` command. This is because the `podman build` command uses Buildah.

### A.2.9 Buildah as a library

Buildah was designed to not only be used as a command-line tool but also to be a Golang-based library. Buildah is being used in a few different tools, such as Podman and the OpenShift image builder. Buildah allows these tools to internally build OCI images. Every time you do a `podman build`, you are executing the Buildah library code.

Having learned how to build container images using Buildah, copy images between container storages using Skopeo, how to manage and run containers on the host using Podman, let's talk about how all these tools are used in the Kubernetes ecosystem.

### A.3 CRI-O: Container Runtime Interface for OCI Containers

When Kubernetes was being developed, it used Docker API internally to run containers. Kubernetes relied on features of Docker that changed from release to release, sometimes breaking Kubernetes. At the same time, CoreOS wanted their alternative container engine called RKT (<https://github.com/rkt/rkt>) to work with Kubernetes. Kubernetes developers decided then to split out the Docker functionality and use a new API called the Container Runtime Interface, CRI. (<https://kubernetes.io/blog/2016/12/container-runtime-interface-cri-in-kubernetes>) This interface allows Kubernetes to use additional container engines besides Docker.

When Kubernetes wants to pull a container image, it calls out to a remote socket via the CRI and asks the listener to pull an OCI image for it. When it wants to launch a Pod/Container, it calls out the socket and asks it to launch the container.

**NOTE** CoreOS was eventually acquired by Red Hat, and the RKT project has ended. Kubernetes has deprecated Docker as a container runtime.

Red Hat saw the CRI as an opportunity to develop a new container engine, which they ended up calling the **C**ontainer **R**untime **I**nterface for **O**CI containers, CRI-O. (<https://cri-o.io/>).



CRI-O is based on the same containers/storage and containers/image libraries as Skopeo, Buildah and Podman, and can be used in conjunction with these tools. CRI-O's primary objective is to replace the Docker service as the container engine for Kubernetes.

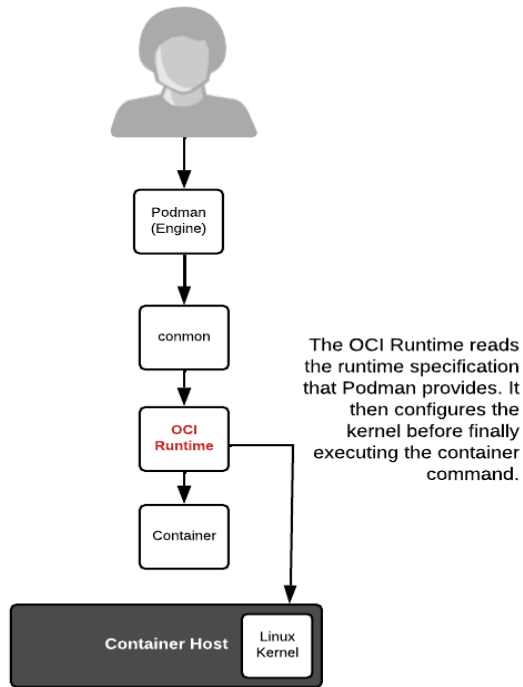
CRI-O is tied to Kubernetes releases. When a new version of Kubernetes is released, the version numbers are synchronized. CRI-O is optimized for Kubernetes workloads, engineers working on it understand what Kubernetes is trying to do and are making sure CRI-O does it in the most efficient way possible. Since CRI-O has no other users, Kubernetes doesn't have to worry about breaking changes in CRI-O.

**NOTE** CRI-O is the core technology used with Red Hat's OpenShift Kubernetes based product. OpenShift uses Podman to install and configure CRI-O before Kubernetes starts running. OpenShift image builder embeds Buildah functionality to allow users to build images within their OpenShift clusters.

# ***Appendix B***

## ***OCI runtimes***

This appendix describes the primary OCI runtimes used with container engines like Podman. As discussed in chapter 1, the OCI runtime (<https://opencontainers.org>) is the executable launched by container engines, including Podman, used to configure the Linux kernel and subsystems to run the kernel, it's last step is to launch the container. The OCI runtime reads the OCI runtime-specification JSON file and then configures the namespaces, security controls, and cgroups, and eventually starts the container process



**Figure B.1.** Podman executes the OCI Runtime to launch the container.

In this appendix you'll learn the four main OCI Runtimes in use.

The `--runtime` option allows you to switch between different OCI runtimes. In the next example you run the same container command twice, each time with a different runtime. In the first command you run the container with a runtime, `crun`, that is defined in the `containers.conf`, so you don't need to specify the path to the runtime.

```
$ podman --runtime crun run --rm ubi8 echo hi #A
hi
```

**#A** The `--runtime` option tells Podman to use the `crun` OCI Runtime, rather than the default.

The default runtime is defined under the `[containers]` table in the `containers.conf` file on the Linux machine.

```
$ grep -iA 3 "Default OCI Runtime" /usr/share/containers/containers.conf
# Default OCI runtime
#
#runtime = "crun" #A
```

#A Podman defaults to crun on most systems, on some older distributions like Red Hat Enterprise Linux, Podman defaults to runc.

In the second example you use the full path of the OCI runtime, `/usr/bin/runc`.

```
$ podman --runtime /usr/bin/runc run -rm ubi8 echo hi
hi
```

If you want to permanently change the default OCI runtime, you can set the `runtimes` option in the `[engine]` table in the `containers.conf` file in your home directory.

```
$ cat > ~/.config/containers/containers.conf << EOF
[engine]
runtime="runc"
EOF
$ podman --help | grep -- runc
--runtime string      Path to the OCI-compatible binary used to run containers.
                       (default "runc")`
```

**NOTE** The `--runtime` option is only available on Linux. Podman `--remote` and therefore Podman on a MAC and Windows does not support the `--runtime` option, so you need to set the `containers.conf` file on the server side.

See the `podman(1)` man page for more information. `man podman`.

OCI runtimes are continuously being developed and experimented with. You can expect innovation to happen in this space going forward. The first container runtime developed and the de facto standard is `runc`.

## B.1 runc

Runc is the original OCI runtime. (<https://github.com/opencontainers/runc>). When the OCI originally formed, Docker donated the `runc` to the OCI to serve as the default implementation of an OCI runtime. The OCI continues to support and develop `runc`. It is written in Golang, and also includes the `libcontainer` library, which is used in many container engines and Kubernetes.

On the `runc` website it states that `runc`, and all of the OCI runtimes, is a low-level tool not designed to be used directly by the end user. It is recommended to be launched by container engines like Podman or Docker.

Recall the container engine's job is to pull the container images to the host, configure and mount the root file system (rootfs) to be used within the container, and finally to write the OCI Runtime JSON file before launching the OCI Runtime.

The OCI runtime specification describes only the content of the JSON file used by OCI runtimes. Because every OCI engine supports the `runc` command line, the other OCI runtimes adopted the same CLI commands and options. This makes it easier for one runtime to replace the other when launched by the container engine. Table B.1, shows the commands supported by `runc` and therefore all OCI runtimes.

**Table B.1: runc commands**

Command	Description
<code>checkpoint</code>	Checkpoint a running container
<code>create</code>	Creates a container
<code>delete</code>	Deletes any resources held by the container often used with detached containers.
<code>events</code>	Displays container events such as OOM notifications, CPU, memory, and IO usage statistics.
<code>init</code>	Initializes the namespaces and launches the process.
<code>kill</code>	Sends the specified signal (default: SIGTERM) to the container's init process.
<code>list</code>	Lists containers started by runc with the given root.
<code>pause</code>	Suspends all processes inside the container.
<code>ps</code>	Displays the processes running inside a container.
<code>restore</code>	Restores a container from a previous checkpoint.
<code>resume</code>	Resumes all processes that have been previously paused.
<code>run</code>	Creates and runs a container.
<code>spec</code>	Creates a new specification file.
<code>start</code>	Executes the user defined process in a created container
<code>state</code>	Outputs the state of a container.
<code>update</code>	Updates container resource constraints.

runc continues to be developed and has a very active community. The problem with runc is that it is written in Golang. Golang was not designed to be a small, often-executed application that needs to start quickly and fork/exec a command and exit quickly. Fork/Exec is a heavy operation in Golang, and although runc attempts to work around this, it ultimately sacrifices "a bit" of performance. The "a bit" can accumulate over time though, so runc performs much better at scale.



## B.2 crun

runc, being written in Golang, is a very heavy executable, 12 megabytes in size. Golang is a great language, but doesn't take advantage of shared libraries. Golang executables take up considerably more memory because of this. The size of runc causes it to be somewhat slower loading during container start. Another problem with Golang, is that it does not support the fork/exec model all that well. It is slower when compared to fork/exec in other languages, e.g. c. This slowness is more important when you are starting and stopping hundreds or thousands of containers, for example on a Kubernetes cluster. Container engines like Podman, also written in Go, generally run for a much longer time so the startup time is not as important. OCI runtimes like runc execute for a very short time and exit quickly.

Giuseppe Scrivano, a contributor to runc and Podman, understood these deficiencies in runc, and wanted to write a compatible OCI runtime in the c language. He created a very lightweight OCI runtime called `crun`.

`crun` describes itself as a *fast and lightweight OCI runtime*. `crun` supports all of the same commands and options as runc. `crun` executable is many times smaller than runc. Execute the `du -s` command to compare the sizes:

```
$ du -s /usr/bin/runc /usr/bin/crun
14640  /usr/bin/runc
392    /usr/bin/crun
```

`crun`, being written in c, supports fork and exec much better than Golang and therefore is much quicker when launching a container.

This also makes it plugin easily to other libraries on the system, and there is some experimentation to use `crun` as a library for processing the OCI Runtime JSON file and launching different types of containers. For example WASM and Windows containers on linux. Also `crun` has potential for launching KVM separated containers based on libkrun.

`crun` is now the default OCI Runtime used by Podman in Fedora and in Red Hat Enterprise Linux 9. runc continues to be supported and is the default OCI runtime in Red Hat Enterprise Linux 8.

`crun` and runc are the two primary OCI Runtimes for managing traditional containers that use Namespace separation.

Both these projects work fairly closely together. When bugs or issues are found in either OCI runtime, they are quickly fixed in both.

See the `crun(1)` man page for more information. `man crun`.

OCI Runtimes is also written to use VM separation, with the primary example of this being Kata Containers.

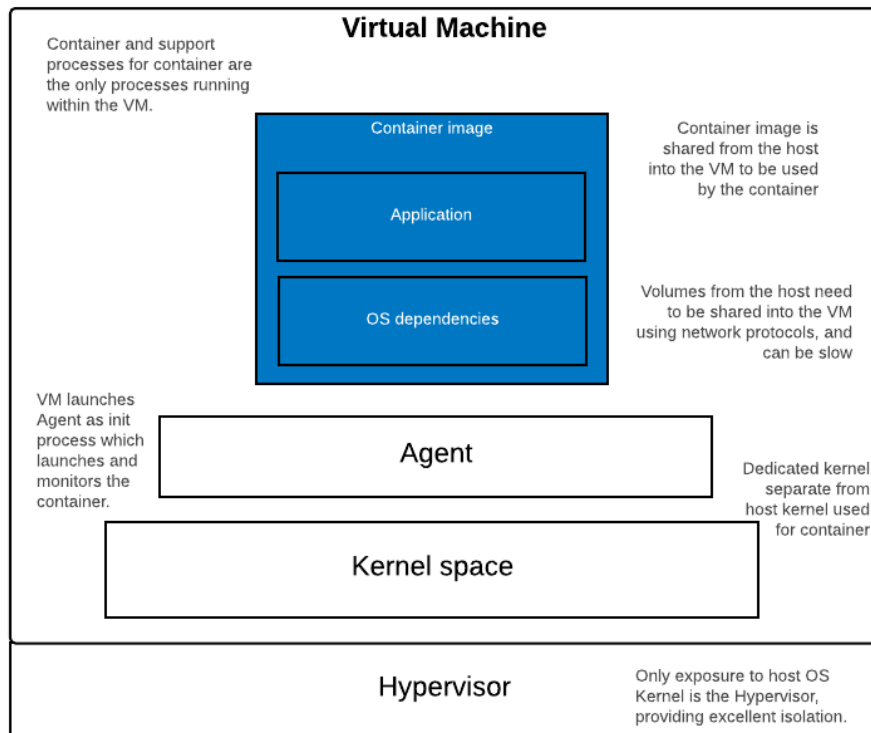
## B.3 kata

The Kata Container project (<https://katacontainers.io>) advertises itself as the following:



*The speed of containers, the security of VMs*

*Kata Containers is an open source container runtime, building lightweight virtual machines that seamlessly plug into the container's ecosystem.*



**Figure B.2** Kata Containers launches a lightweight Virtual Machine, which only runs the container.

Kata Containers use VM technology for launching each container. It is very different from launching a VM and running Podman within it. A standard VM has an init system which launches all sorts of services, like logging systems, cron, etc. Whereas a Kata container launches a micro OS, which runs only the container and its support services. As its only purpose is to launch the container, when the container exits, this VM goes away.

I believe that running containers within VM/hypervisor separation gives you better security separation compared to traditional container separation, where containers communicate directly with the host kernel. A VM-separated container has to first break out of containment inside of the VM, then figure a way to break out of the Hypervisor only then face the attacking of the host kernel.

While VM-separated containers are more secure, this does come with some downsides. There is a decent amount of overhead in starting a Kata container, configuring the hypervisor, launching the kernel and other processes within the VM, and then finally the container. VM's memory, CPU, and so on have to be preallocated and are difficult to change. Running Kata within a VM in the cloud is often not allowed, or at least more expensive, because most of the cloud vendors frown upon nested virtualization.

Finally and most importantly VM-separated containers by their very nature have difficulties sharing content with other containers and the host operating system. The biggest issue is with volumes.

While sharing content with the host machine in traditional containers is just a bind mount, in VM-separate containers, bind mounts do not work. Since the processes on the host and in the container are running with two different kernels, you need a network protocol to share content. Kata containers originally used NFS and Plan 9 networked file systems. Reading/writing data over these networked file systems is going to be considerably slower than native file system reads and writes you get with a bind mount.

Virtiofs is a new file system that has the properties of a network file system, but lets virtual machines access files on the host. It is able to show big improvements in speed over the network based file systems, while still remaining under heavy development.

Kata containers have two different ways to be launched. Kata has traditionally OCI command line, kata-runtime, based on the `runc` command supported by Podman. You can see the paths defined in `containers.conf`, on the Linux machine, by searching for `#kata`.

```
$ grep -A 9 '^#kata' /usr/share/containers/containers.conf
#kata = [
#  "/usr/bin/kata-runtime",
#  "/usr/sbin/kata-runtime",
#  "/usr/local/bin/kata-runtime",
#  "/usr/local/sbin/kata-runtime",
#  "/sbin/kata-runtime",
#  "/bin/kata-runtime",
#  "/usr/bin/kata-qemu",
#  "/usr/bin/kata-fc",
#]
```

Bottom line on Kata containers is that you get better security with a performance overhead. Depending on the workload you can choose the best OCI runtime for your application.

## B.4 gVisor



The last OCI Runtime I cover in this appendix is gVisor. (<https://gvisor.dev/>)

The gVisor website advertises itself as an **application kernel** for **containers** that provides efficient defense-in-depth anywhere.

gVisor includes an OCI runtime called `runsc` and works with Podman and other container engines. The gVisor project calls itself an application kernel, written in Golang, that implements a substantial portion of the Linux system call interface. It provides an additional layer of isolation between running applications and the host operating system. Google engineering wrote the original versions of gVisor and claim that the bulk of the containers that Google Cloud run use the gVisor OCI runtime.

gVisor is somewhat similar to VM-isolated containers, in that gVisor is intercepting almost all system calls from within the container and then processing them. gVisor describes itself as an application kernel for containers written in Golang, limiting the access to the host kernel. At the same time, it does not have the same issue of a nested virtualization as Kata.

However, gVisor introduces performance penalty with additional CPU cycles and higher memory usage. This may introduce increased latency or reduced throughput or both. gVisor is also an independent implementation of the system call surface, meaning that many of the subsystems or specific calls are not as optimized as more mature implementations.

# *Appendix C*

## *Getting Podman*

Podman is a great tool for working with containers, but how do you get it installed onto your system? What packages are required to make it work? This appendix covers how to install or build Podman on your system.

### **C.1 Installing Podman**

Podman is available for almost all Linux distributions via their package managers. It is also available on Mac, Windows and FreeBSD platforms. The official podman.io site, <https://podman.io/getting-started/installation>, is regularly updated with new instructions on how to install Podman for different distributions. Most of the content in this appendix originates from this site.

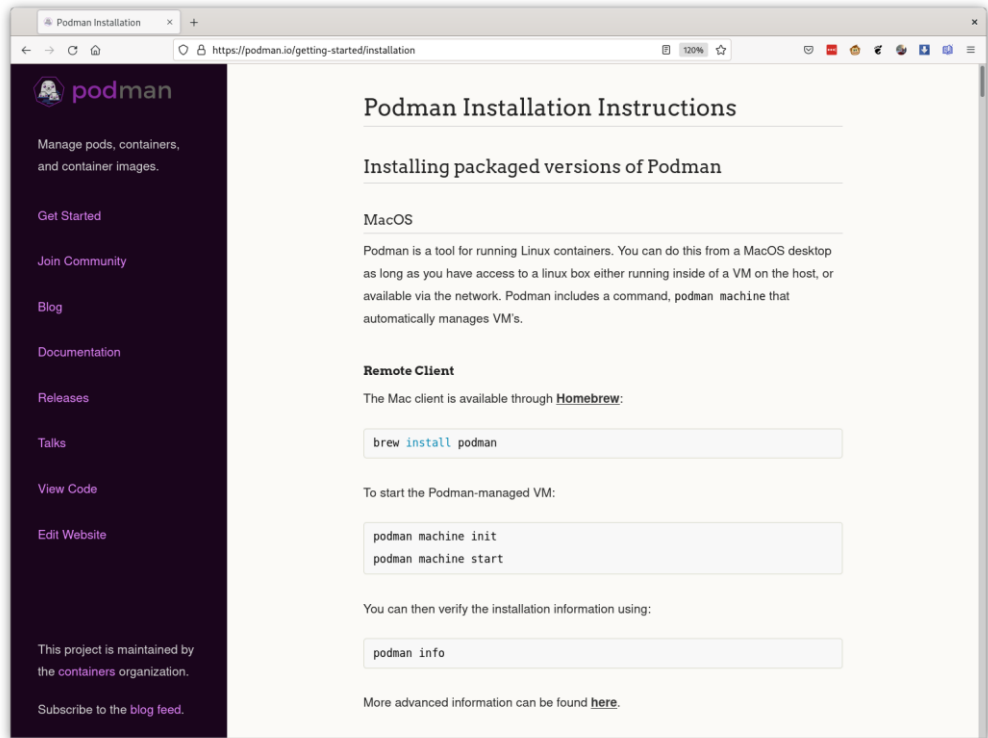


Figure C.1 Podman installation instructions web site

### C.1.1 MacOS

Because Podman is a tool for running Linux containers, you can use it on a MacOS desktop only if you have access to a linux box, running either locally or remotely. To make this process somewhat easier, Podman includes a command, `podman machine` to automatically manage VMs.

#### HOME BREW

The Mac client is available through Homebrew (<https://brew.sh/>):

```
$ brew install podman
```

Podman has the ability to install a VM and run a linux instance on your machine using the `podman machine` command. On a Mac, you must execute the following commands to install and start the Linux VM to successfully run containers locally.

```
$ podman machine init
$ podman machine start
```

Optionally you can use the `podman system connection` command to set up ssh connections to remote Linux machines running the podman service.

You can then verify the installation information using:

```
$ podman info
```

The Podman command is running natively on the MAC but communicating with an instance of Podman running within the virtual machine.

### C.1.2 Windows

Because Podman is a tool for running Linux containers, you can use it on a Windows desktop only if you have access to a linux box, running either locally or remotely. On Windows, Podman can also utilize the Windows Subsystem for Linux system.

#### WINDOWS REMOTE CLIENT

You can retrieve the latest Windows remote client on the <https://github.com/containers/podman/releases> site.

Once installed you can configure the Windows remote client to connect to a Linux server using the `podman system connection` command. Find out more about this process at <https://docs.podman.io/en/latest/markdown/podman-system-connection.1.html>.

#### WINDOWS SUBSYSTEM FOR LINUX (WSL) 2.0

See Windows documentation on Installing WSL 2.0 and then pick a distribution that includes Podman including many described below. Alternatively, `podman machine init` command can bootstrap it all for you by automatically installing and configuring WSL, downloading and provisioning Fedora Core VM on it and creating corresponding ssh connections for the Podman remote client.

**NOTE** WSL 1.0 is not supported.

### C.1.3 Arch Linux & Manjaro Linux

Arch Linux and Manjaro Linux use the pacman tool to install software.

```
$ sudo pacman -S podman
```

### C.1.4 CentOS

Podman is available in the default Extras repos for CentOS 7 and in the AppStream repo for CentOS 8 and Stream.

```
$ sudo yum -y install podman
```

### C.1.5 Debian

The podman package is available in the Debian 11 (Bullseye) repositories and later.

```
$ sudo apt-get -y install podman
```

### C.1.6 Fedora

```
$ sudo dnf -y install podman
```

### C.1.7 Fedora-CoreOS, Fedora SilverBlue

Podman comes preinstalled on these distributions. No need to install

### C.1.8 Gentoo

```
$ sudo emerge app-emulation/podman
```

### C.1.9 OpenEmbedded

Bitbake recipes for Podman and its dependencies are available in the meta-virtualization layer (<https://git.yoctoproject.org/cgit/cgit.cgi/meta-virtualization/>)

Add the layer to your OpenEmbedded build environment and build Podman using:

```
$ bitbake podman
```

### C.1.10 openSUSE

```
sudo zypper install podman
```

### C.1.11 openSUSE Kubic

The openSUSE Kubic distribution has Podman built-in, no need to install

### C.1.12 Raspberry Pi OS arm64

The Raspberry Pi OS uses the standard Debian's repositories, so it is fully compatible with Debian's arm64 repository.

```
$ sudo apt-get -y install podman
```

### C.1.13 Red Hat Enterprise Linux

#### RHEL7

Make sure you have a RHEL7 subscription, then enable the *extras channel* and install Podman.

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

**NOTE** RHEL7 is no longer receiving updates to the Podman package except for security fixes.

#### RHEL8

Podman is included in the container-tools module, along with Buildah and Skopeo.

```
$ sudo yum module enable -y container-tools:rhel8
$ sudo yum module install -y container-tools:rhel8
```



**RHEL9 AND BEYOND**

```
$ sudo yum install podman
```

**C.1.14 Ubuntu**

The podman package is available in the official repositories for Ubuntu 20.10 and newer.

```
$ sudo apt-get -y update
$ sudo apt-get -y install podman
```

**C.2 Building from source code**

I usually advise people to get the packaged versions of Podman, because successfully running Podman on Linux requires having additional tools installed, such as `common` (container monitor), `containernetworking-plugins` (Network configuration) and `containers-common` (general configuration). While the process of building Podman from the source code is not that complicated, the list of dependencies differs from one linux distribution to another. You can always check the latest instructions on this page: <https://podman.io/getting-started/installation#building-from-scratch>

**C.3 Podman Desktop**

There is also a GUI for browsing, managing, inspecting containers and images from different container engines available at <https://github.com/containers/podman-desktop>. It can as well connect to multiple engines at the same time and provides an unified interface. This is a relatively new project under heavy development, so expect some rough edges.

To provide some background and context on this, Docker Inc in Sept 2021 announced that they are going to be charging money for the previously free version of Docker Desktop on macOS. The Docker announcement has caused a lot of people to look for the replacement.

**C.4 Summary**

- Podman is a tool for running Linux containers, so it runs only on Linux
- Podman is available in default package repositories of the most major Linux distributions.
- Podman is available as a remote client on Macs and Windows which connects to either local or remote Linux box
- Podman provides a special command for Linux VM management on macOS and Windows.
- Podman can be built from source code, but requires many other tools to run successfully.
- Podman Desktop is an alternative for a popular Docker Desktop.

# Appendix D

## Contributing to Podman

The number one thing that I love about Open Source is the community effort. It is great to be able to contribute to a project and better yet, get people to contribute to your project. The analogy I like to use comes from Grimms' Fairy Tales, *The Elves* (<https://sites.pitt.edu/~dash/grimm039.html>).

*A shoemaker, through no fault of his own, had become so poor that he had only leather enough for a single pair of shoes. He cut them out one evening, then went to bed, intending to finish them the next morning. Having a clear conscience, he went to bed peacefully, commended himself to God, and fell asleep. The next morning, after saying his prayers, he was about to return to his work when he found the shoes on his workbench, completely finished.*

The story goes on to describe a couple of Elves that come by each night and finish the shoes. I see this as the way Open Source works. Basically the people doing little contributions, bug reporting, bug fixes, document fixes, feature requests and publicizing the project are all the Elves. Sometimes I even go to bed and someone fixes a problem I was attempting to deal with the night before. And sometimes the Elves grow up to be maintainers. Some small contributions over time grow and these developers end up being core members of the Podman team. Some we even hired.

### D.1 Joining the community

Each small change helps make the project better. When I talk to college students about Open Source, I tell them about the unique opportunities that they have, which were not around when I was a student. They can make a contribution to a software project or product and then to list them on their resume. When interviewing a student for an internship or a job, having a few github.com contributions on a resume is very impressive.

Podman and the underlying technologies are always looking for new contributions. No contribution is too small – from a spelling mistake in a man page up to a full blown feature. You don't have to be a software developer to contribute. We are always looking for help on

documentation, web design for podman.io, as well as software. Many great ideas come from users of the product. Just reporting a bug or reporting what you don't like can lead to fresh ideas that improve the project. I often ask people who have set up complicated environments using Podman to blog about it so others can learn.

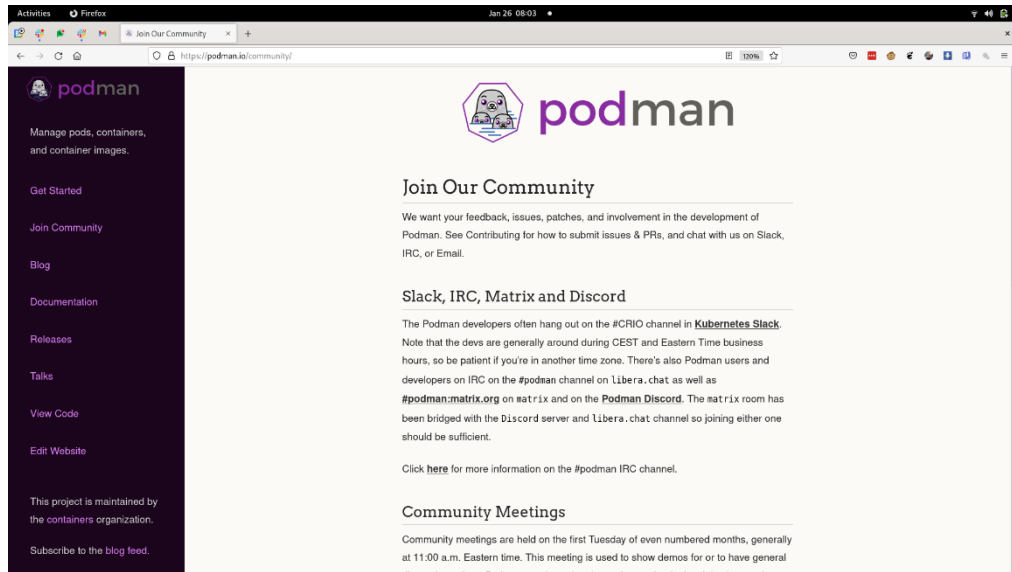


Figure D.1 Podman's community page (<https://podman.io/community>)

Podman is an inclusive community, as are all of the [github.com/containers](https://github.com/containers) projects. The code of conduct statement for the containers project at <https://github.com/containers/common/blob/main/CODE-OF-CONDUCT.md> states the following:

*"As contributors and maintainers of the projects under the <https://github.com/containers> repository, and in the interest of fostering an open and welcoming community, we pledge to respect all people who contribute through reporting issues, posting feature requests, updating documentation, submitting pull requests or patches, and other activities to any of the projects under the containers umbrella."*

## D.2 Podman on github.com

Issues, discussions and pull requests reside on the [github.com/containers/podman](https://github.com/containers/podman) repository. As of this writing the project has over 1,200 forks and 12,000 stars. Bottom line: it is a very active project.

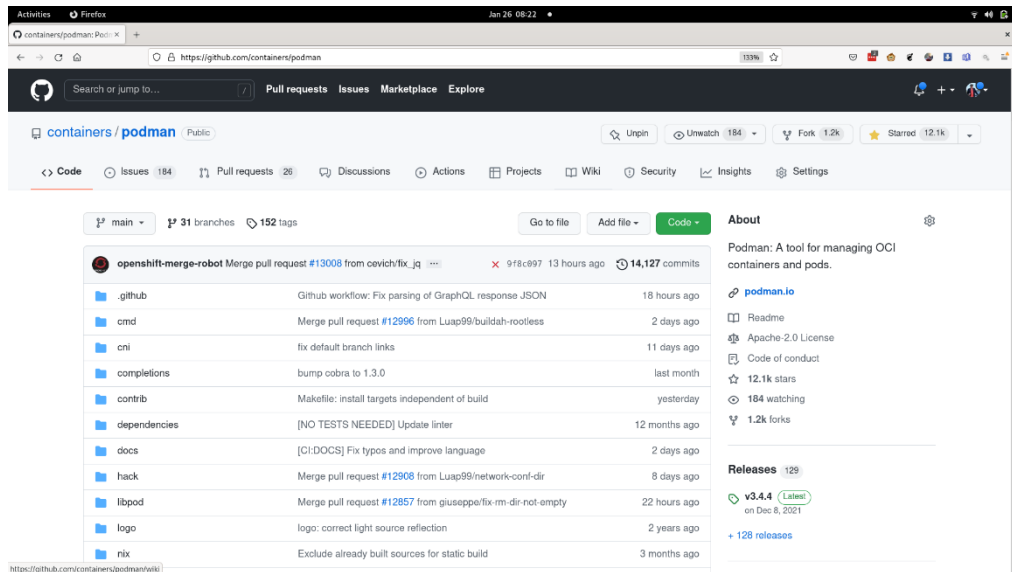


Figure D.2 Podman's github page [github.com/containers/podman](https://github.com/containers/podman)

You can also communicate directly with the core maintainers on IRC on the #podman channel on libera.chat. The IRC channel is also linked to #podman:matrix.org (<https://matrix.to/#/#podman:matrix.org>) on matrix and the Podman Discord (<https://discord.com/invite/x5GzFF6QH4>) for web access.

There is also a low volume Mailing list that you can join by sending an email to [podman-join@lists.podman.io](mailto:podman-join@lists.podman.io).

Finally you can follow @podman\_io on twitter, or follow me @rhatdan.

# Appendix E

## Podman on macOS

### This appendix covers

- Installing podman on a macOS
- Using `podman machine init` command to download a VM with Podman service installed
- Using `podman` command to communicate with the Podman service running in the VM
- Starting or stopping the VM with the `podman machine start/stop` commands

Podman is a tool for launching Linux containers. Linux containers require a Linux kernel. As much as I'd love to convince the world to move to the Linux Desktop like I do, most users work on macOS and Windows operating systems. Perhaps even you. If you use the Linux Desktop, hooray, and if you don't use a macOS machine, feel free to skip this appendix.

Because you did not skip this appendix, I think you want to create Linux containers without having to SSH into a Linux box. You want to use native software development tools and keep development locally.

One way would be to run Podman as a service on a Linux box and use the `podman --remote` command to communicate with this service. Podman provides the `podman system connection` command to configure how podman communicates with a Linux box. However, the problem with this approach is that it is a meticulous process and requires a number of manual steps. Please refer to this web page for an updated tutorial on this process: [https://github.com/containers/podman/blob/main/docs/tutorials/remote\\_client.md](https://github.com/containers/podman/blob/main/docs/tutorials/remote_client.md)

A better way would be to use a new command, `podman machine`, which encapsulates all these steps and improves your experience with managing a Linux box to be used for podman-remote.

In this chapter, you'll learn how to install Podman in macOS and then use the `podman machine` commands to install, configure and manage the VM to allow you to use the native Podman client to launch containers.

## E.1 Podman on macOS

The first step to launching Podman on a macOS is installing it. The macOS client is available through Homebrew (<https://brew.sh/>).

**NOTE** Homebrew describes itself as ‘... the easiest and most flexible way to install the UNIX tools Apple didn’t include with macOS.’

Homebrew is the best way to get open-source software installed on your macOS. If you do not currently have Homebrew installed on your macOS, open a terminal and install it with the following command at the prompt:

```
$ /bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

Now run the following `brew` command to install a trimmed down version of Podman, with only `--remote` support, into the `/opt/homebrew/bin` directory:

```
$ brew install podman
```

If you don’t have access to a Linux VM or a remote Linux server, Podman allows you to create a locally running VM using the `podman machine` command. It makes this easy by creating and configuring a VM with a Podman service enabled.

**NOTE** If you have an existing Linux machine, you can use the `podman system connection` commands to set up connections to those machines.

### E.1.1 Using podman machine

The `podman machine` commands allow you to pull a VM from the internet, start it, stop or remove it. This VM is pre-configured with the Podman service. Additionally, this command creates the `ssh` connection and adds this information to the `podman system connection` datastore greatly simplifying the process of setting up a `podman-remote` environment. Table E.1, lists all of the `podman machine` subcommands used to manage the lifecycle of the Podman virtual machine.

**Table E.1 Podman machine commands**

Command	Description
<code>init</code>	Initialize a new virtual machine
<code>list</code>	List virtual machines
<code>rm</code>	Remove a virtual machine
<code>ssh</code>	Ssh into a virtual machine. It is useful for entering the virtual machine and running the native Podman commands. Some Podman commands are not supported remotely, and you might want to change some configurations inside the VM.
<code>start</code>	Start a virtual machine.
<code>stop</code>	Stop a virtual machine. If you are not running containers, you might want to shut down the VM to save system resources.

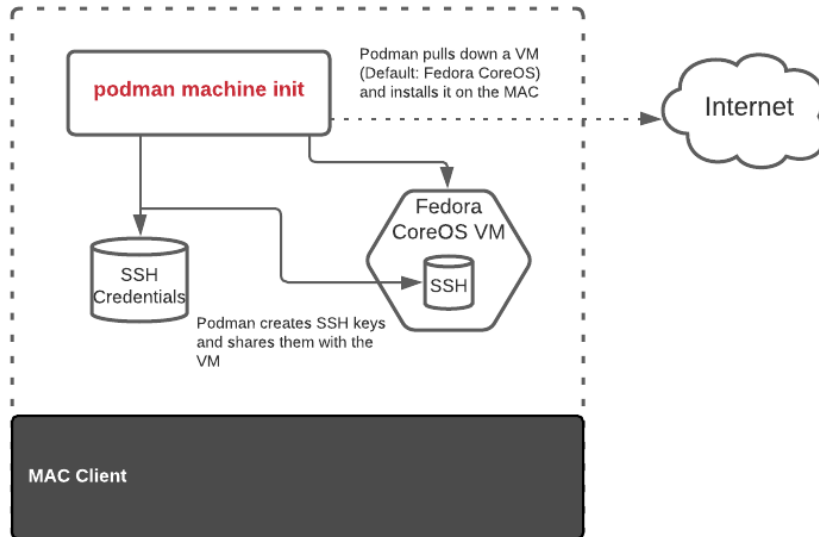
The first step is to initialize a new VM in your system using the `podman machine init` command, described in the following section.

#### **PODMAN MACHINE INIT**

The `podman machine init` command downloads and configures a VM on your macOS system.

**NOTE** The VM is relatively large and takes a few minutes to download.

By default, it downloads the latest released `fedora-coreos` image (<https://getfedora.org/en/coreos>) if it was not downloaded before. Fedora CoreOS is a minimal operating system designed to run containers.



**Figure E.1** The `podman machine init` command pulling the VM and configuring the ssh connections

```
$ podman machine init
Downloading VM image: fedora-coreos-35.20211215.2.0-qemu.x86_64.qcow2.xz #A [=====>----
-----] 111.0MiB / 620.7MiB
Downloading VM image: fedora-coreos-35.20211215.2.0-qemu.x86_64.qcow2.xz: done
Extracting compressed file #B
```

**#A** Podman finds and downloads the latest fedora-coreos qcow image onto your system.

**#B** After downloading the image, Podman decompresses the image and configure qemu to be ready to execute it and configures the ssh connection in to the podman system connection datastore

Podman pre-configures the VM with the amount of memory, disk size, and CPUs for it to use. These values can be configured using `init` subcommand options. Table E.2 describes these options.



**Table E.2 Podman machine init command options**

Option	Description
<code>--cpus uint</code>	Number of CPUs (default 1)
<code>--disk-size uint</code>	Disk size in GB (default 10). It is an important setting to consider since it limits the number of containers and images allowed to be used within the VM. If you have the space, I recommend increasing the field.
<code>--image-path string</code>	Path to qcow image (default "testing"). Podman has two built-in Fedora CoreOS images that it can pull, "testing" and "stable". You can also select other OS's and VMs to download, but the VMs must support CoreOS/ignition files. ( <a href="https://coreos.github.io/ignition/">https://coreos.github.io/ignition/</a> ).
<code>--memory integer</code>	Memory in MB (default 2048). The virtual machine requires a certain amount of memory to run and depending on the containers you want to run within the VM, you might need more.

Once `podman machine init` finishes downloading and installing the VM, you can view the VM with the `podman machine list` command. Notice the `*` indicates the default VM to be used. The `podman machine` command currently only supports running one VM at a time.

```
$ podman machine list
NAME                VM TYPE   CREATED      LAST UP      CPUS  MEMORY  DISK SIZE
podman-machine-default*  qemu     2 minutes ago  2 minutes ago  1     2.147GB 10.74GB
```

In the next section let's examine the automatically created SSH connection.

#### PODMAN MACHINE SSH CONFIGURATION

The `podman machine init` command provides the OS with the Ignition config which includes an SSH key for the `core` user. Then, Podman adds SSH connections on the client machine for the `rootless` and `rootful` modes and configures the user account, adds required packages and configurations within the VM. The SSH configuration allows for password-less SSH commands to the `core` and `root` accounts from the client. The `podman machine init` command also configures the podman system connection information (described in section 9.5.4, chapter 9). The system connection database is configured for both the `rootful` user and the `rootless` user within the VM. If no previous connections are present, the `podman machine init` command will make the newly created connection a default one.

You can examine all of the connections using the `podman system connection list` command. The default connection `podman-machine-default` is the `rootless` connection:

```
$ podman system connection list
Name          URI
Identity      Default
podman-machine-default  ssh://core@localhost:50107/run/user/501/podman/podman.sock
/Users/danwalsh/.ssh/podman-machine-default  true
podman-machine-default-root  ssh://root@localhost:50107/run/podman/podman.sock
/Users/danwalsh/.ssh/podman-machine-default  false
```

Sometimes containers you want to execute require root privileges and can not run in rootless modes. For this, you can modify the system connection to default to the rootful service using the `podman system connection default` command:

```
$ podman system connection default podman-machine-default-root
```

View the connections again to confirm the default connection is now **podman-machine-default-root**.

```
$ $ podman system connection list
Name          URI
Identity      Default
podman-machine-default  ssh://core@localhost:50107/run/user/501/podman/podman.sock
/Users/danwalsh/.ssh/podman-machine-default  false
podman-machine-default-root  ssh://root@localhost:50107/run/podman/podman.sock
/Users/danwalsh/.ssh/podman-machine-default  true
n-machine-default  ssh://root@localhost:38243/run/podman/podman.sock
```

Now all Podman commands connect directly to the Podman service running within the root account. Change the default connection back to the rootless user using the `podman system connection default` command again

```
$ podman system connection default podman-machine-default
```

If you attempt to run a Podman container at this point it fails because the VM is not actually running. You need to start the VM.

### STARTING THE VM

After adding a VM and setting a specific connection as a default one, let's try running a various podman command:

```
$ podman version
Cannot connect to Podman. Please verify your connection to the Linux system using `podman
system connection list`, or try `podman machine init` and `podman machine start` to
manage a new Linux VM
Error: unable to connect to Podman. failed to create sshClient: Connection to bastion host
(ssh://root@localhost:38243/run/podman/podman.sock) failed.: dial tcp [::1]:38243:
connect: connection refused
```

As the error points out, the VM is not running and must be started.

You start a single VM using the `podman machine start` command. Podman only supports running one VM at a time. By default, the start command starts the default VM. If you have multiple VMs and you want to start a different VM, you can specify the optional machine name.

```
$ podman machine start
INFO[0000] waiting for clients...
INFO[0000] listening tcp://127.0.0.1:7777
INFO[0000] new connection from @ to /run/user/3267/podman/qemu_podman-machine-default.sock
Waiting for VM ...
macOSShine "podman-machine-default" started successfully
```

You are now ready to begin running podman commands on the Linux box which runs the Podman service. Run the `podman version` command to confirm the client and server are configured correctly. If not, the Podman commands should instruct you on how to configure the system.

```
$ podman version
Client:
Version:      4.1.0
API Version:  4.1.0
Go Version:   go1.18.1
Built:        Thu May  5 16:07:47 2022
OS/Arch:      darwin/arm64
Server:
Version:      4.1.0
API Version:  4.1.0
Go Version:   go1.18
Built:        Fri May  6 12:16:38 2022
OS/Arch:      linux/arm64
```

Now you can use the Podman commands that you learned in the previous chapters directly on macOS. When you are done with the working with containers in the VM, you probably should shut it down to save on resources.

**NOTE** Podman is supported on M1 Arm64 machines as well as the X86 platforms. Podman machine init downloads the matching architecture VM, allowing you to build images for that architecture. Support for building images on other architectures is being worked on as of this writing.

### STOPPING THE VM

The `podman machine stop` command allows you to shut down all containers within the VM as well as the VM itself:

```
$ podman machine stop
```

When you need to start using containers again, launch the VM with the `podman machine start` command.

**NOTE** All of the `podman machine` commands work on Linux as well and allow you to test different versions of Podman at the same time. Podman on Linux is the complete command, therefore you need to use the `--remote` option to communicate with the Podman service running within the VM launched by the podman machine. On non-linux platforms the `--remote` option is not required since the client is preconfigured in `--remote` mode.

## E.2 Summary

- Linux containers require a Linux kernel, meaning running containers on a macOS requires a virtual machine running Linux
- Podman on a macOS is not running containers locally on the macOS. The Podman command is actually communicating with the Podman service running on a Linux machine
- The `podman machine init` pulls down and installs a Fedora CoreOS virtual machine onto your platform which is running the Podman service
- The `podman machine init` command also sets up the SSH environment required to allow the Podman remote client to communicate with the Podman server inside of the VM

# Appendix F

## Podman on Windows

### This appendix covers

- Installing podman on Windows
- Using `podman machine init` command to create a Fedora-based WSL2 distribution running Podman
- Using the `podman` command on Windows to communicate with the Podman service running in the WSL2 instance
- Starting or stopping the WSL2 instance with the `podman machine start/stop` commands

Podman is a tool for launching Linux containers. Linux containers require a Linux kernel. As much as I'd love to convince the world to move to the Linux Desktop like me, most users work on Mac and Windows operating systems. Perhaps even you. If you use the Linux Desktop, hooray, and if you don't use a Windows machine, feel free to skip this chapter.

Because you did not skip this chapter, I think you want to create Linux containers without having to SSH into a Linux machine and create the container there. You want to use native software development tools and keep their software local to their machines.

On Linux, Podman can be run as a service allowing remote connections to launch containers. Then from another system, the `podman --remote` command can be used to communicate with the remote Podman service to launch a container.

Further, you can use `podman system connection` to configure `podman --remote` to communicate with a remote Linux machine running the Podman service over SSH without providing a URL to every command. The problem with all of this is that someone has to configure the remote machine with the correct version of the Podman service, and then you have to configure the SSH session.

Realizing that this experience is not optimal for new users of Podman on a Windows desktop, Podman added a new command, `podman machine`. The `podman machine` command makes it easy to create and manage a WSL2 based Linux environment with Podman pre-

installed and configured. The Podman command on Windows is actually a thinned down Podman command with only `podman --remote` support.

In this chapter, you'll learn how to install Podman onto your Windows machine and then use the `podman machine` commands to install, configure and manage the WSL2 instance.

## F.1 First Steps

The `podman machine` command on Windows accepts all the same commands as those used on Linux and Mac with very similar behavior. Still, there are a few differences since the underlying backend on Windows is based on Windows Subsystem for Linux (<https://docs.microsoft.com/en-us/windows/wsl/>) instead of the VM as in the other operating systems.

WSL v2 involves using the Windows Hyper-V hypervisor; however, unlike a standard VM - based approach, WSL2 shares the same virtual machine and Linux kernel instance across every Linux distribution instance installed by the user. As an example if you create two WSL2 distributions, and you run `dmesg` on each instance, you see the same output, since the same Kernel is hosting both.

**NOTE** WSL V1 doesn't work with Podman, you must upgrade your Windows machine to an OS version that supports WSL V2. For x64 systems: Windows Version 1903 or higher, with Build 18362 or higher. For ARM64 systems: Windows Version 2004 or higher, with Build 19041 or higher.

Running Podman with WSL2 enables efficient resource sharing between the host and all running instances in exchange for less isolation. Keep in mind that the `podman machine` command shares the same kernel with any other distributions you have running and be cautious when manipulating any kernel level setting (network interfaces, netfilter policy, and so on) in any distribution, because you may unintentionally impact containers executed by Podman.

### F.1.1 Prerequisites

Podman for Windows requires Windows 10 (Build 19041 or later) or Windows 11. As WSL v2 uses a hypervisor, your computer must have virtualization instructions enabled (e.g., Intel VT-x or AMD-V). Additionally, the hypervisor requires Second Level Address Translation (SLAT) support. Finally, your system must either have internet connectivity or an offline copy of all software to be fetched by the Podman machine.

**NOTE** If at any time you experience the errors 0x80070003 or 0x80370102 (or any error indicating the virtual machine can not be started), you most likely have virtualization disabled. Check your BIOS (or WSL2 instance) settings to verify VT-x/AMD-V/WSL2 instance and SLAT are enabled.

While not required, installing Windows Terminal (as opposed to the standard CMD command application or Powershell) is strongly recommended (future versions of Windows 11 include it by default). In addition to having modern terminal features like transparent cut and

paste and tiled screens, it also offers direct WSL and PowerShell integration, making it easy to switch between environments. You can install it via the Windows store or via winget:

```
PS C:\Users\User> winget install Microsoft.WindowsTerminal
```

## F.1.2 Installing Podman

Installing Podman is straightforward. Go to the podman site or the Podman Github repository and download the latest Podman MSI Windows Installer in the [releases](#) section.

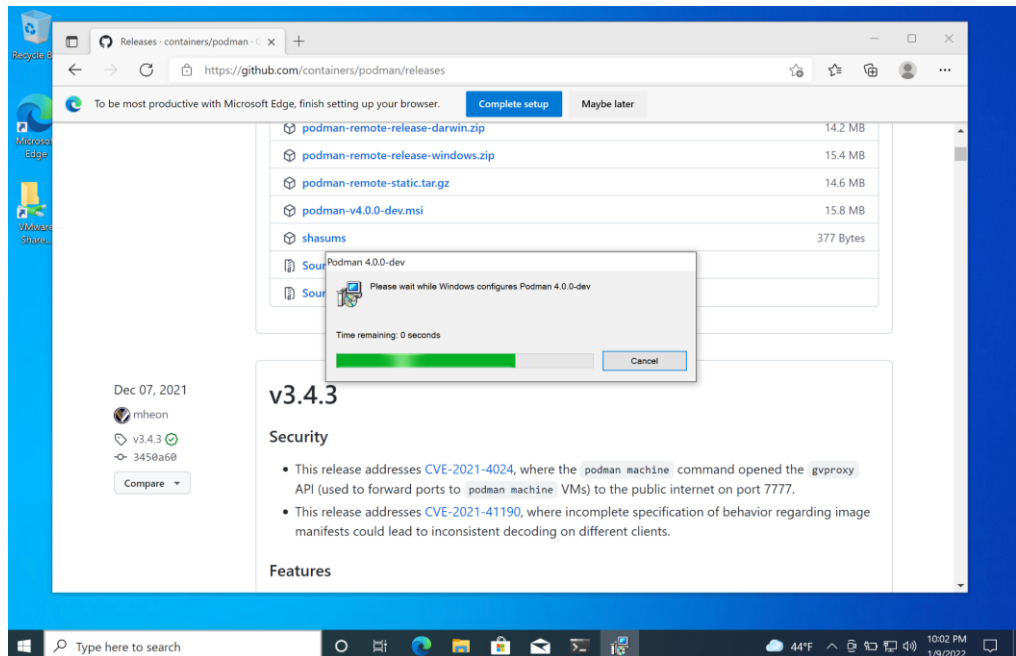
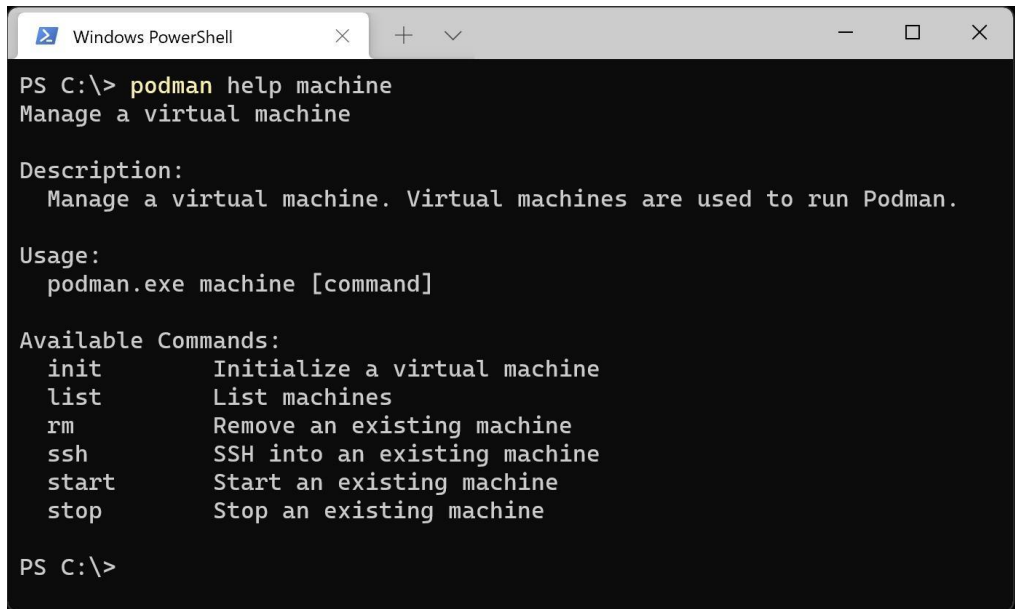


Figure F.1 Downloading and running the podman installer

After running the installer, open a terminal (use the wt command if you installed Windows Terminal as recommended), and execute your first podman command:

A screenshot of a Windows PowerShell terminal window. The title bar shows 'Windows PowerShell' with standard window controls. The terminal text is as follows:

```
PS C:\> podman help machine
Manage a virtual machine

Description:
  Manage a virtual machine. Virtual machines are used to run Podman.

Usage:
  podman.exe machine [command]

Available Commands:
  init      Initialize a virtual machine
  list      List machines
  rm        Remove an existing machine
  ssh       SSH into an existing machine
  start     Start an existing machine
  stop      Stop an existing machine

PS C:\>
```

Figure F.2 Podman commands running within the Windows Terminal

#### AUTOMATIC WSL INSTALLATION

If WSL is not installed on your Windows system, Podman installs it for you. Simply execute the `podman machine init` command (as described in F.2.1) to create your first machine instance, and Podman prompts you for permission to install WSL. The WSL install process requires a reboot but resumes execution of the machine creation process (Be sure to wait a few minutes for the terminal to relaunch and install)

If you prefer a manual installation, refer to the WSL installation guide: <https://docs.microsoft.com/en-us/windows/wsl/install>



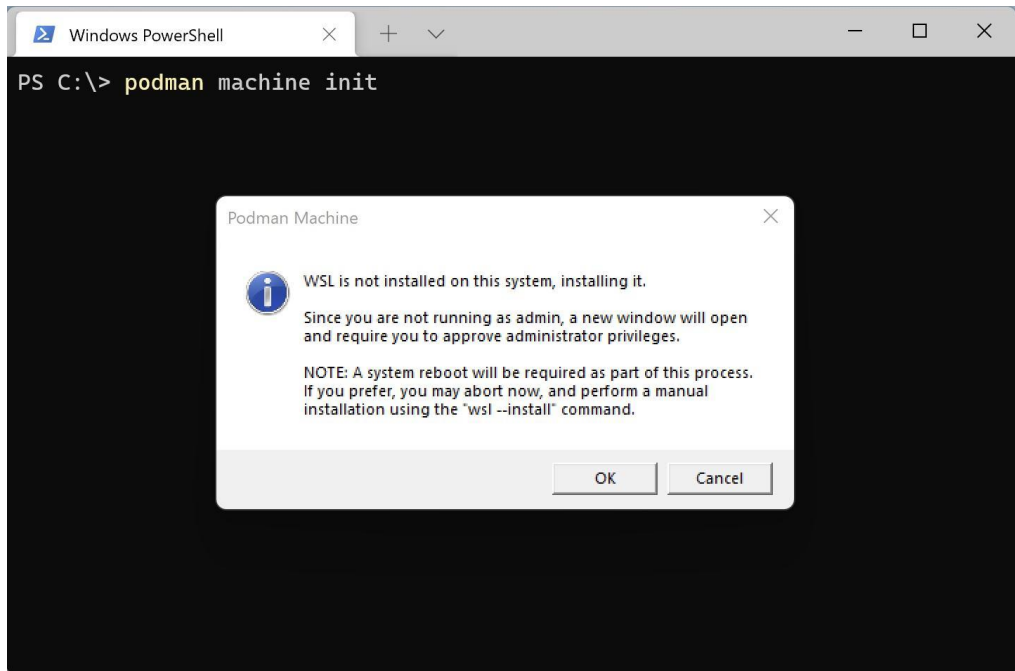


Figure F.3 The podman machine init starts the WSL Install

## F.2 Using podman machine

The setup and use of the Linux environment is made easy through the use of `podman machine` commands. On Windows, these commands create and manage a WSL2 distribution including downloading a base Linux image and packages from the internet and setting everything up for you. The WSL2 distribution is pre-configured with the Podman service, and ssh connection configuration is automatically added to the `podman system connection` datastore. The final result is the ability to easily run Podman commands on your Windows desktop as if it was a Linux system. Table F.1 lists all of the `podman machine` commands used to manage the lifecycle of the WSL2 backed Linux environment.

**Table F.1 Podman machine commands**

Command	Description
<code>init</code>	Initialize a new WSL2 based machine instance
<code>list</code>	List WSL2 machines
<code>rm</code>	Remove a WSL2 machine instance.
<code>set</code>	Sets an updatable WSL machine setting
<code>ssh</code>	Ssh into a WSL2 machine instance. Useful for entering the WSL2 instance and running the native Podman commands. Some Podman commands are not supported remotely, and you might want to change some configurations inside the WSL2 instance.
<code>start</code>	Start a WSL2 machine instance.
<code>stop</code>	Stop a WSL2 machine instance. If you are not running containers, you might want to stop to save system resources.

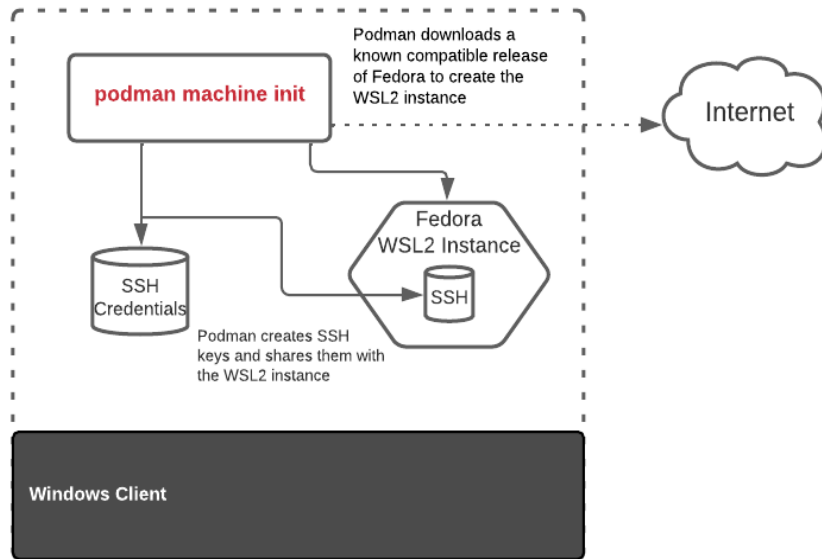
After installing podman (see section F.1.2) The first step is to create a WSL2 machine instance on your system using the `podman machine init` command, described in the following section.

### F.2.1 Podman machine init

As shown in Fig F.5, you can use the `podman machine init` command to automate the installation of a WSL2-based Linux environment which hosts a Podman service for running containers

**NOTE** In addition to the base image, a number of packages must be downloaded and installed, which can take several minutes to complete.

By default, `podman machine init` downloads a known compatible release of `Fedora` to create the WSL2 instance (<https://getfedora.org>). Fedora is used since it is well integrated with Podman and is the operating system used by most of the Podman core developers.



**Figure F.4** The `podman machine init` command creating the WSL2 distribution and configuring ssh connections

The following shows the condensed output from running the `podman machine init` command.

```
PS C:\Users\User> podman machine init
Downloading VM image: fedora-35.20211125-x86_64.tar.xz: done
Extracting compressed file
Importing operating system into WSL (this may take 5+ minutes on a new WSL install)...
Installing packages (this will take awhile)...
Fedora 35 - x86_64                    5.5 MB/s | 79 MB    00:14
Complete!
Configuring system...
Generating public/private ed25519 key pair.
Machine init complete
To start your machine run:
    podman machine start
```

Table F.2 explains the init options that allow you to customize the default settings.

**Table F.2 Podman machine init command options**

Option	Description
<code>--cpus uint</code>	Not used
<code>--disk-size uint</code>	Not used
<code>--image-path string</code>	On Windows, this option refers to the Fedora distribution number (e.g., 35). As with Linux and Mac, you can also specify an arbitrary URL or filesystem location with a custom image, but podman expects a Fedora-derived layout.
<code>--memory integer</code>	Not used
<code>--rootful</code>	Whether this machine instance should be rootful or rootless

**NOTE** The physical limits specified in the table above (CPU, memory, disk) are currently ignored on Windows since the Windows Subsystem for Linux (WSL) backend dynamically resizes and shares resources across distributions. If you need to constrain resources, you can configure those limits in your users' `.wslconfig` file. However, they apply globally to all WSL2 distros, since they share the same underlying VM

## F.2.2 Podman machine SSH configuration

The `podman machine init` command creates an account within the WSL2 instance. By default, the user in Fedora is `user@localhost`. Podman configures SSH on the client machine and the new user account and root within the WSL2 instance. The SSH configuration allows for password-less SSH commands to the `user` and `root` accounts from the client. The `podman machine init` command also configures the podman system connection information (described in section 9.5.4, chapter 9). The system connection database is configured for both the rootful user and the rootless user within the WSL2 instance. If you do not have any existing connections, the `podman machine init` command creates and sets as a default one the rootless user connection to your WSL2 instance.

You can examine all of the connections using the `podman system connection list` command. The default connection `podman-machine-default` is the rootless connection:

```
PS C:\Users\User> podman system connection ls
Name                               URI                               Identity                               Default
podman-machine-default            ssh://user@localhost:57051..     podman-machine-default              true
podman-machine-default-root       ssh://root@localhost:57051..     podman-machine-default              false
```

Sometimes containers you want to execute require root privileges and can not run in rootless modes. You can change the default connection to rootful by switching the default mode for the created machine instance. Modify the default to rootful service using the `podman machine set` command:

```
PS C:\Users\User> podman machine set --rootful
```

View the connections again to confirm the default is now `podman-machine-default-root`.

```
PS C:\Users\User> podman system connection ls
```

Name	URI	Identity	Default
podman-machine-default	ssh://user@localhost:57051..	podman-machine-default	false
podman-machine-default-root	ssh://root@localhost:57051..	podman-machine-default	true

Now all Podman commands connect directly to the Podman service running within the root account. Change the default connection back to the rootless user using the podman machine set command again:

```
PS C:\Users\User> podman machine set --rootful=false
```

If you attempt to run a Podman container at this point it fails because the machine instance is not actually running. You need to start the machine instance.

### F.2.3 Starting the WSL2 instance

Attempting to execute the `podman version` command fails because the WSL2 instance is not started:

```
PS C:\Users\User> podman version
Cannot connect to Podman. Please verify your connection to the Linux system using `podman
system connection list`, or try `podman machine init` and `podman machine start` to
manage a new Linux Linux VM
Error: unable to connect to Podman. failed to create sshClient: Connection to bastion host
(ssh://root@localhost:38243/run/podman/podman.sock) failed.: dial tcp [::1]:38243:
connect: connection refused
```

As the error points out, the virtualized Linux environment (the WSL2 machine instance) is not running and must be started.

You start a single WSL2 instance using the `podman machine start` command. By default, it starts the default WSL2 instance, `podman-machine-default`. If you have multiple WSL2 instances and want to start a different WSL2 instance, you can specify the optional machine name for the `podman machine start` command.

```
PS C:\Users\User> podman machine start
Starting machine "podman-machine-default"
This machine is currently configured in rootless mode. If your containers
require root permissions (e.g. ports < 1024), or if you run into compatibility
issues with non-podman clients, you can switch using the following command:
    podman machine set --rootful
API forwarding listening on: npipe://./pipe/docker_engine
Docker API clients default to this address. You do not need to set DOCKER_HOST.
Machine "podman-machine-default" started successfully
```

You are now ready to begin running podman commands on the host which communicate with the Podman service running in the WSL2 instance. Run the `podman version` command to confirm the client and server are configured correctly. If not, the Podman commands instruct you on how to configure the system.

```

PS C:\Users\User> podman version
Client:      Podman Engine
Version:     4.0.0-dev
API Version: 4.0.0-dev
Go Version:  go1.17.1
Git Commit:  bac389043f268e632c45fed7b4e88bdefd2d95e6-dirty
Built:      Wed Feb 16 00:33:20 2022
OS/Arch:    windows/amd64
Server:     Podman Engine
Version:     4.0.1
API Version: 4.0.1
Go Version:  go1.16.14
Built:      Fri Feb 25 13:22:13 2022
OS/Arch:    linux/amd64

```

Now you can use the Podman commands that you learned in the previous chapters directly on Windows but understand that Podman on Windows is equivalent to `podman --remote` talking remotely to the Podman service within the WSL2 instance.

## F.2.4 Using Podman machine commands

After your machine instance is running, you can perform podman commands in your PowerShell prompt as if running within Windows.

```

PS C:\Users\User> podman run ubi8-micro date
Thu Jan 6 05:09:59 UTC 2022

```

## F.2.5 Stopping the WSL2 instance

When you are done using containers on your system, you might want to shut down the WSL2 instance to save on system resources. Use the `podman machine stop` command to shut down all containers within the WSL2 instance as well as the WSL2 instance.

```

PS C:\Users\User> podman machine stop

```

When you need to start using containers again, launch the WSL2 instance with the `podman machine start` command.

**NOTE** All of the `podman machine` commands work on Linux as well and allow you to test different versions of Podman at the same time. Podman on Linux is the complete command, therefore you need to use the `--remote` option to communicate with the Podman service running within the WSL2 instance launched by the `podman machine` command. On non-linux platforms the `--remote` option is not required since the client is preconfigured in `--remote` mode.

## F.2.6 Listing machines

You can list the available machine instances using the `podman machine ls` command. The values returned by this command on Windows reflect current active usage, as opposed to fixed resource limits as is the case on Mac and Linux. Disk storage reflects the disk space currently allocated to each machine instance. The CPU values convey the # of CPUs on the Windows host (unless limited by WSL) repeated per machine instance. The returned memory values are

also repeated (with slight variation from sampling variance) and reflect the total amount of memory used by the Linux Kernel for ALL distributions in use (since it is shared). In other words, for total usage, you sum the disk sizes, but not memory and CPU.

```
PS C:\Users\User> podman machine ls
```

NAME	SIZE	VM TYPE	CREATED	LAST UP	CPUS	MEMORY	DISK
podman-machine-default		wsl	3 days ago	Running	4	528.4MB	845.2MB
other		wsl	4 minutes ago	Running	4	524.5MB	778MB

## F.2.7 Using Podman at the WSL prompt

In addition to the `podman machine ssh` command, you can also access the podman machine guest using the WSL prompt. If you are running Windows Terminal, the podman machine guests (names prefixed by podman) are in the down arrow dropdown. Alternatively, you can drop into a WSL shell from any PowerShell prompt by using the `wsl` command and specifying the backing distribution name. For example, the default instance created by `podman machine init` is named `podman-machine-default`.

You can use either approach to manage the guest and execute podman commands inside a full-featured Linux shell environment.

```
PS C:\Users\User> wsl -d podman-machine-default
[root@WIN10PRO /]# podman version
Client:      Podman Engine
Version:     4.0.1
API Version: 4.0.1
Go Version:  go1.16.14

Built:      Fri Feb 25 13:22:13 2022
OS/Arch:    linux/amd64
```

## F.2.8 Updating Fedora

Since the Windows machine implementation is based on Fedora and not Fedora CoreOS, fixes and enhancements are not automatic. They must be explicitly initiated on the guest using Fedora's package management command, DNF. Further, upgrading to a new version of Fedora requires exporting any data you need to preserve and using `podman machine init` to create a second machine instance (or replacing the existing one after a `podman machine rm` command).

**NOTE** Currently it is difficult to run Fedora CoreOS inside of WSL, so it was decided to default to Fedora. If Windows support for CoreOS changes in the future, podman machine will move to Fedora CoreOS.

As an example, to pull the latest packages for the version of Fedora running on the podman guest perform the following command:

```
PS C:\Users\User> podman machine ssh dnf upgrade -y
Warning: Permanently added '[localhost]:52581' (ED25519) to the list of known hosts.
Last metadata expiration check: 1:18:35 ago on Wed Jan  5 21:13:15 2022.
Dependencies resolved.
...
Complete!
```

### F.2.9 Advanced Stopping and Restarting

Normally, to stop and restart Podman, use the respective `podman machine stop` and `podman machine start` commands. Stopping the machine is the preferred approach since system services can come to a clean stop. However, in some cases, you may wish to force a hard restart of the WSL facilities, including the shared Linux kernel, which stays active even after a machine stop. To kill all processes associated with a WSL distribution, use the `wsl --terminate <machine name>` command. To shutdown the Linux kernel, killing all running distributions, use the `wsl --shutdown` command. After these commands are issued, you can use a standard `podman machine start` command to relaunch your instance.

```
PS C:\Users\User> wsl --shutdown
PS C:\Users\User> podman machine start
Starting machine...
Machine "podman-machine-default" started successfully
```

## F.3 Summary

- Linux containers require a Linux kernel, meaning running containers on a Mac or Windows platform requires a virtual machine running Linux.
- Podman on Windows is not running containers locally on Windows. The Podman command is actually `podman --remote` communicating with the Podman service running on a Linux machine backed by WSL2.
- The `podman machine init` pulls down and installs a virtual Linux environment onto your platform which runs the Podman service
- The `podman machine init` command also sets up the SSH environment required to allow the Podman remote client to communicate with the Podman server inside of the WSL2 instance
- Podman on Windows with WSL, is the full Podman command. WSL is running the Podman commands under the Linux kernel, even though it feels like it is running natively on the Windows machine.