Containers and Pods 101

We are Opensource.com

Opensource.com is a community website publishing stories about creating, adopting, and sharing open source solutions. Visit Opensource.com to learn more about how the open source way is improving technologies, education, business, government, health, law, entertainment, humanitarian efforts, and more.

Do you have an open source story to tell? Submit a story idea at opensource.com/story

Email us at open@opensource.com

Supported by
Red Hat

Table of Contents

A sysadmin's guide to containers

By Daniel J Walsh

The term "containers" is heavily overused. Also, depending on the context, it can mean different things to different people.

Traditional Linux containers are really just ordinary processes on a Linux system. These groups of processes are isolated from other groups of processes using resource constraints (control groups [cgroups]), Linux security constraints (Unix permissions, capabilities, SELinux, AppArmor, seccomp, etc.), and namespaces (PID, network, mount, etc.).

If you boot a modern Linux system and took a look at any process with `cat /proc/PID/cgroup`, you see that the process is in a cgroup.

If you look at `/proc/PID/status`, you see capabilities. If you look at `/proc/self/attr/current`, you see SELinux labels. If you look at `/proc/PID/ns`, you see the list of namespaces the process is in. So, if you define a container as a process with resource constraints, Linux security constraints, and namespaces, by definition every process on a Linux system is in a container. This is why we often say Linux is containers, containers are Linux. **Container runtimes** are tools that modify these resource constraints, security, and namespaces and launch the container.

Docker introduced the concept of a **container image**, which is a standard TAR file that combines:

- **Rootfs (container root filesystem):** A directory on the system that looks like the standard root (`/`) of the operating system. For example, a directory with `/usr`, `/var`, `/home`, etc.
- **JSON file (container configuration):** Specifies how to run the rootfs; for example, what **command** or **entrypoint** to run in the rootfs when the container starts; **environment variables** to set for the container; the container's **working directory**; and a few other settings.

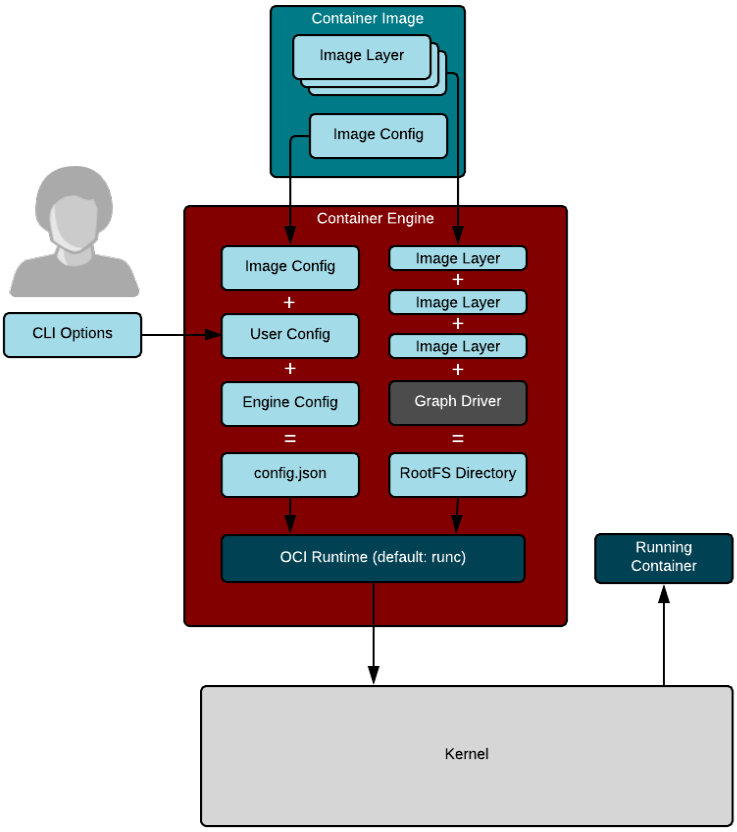
Docker "tar's up" the rootfs and the JSON file to create the **base image**. This enables you to install additional content on the rootfs, create a new JSON file, and tar the difference between the original image and the new image with the updated JSON file. This creates a **layered image**.

The definition of a container image was eventually standardized by the Open Container Initiative (OCI) standards body as the OCI Image Specification.

Tools used to create container images are called **container image builders**. Sometimes container engines perform this task, but several standalone tools are available that can build container images.

Docker took these container images (**tarballs**) and moved them to a web service from which they could be pulled, developed a protocol to pull them, and called the web service a **container registry**.

Container engines are programs that can pull container images from container registries and reassemble them onto **container storage**. Container engines also launch **container runtimes** (see below).



Linux container internals. Illustration by Scott McCarty. CC BY-SA 4.0

Container storage is usually a **copy-on-write** (COW) layered filesystem. When you pull down a container image from a container registry, you first need to untar the rootfs and place it on disk. If you have multiple layers that make up your image, each layer is downloaded and stored on a different layer on the COW filesystem. The COW filesystem allows each layer to be stored separately, which maximizes sharing for layered images. Container engines often support multiple types of container storage, including `overlay`, `devicemapper`, `btrfs`, `aufs`, and `zfs`.

After the container engine downloads the container image to container storage, it needs to create a **container runtime configuration.** The runtime configuration combines input from the caller/user along with the content of the container image specification. For example, the caller might want to specify modifications to a running container's security, add additional environment variables, or mount volumes to the container.

The layout of the container runtime configuration and the exploded rootfs have also been standardized by the OCI standards body as the OCI Runtime Specification.

Finally, the container engine launches a **container runtime** that reads the container runtime specification; modifies the Linux cgroups, Linux security constraints, and namespaces; and launches the container command to create the container's **PID 1**. At this point, the container engine can relay `stdin/stdout` back to the caller and control the container (e.g., stop, start, attach).

Note that many new container runtimes are being introduced to use different parts of Linux to isolate containers. People can now run containers using KVM separation (think mini virtual machines) or they can use other hypervisor strategies (like intercepting all system calls from processes in containers). Since we have a standard runtime specification, these tools can all be launched by the same container engines. Even Windows can use the OCI Runtime Specification for launching Windows containers.

At a much higher level are **container orchestrators.** Container orchestrators are tools used to coordinate the execution of containers on multiple different nodes. Container orchestrators talk to container engines to manage containers. Orchestrators tell the container engines to start containers and wire their networks together. Orchestrators can monitor the containers and launch additional containers as the load increases.

3 steps to start running containers today

By Seth Kenlon

Whether you're interested in them as part of your job, for future job opportunities, or just out of interest in new technology, containers can seem pretty overwhelming to even an experienced systems administrator. So how do you actually get started with containers? And what's the path from containers to Kubernetes? Also, why is there a path from one to the other at all? As you might expect, the best place to start is the beginning.

1. Understanding containers

On second thought, starting at the beginning arguably dates back to early BSD and their special chroot jails, so skip ahead to the middle instead.

Not so very long ago, the Linux kernel introduced *cgroups*, which enables you to "tag" processes with something called a *namespace*. When you group processes together into a namespace, those processes act as if nothing outside that namespace exists. It's as if you've put those processes into a sort of container. Of course, the container is virtual, and it exists inside your computer. It runs on the same kernel, RAM, and CPU that the rest of your operating system is running on, but you've contained the processes.

Pre-made containers get distributed with just what's necessary to run the application it contains. With a container engine, like Podman, Docker, or CRI-O, you can run a containerized application without installing it in any traditional sense. Container engines are often cross-platform, so even though containers run Linux, you can launch containers on Linux, macOS, or Windows.

More importantly, you can run more than one container of the same application when there's high demand for it.

Now that you know what a container is. The next step is to run one.

2. Run a container

Before running a container, you should have a reason for running a container. You can make up a reason, but it's helpful for that reason to interest you, so you're inspired actually to use the container you run. After all, running a container but never using the application it provides only proves that you're not noticing any failures, but using the container demonstrates that it works.

I recommend WordPress as a start. It's a popular web application that's easy to use, so you can test drive the app once you've got the container running. While you can easily set up a WordPress container, there are many configuration options, which can lead you to discover more container options (like running a database container) and how containers communicate.

I use Podman, which is a friendly, convenient, and daemonless container engine. If you don't have Podman available, you can use the Docker command instead. Both are great open source container engines, and their syntax is identical (just type `docker` instead of `podman`). Because Podman doesn't run a daemon, it requires more setup than Docker, but the ability to run rootless daemonless containers is worth it.

If you're going with Docker, you can skip down to the WordPress subheading. Otherwise, open a terminal to install and configure Podman:

```
$ sudo dnf install podman
```

Containers spawn many processes, and normally only the root user has permission to create thousands of process IDs. Add some extra process IDs to your user by creating a file called `/etc/subuid` and defining a suitably high start UID with a suitable large number of permitted PIDs:

```
seth:200000:165536
```

Do the same for your group in a file called `/etc/subgid`. In this example, my primary group is `staff` (it may be `users` for you, or the same as your username, depending on how you've configured your system.)

```
staff:200000:165536
```

Finally, confirm that your user is also permitted to manage thousands of namespaces:

```
$ sysctl --all --pattern user_namespaces
user.max_user_namespaces = 28633
```

If your user doesn't have permission to manage at least 28,000 namespaces, increase the number by creating the file `/etc/sysctl.d/userns.conf` and enter:

```
user.max_user_namespaces=28633
```

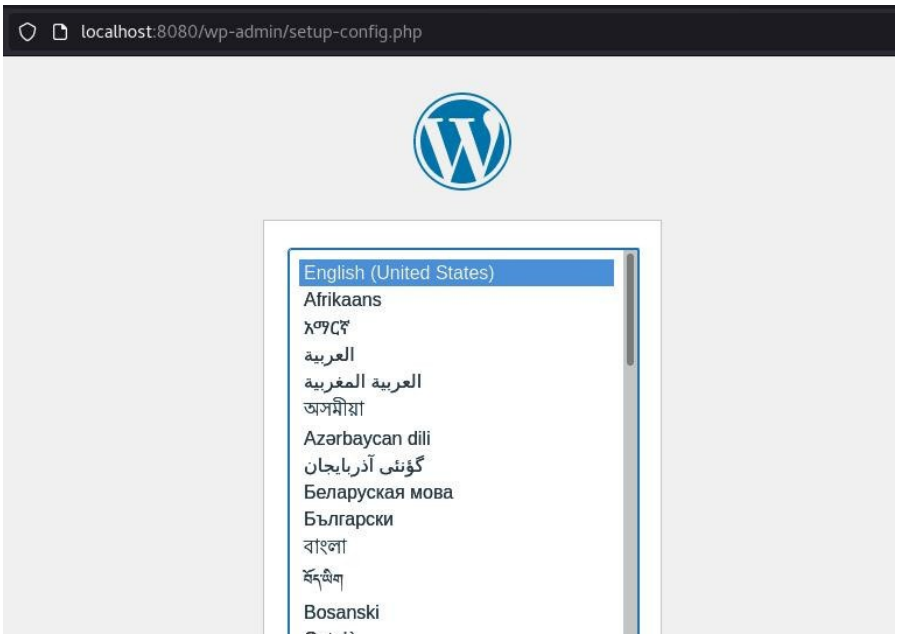
Running WordPress as a container

Now, whether you're using Podman or Docker, you can pull a WordPress container from a container registry online and run it. You can do all this with a single Podman command:

```
$ podman run --name mypress -p 8080:80 -d wordpress
```

Give Podman a few moments to find the container, copy it from the internet, and start it up.

Start a web browser once you get a terminal prompt back and navigate to `localhost:8080`. WordPress is running, waiting for you to set it up.



(Seth Kenlon, CC BY-SA 4.0)

It doesn't take long to reach your next hurdle, though. WordPress uses a database to keep track of data, so you need to provide it with a database where it can store its information.

Before continuing, stop and remove the WordPress container:

```
$ podman stop mypress
$ podman rm mypress
```

3. Run containers in a pod

Containers are, by design and, as their name suggests, self-contained. An application running in a container isn't supposed to interact with applications or infrastructure outside of its container. So when one container requires another container to function, one solution is to put those two containers inside a bigger container called a *pod*. A pod ensures that its containers can share important namespaces to communicate with one another.

Create a new pod, providing a name for the pod and which ports you want to be able to access:

```
$ podman pod create --name wp_pod --publish 8080:80
```

Confirm that the pod exists:

```
$ podman pod list
POD ID        NAME     STATUS    INFRA ID      # OF CONTAINERS
100e138a29bd  wp_pod   Created   22ace92df3ef   1
```

Add a container to a pod

Now that you have a pod for your interdependent containers, you launch each container by specifying a pod for it to run in.

First, launch a database. You can make up your own credentials as long as you use those same credentials when connecting to the database from WordPress.

```
$ podman run --detach \
--pod wp_pod \
-restart=always \
-e MYSQL_ROOT_PASSWORD="badpassword0" \
-e MYSQL_DATABASE="wp_db" \
-e MYSQL_USER="tux" \
-e MYSQL_PASSWORD="badpassword1" \
--name=wp_db mariadb
```

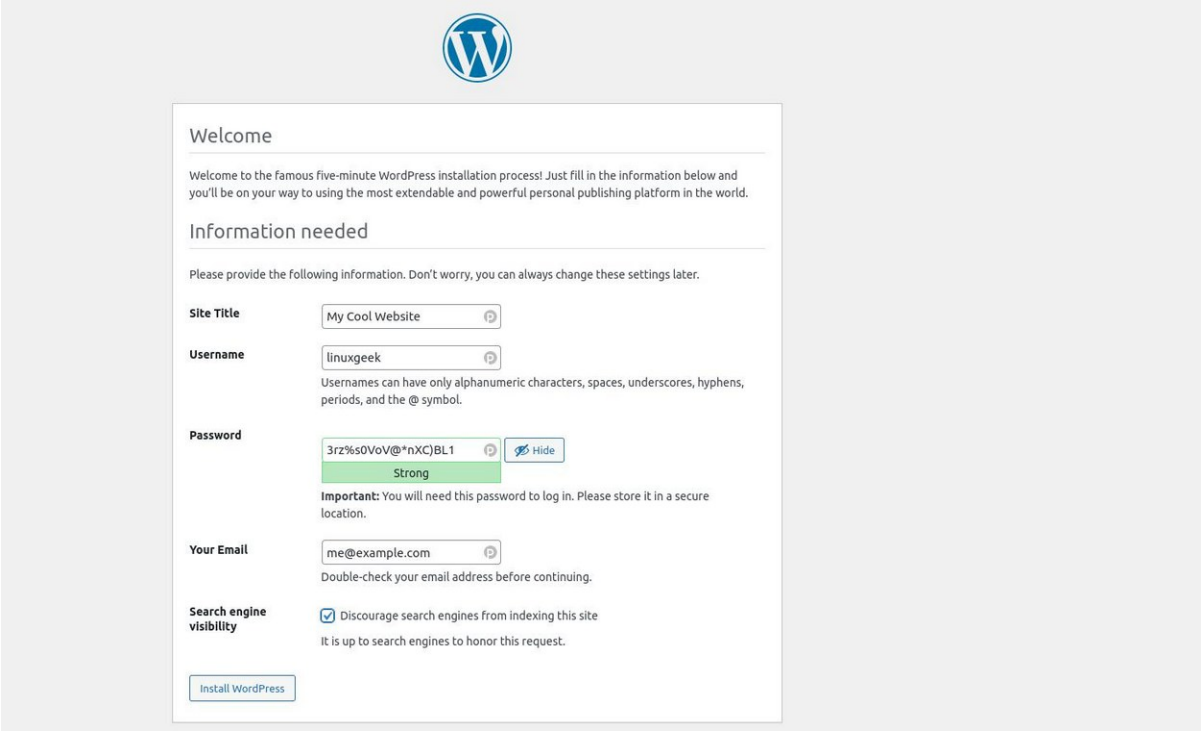
Next, launch the WordPress container into the same pod:

```
$ podman run --detach \
--restart=always --pod=wp_pod \
-e WORDPRESS_DB_NAME="wp_db" \
-e WORDPRESS_DB_USER="tux" \
-e WORDPRESS_DB_PASSWORD="badpassword1" \
-e WORDPRESS_DB_HOST="127.0.0.1" \

--name mypress wordpress
```

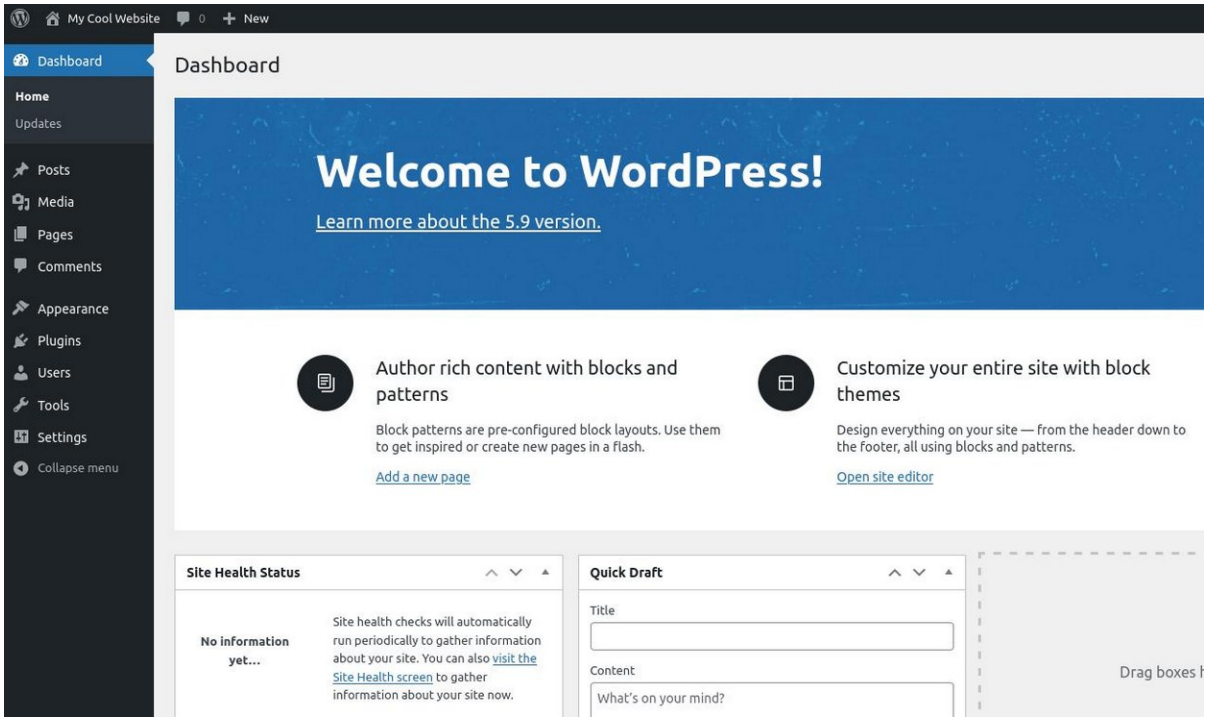
Now launch your favorite web browser and navigate to `localhost:8080`.

This time, the setup goes as expected. WordPress connects to the database because you've passed those environment variables while launching the container.



(Seth Kenlon, CC BY-SA 4.0)

After you've created a user account, you can log in to see the WordPress dashboard.



(Seth Kenlon, CC BY-SA 4.0)

Next steps

You've created two containers, and you've run them in a pod. You know enough now to run services in containers on your own server. If you want to move to the cloud, containers are, of course, well-suited for that. With tools like Kubernetes and OpenShift, you can automate the process of launching containers and pods on a cluster.

How I build my personal website using containers with a Makefile

By Chris Collins

The `make` utility and its related Makefile have been used to build software for a long time. The Makefile defines a set of commands to run, and the `make` utility runs them. It is similar to a Dockerfile or Containerfile—a set of commands used to build container images.

Together, a Makefile and Containerfile are an excellent way to manage a container-based project. The Containerfile describes the contents of the container image, and the Makefile describes how to manage the project itself: kicking the image build, testing, and deployment, among other helpful commands.

Make targets

The Makefile consists of "targets": one or more commands grouped under a single command. You can run each target by running the `make` command followed by the target you want to run. This command runs a target called `image_build`, defined in a Makefile:

```
$ make image_build
```

This is the beauty of the Makefile. You can build a collection of targets for each task that needs to be performed manually. In the context of a container-based project, this includes building the image, pushing it to a registry, testing the image, and even deploying the image and updating the service running it. I use a Makefile for my personal website to do all these tasks in an easy, automated way.

Build, test, deploy

I build my website using Hugo, a static website generator that builds static HTML from YAML files. I use Hugo to build the HTML files for me, then build a container image with those files and Caddy, a fast and simple web server, and run that image as a container. (Both Hugo and Caddy are open source, Apache-licensed projects.) I use a Makefile to make building and deploying that image to production much easier.

The first target in the Makefile is appropriately the `image_build` command:

```
image_build:
     podman build --format docker -f Containerfile -t $(IMAGE_REF):$(HASH) .
```

This target invokes Podman to build an image from the Containerfile included in the project. There are some variables in the command above—what are they? Variables can be specified in the Makefile, similarly to Bash or a programming language. I use them for a variety of things within the Makefile, but the most useful is building the image reference to be pushed to remote container image registries:

```
# Image values
REGISTRY := "us.gcr.io"
PROJECT := "my-project-name"
IMAGE := "some-image-name"
IMAGE_REF := $(REGISTRY)/$(PROJECT)/$(IMAGE)
# Git commit hash
HASH := $(shell git rev-parse --short HEAD)
```

Using these variables, the `image_build` target builds an image reference like `us.gcr.io/my-project-name/my-image-name:abc1234` using the short Git revision hash as the image tag so that it can be tied to the code that built it easily.

The Makefile then tags that image as `:latest`. I don't generally use `:latest` for anything in production, but further down in this Makefile, it will come in useful for cleanup:

```
image_tag:
     podman tag $(IMAGE_REF):$(HASH) $(IMAGE_REF):latest
```

So, now the image has been built and needs to be validated to make sure it meets some minimum requirements. For my personal website, this is honestly just, "does the webserver start and return something?" This could be accomplished with shell commands in the Makefile, but it was easier for me to write a Python script that starts a container with Podman, issues an

HTTP request to the container, verifies it receives a reply, and then cleans up the container. Python's "try, except, finally" exception handling is perfect for this and considerably easier than replicating the same logic from shell commands in a Makefile:

```
#!/usr/bin/env python3
import time
import argparse
from subprocess import check_call, CalledProcessError
from urllib.request import urlopen, Request
parser = argparse.ArgumentParser()
parser.add_argument('-i', '--image', action='store', required=True, help='image
name')
args = parser.parse_args()
print(args.image)
try:
    check_call("podman rm smk".split())
except CalledProcessError as err:
    pass
check_call(
    "podman run --rm --name=smk -p 8080:8080 -d {}".format(args.image).split()
)
time.sleep(5)
r = Request("http://localhost:8080", headers={'Host': 'chris.collins.is'})
try:
    print(str(urlopen(r).read()))
finally:
    check_call("podman kill smk".split())
```

This could be a more thorough test. For example, during the build process, the Git revision hash could be built into the response, and the test could check that the response included the expected hash. This would have the benefit of verifying that at least some of the expected content is there.

If all goes well with the tests, then the image is ready to be deployed. I use Google's Cloud Run service to host my website, and like any of the major cloud services, there is an excellent command-line interface (CLI) tool that I can use to interact with the service. Since Cloud Run is a container service, deployment consists of pushing the images built locally to a remote container registry, and then kicking off a rollout of the service using the `gcloud` CLI tool.

You can do the push using Podman or Skopeo (or Docker, if you're using it).

My push target pushes the `$(IMAGE_REF):$(HASH)` image and also the `:latest` tag:

```
push:
	podman push --remove-signatures $(IMAGE_REF):$(HASH)
	podman push --remove-signatures $(IMAGE_REF):latest
```

After the image has been pushed, use the `gcloud run deploy` command to roll out the newest image to the project and make the new image live. Once again, the Makefile comes in handy here. I can specify the `--platform` and `--region` arguments as variables in the Makefile so that I don't have to remember them each time. Let's be honest: I write so infrequently for my personal blog, there is zero chance I would remember these variables if I had to type them from memory each time I deployed a new image:

```
rollout:
	gcloud run deploy $(PROJECT) --image $(IMAGE_REF):$(HASH) --platform $
(PLATFORM) --region $(REGION)
```

More targets

There are additional helpful `make` targets. When writing new stuff or testing CSS or code changes, I like to see what I'm working on locally without deploying it to a remote server. For this, my Makefile has a `run_local` command, which spins up a container with the contents of my current commit and opens my browser to the URL of the page hosted by the locally running webserver:

```
.PHONY: run_local
run_local:
	podman stop mansmk ; podman rm mansmk ; podman run --name=mansmk --rm
-p $(HOST_ADDR):$(HOST_PORT):$(TARGET_PORT) -d $(IMAGE_REF):$(HASH) && $(BROWSER)
$(HOST_URL):$(HOST_PORT)
```

I also use a variable for the browser name, so I can test with several if I want to. By default, it will open in Firefox when I run `make run_local`. If I want to test the same thing in Google, I run `make run_local BROWSER="google-chrome"`.

When working with containers and container images, cleaning up old containers and images is an annoying chore, especially when you iterate frequently. I include targets in my Makefile for handling these tasks, too. When cleaning up a container, if the container doesn't exist, Podman or Docker will return with an exit code of 125. Unfortunately, `make` expects each command to return 0 or it will stop processing, so I use a wrapper script to handle that case:

```
#!/usr/bin/env bash
ID="${@}"
podman stop ${ID} 2>/dev/null
if [[ $?  == 125 ]]
then
  # No such container
  exit 0
elif [[ $? == 0 ]]
then
  podman rm ${ID} 2>/dev/null
else
  exit $?
fi
```

Cleaning images requires a bit more logic, but it can all be done within the Makefile. To do this easily, I add a label (via the Containerfile) to the image when it's being built. This makes it easy to find all the images with these labels. The most recent of these images can be identified by looking for the `:latest` tag. Finally, all of the images, except those pointing to the image tagged with `:latest`, can be deleted:

```
clean_images:
     $(eval LATEST_IMAGES := $(shell podman images --filter
"label=my-project.purpose=app-image" --no-trunc | awk '/latest/ {print $$3}'))
     podman images --filter "label=my-project.purpose=app-image" –no-trunc
--quiet | grep -v $(LATEST_IMAGES) | xargs --no-run-if-empty --max-lines=1 podman
image rm
```

This is the point where using a Makefile for managing container projects really comes together into something cool. To this point, the Makefile includes commands for building and tagging images, testing, pushing images, rolling out a new version, cleaning up a container, cleaning up images, and running a local version. Running each of these with `make image_build && make image_tag && make test`... etc. is considerably easier than running each of the original commands, but it can be simplified even further.

A Makefile can group commands into a target, allowing multiple targets to run with a single command. For example, my Makefile groups the `image_build` and `image_tag` targets under the `build` target, so I can run both by simply using `make build`. Even better, these targets can be further grouped into the default `make` target, `all`, allowing me to run all of them in order by executing `make all` or more simply, `make`.

For my project, I want the default `make` action to include everything from building the image to testing, deploying, and cleaning up, so I include the following targets:

```
.PHONY: all

all: build test deploy clean

.PHONY: build image_build image_tag

build: image_build image_tag

.PHONY: deploy push rollout

deploy: push rollout

.PHONY: clean clean_containers clean_images

clean: clean_containers clean_images
```

This does everything I've talked about in this article, except the `make run_local` target, in a single command: `make`.

Conclusion

A Makefile is an excellent way to manage a container-based project. By combining all the commands necessary to build, test, and deploy a project into `make` targets within the Makefile, all the "meta" work—everything aside from writing the code—can be simplified and automated. The Makefile can even be used for code-related tasks: running unit tests, maintaining modules, compiling binaries and checksums. While it can't yet write code for you, using a Makefile combined with the benefits of a containerized, cloud-based service can `make` (wink, wink) managing many aspects of a project much easier.

Podman: A more secure way to run containers

By Daniel J Walsh

Before I get into the main topic of this article, Podman and containers, I need to get a little technical about the Linux audit feature.

What is audit?

The Linux kernel has an interesting security feature called **audit**. It allows administrators to watch for security events on a system and have them logged to the audit.log, which can be stored locally or remotely on another machine to prevent a hacker from trying to cover his tracks.

The **/etc/shadow** file is a common security file to watch, since adding a record to it could allow an attacker to get return access to the system. Administrators want to know if any process modified the file. You can do this by executing the command:

```
# auditctl -w /etc/shadow
```

Now let's see what happens if I modify the /etc/shadow file:

```
# touch /etc/shadow
# ausearch -f /etc/shadow -i -ts recent
type=PROCTITLE msg=audit(10/10/2018 09:46:03.042:4108) : proctitle=touch
/etc/shadow
type=SYSCALL msg=audit(10/10/2018 09:46:03.042:4108) : arch=x86_64 syscall=openat
success=yes exit=3 a0=0xffffff9c a1=0x7ffdb17f6704 a2=O_WRONLY|O_CREAT|O_NOCTTY|
O_NONBLOCK a3=0x1b6 items=2 ppid=2712 pid=3727 auid=dwalsh uid=root gid=root
euid=root suid=root fsuid=root egid=root sgid=root fsgid=root tty=pts1 ses=3
comm=touch
exe=/usr/bin/touch subj=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
key=(null)
```

There's a lot of information in the audit record, but I highlighted that it recorded that root modified the /etc/shadow file and the owner of the process' audit UID (**auid**) was **dwalsh**.

Did the kernel do that?

Tracking the login UID

There is a field called **loginuid**, stored in **/proc/self/loginuid**, that is part of the proc struct of every process on the system. This field can be set only once; after it is set, the kernel will not allow any process to reset it.

When I log into the system, the login program sets the loginuid field for my login process.

My UID, dwalsh, is 3267.

```
$ cat /proc/self/loginuid
3267
```

Now, even if I become root, my login UID stays the same.

```
$ sudo cat /proc/self/loginuid
3267
```

Note that every process that's forked and executed from the initial login process automatically inherits the loginuid. This is how the kernel knew that the person who logged was dwalsh.

Containers

Now let's look at containers.

```
$ sudo podman run fedora cat /proc/self/loginuid
3267
```

Even the container process retains my loginuid. Now let's try with Docker.

```
$ sudo docker run fedora cat /proc/self/loginuid
4294967295
```

Why the difference?

Podman uses a traditional fork/exec model for the container, so the container process is an offspring of the Podman process. Docker uses a client/server model. The **docker** command I executed is the Docker client tool, and it communicates with the Docker daemon via a

client/server operation. Then the Docker daemon creates the container and handles communications of stdin/stdout back to the Docker client tool.

The default loginuid of processes (before their loginuid is set) is 4294967295. Since the container is an offspring of the Docker daemon and the Docker daemon is a child of the init system, we see that systemd, Docker daemon, and the container processes all have the same loginuid, 4294967295, which audit refers to as the *unset* audit UID.

```
cat /proc/1/loginuid
4294967295
```

How can this be abused?

Let's look at what would happen if a container process launched by Docker modifies the /etc/shadow file.

```
$ sudo docker run --privileged -v /:/host fedora touch /host/etc/shadow
$ sudo ausearch -f /etc/shadow -i
type=PROCTITLE msg=audit(10/10/2018 10:27:20.055:4569) :
proctitle=/usr/bin/coreutils
--coreutils-prog-shebang=touch /usr/bin/touch /host/etc/shadow
type=SYSCALL msg=audit(10/10/2018 10:27:20.055:4569) : arch=x86_64 syscall=openat
success=yes exit=3 a0=0xffffff9c a1=0x7ffdb6973f50 a2=O_WRONLY|O_CREAT|O_NOCTTY|
O_NONBLOCK a3=0x1b6 items=2 ppid=11863 pid=11882 auid=unset uid=root gid=root
euid=root suid=root fsuid=root egid=root sgid=root fsgid=root tty=(none)
ses=unset
comm=touch exe=/usr/bin/coreutils subj=system_u:system_r:spc_t:s0 key=(null)
```

In the Docker case, the auid is unset (4294967295); this means the security officer might know that a process modified the /etc/shadow file but the identity was lost.

If that attacker then removed the Docker container, there would be no trace on the system of who modified the /etc/shadow file.

Now let's look at the exact same scenario with Podman.

```
$ sudo podman run --privileged -v /:/host fedora touch /host/etc/shadow
$ sudo ausearch -f /etc/shadow -i
type=PROCTITLE msg=audit(10/10/2018 10:23:41.659:4530) :
proctitle=/usr/bin/coreutils
--coreutils-prog-shebang=touch /usr/bin/touch /host/etc/shadow
type=SYSCALL msg=audit(10/10/2018 10:23:41.659:4530) : arch=x86_64 syscall=openat
success=yes exit=3 a0=0xffffff9c a1=0x7fffdffd0f34 a2=O_WRONLY|O_CREAT|O_NOCTTY|
O_NONBLOCK a3=0x1b6 items=2 ppid=11671 pid=11683 auid=dwalsh uid=root gid=root
euid=root suid=root fsuid=root egid=root sgid=root fsgid=root tty=(none) ses=3
```

```
comm=touch
exe=/usr/bin/coreutils subj=unconfined_u:system_r:spc_t:s0 key=(null)
```

Everything is recorded correctly with Podman since it uses traditional fork/exec.

This was just a simple example of watching the /etc/shadow file, but the auditing system is very powerful for watching what processes do on a system. Using a fork/exec container runtime for launching containers (instead of a client/server container runtime) allows you to maintain better security through audit logging.

Final thoughts

There are many other nice features about the fork/exec model versus the client/server model when launching containers. For example, systemd features include:

- **SD_NOTIFY:** If you put a Podman command into a systemd unit file, the container process can return notice up the stack through Podman that the service is ready to receive tasks. This is something that can't be done in client/server mode.
- **Socket activation:** You can pass down connected sockets from systemd to Podman and onto the container process to use them. This is impossible in the client/server model.

The nicest feature, in my opinion, is **running Podman and containers as a non-root user**. This means you never have give a user root privileges on the host, while in the client/server model (like Docker employs), you must open a socket to a privileged daemon running as root to launch the containers. There you are at the mercy of the security mechanisms implemented in the daemon versus the security mechanisms implemented in the host operating systems—a dangerous proposition.

How to SSH into a running container

By Seth Kenlon

Containers have shifted the way we think about virtualization. You may remember the days (or you may still be living them) when a virtual machine was the full stack, from virtualized BIOS, operating system, and kernel up to each virtualized network interface controller (NIC). You logged into the virtual box just as you would your own workstation. It was a very direct and simple analogy.

And then containers came along, starting with LXC and culminating in the Open Container Initiative (OCI), and that's when things got complicated.

Idempotency

In the world of containers, the "virtual machine" is only mostly virtual. Everything that doesn't need to be virtualized is borrowed from the host machine. Furthermore, the container itself is usually meant to be ephemeral and idempotent, so it stores no persistent data, and its state is defined by configuration files on the host machine.

If you're used to the old ways of virtual machines, then you naturally expect to log into a virtual machine in order to interact with it. But containers are ephemeral, so anything you do in a container is forgotten, by design, should the container need to be restarted or respawned.

The commands controlling your container infrastructure (such as **oc, crictl**, **lxc**, and **docker**) provide an interface to run important commands to restart services, view logs, confirm the existence and permissions modes of an important file, and so on. You should use the tools provided by your container infrastructure to interact with your application, or else edit configuration files and relaunch. That's what containers are designed to do.

For instance, the open source forum software Discourse is officially distributed as a container image. The Discourse software is *stateless*, so its installation is self-contained within **/var/discourse**. As long as you have a backup of **/var/discourse**, you can always restore the

forum by relaunching the container. The container holds no persistent data, and its configuration file is **/var/discourse/containers/app.yml**.

Were you to log into the container and edit any of the files it contains, all changes would be lost if the container had to be restarted.

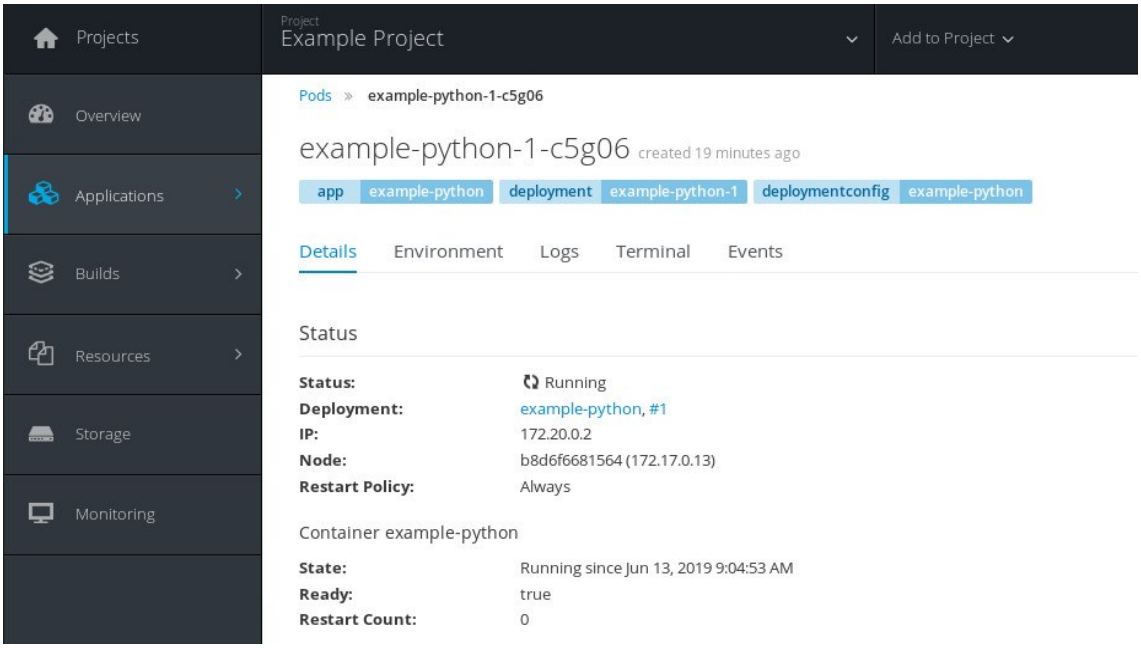
LXC containers you're building from scratch are more flexible, with configuration files (in a location defined by you) passed to the container when you launch it.

A build system like Jenkins usually has a default configuration file, such as **jenkins.yaml**, providing instructions for a base container image that exists only to build and run tests on source code. After the builds are done, the container goes away.

Now that you know you don't need SSH to interact with your containers, here's an overview of what tools are available (and some notes about using SSH in spite of all the fancy tools that make it redundant).

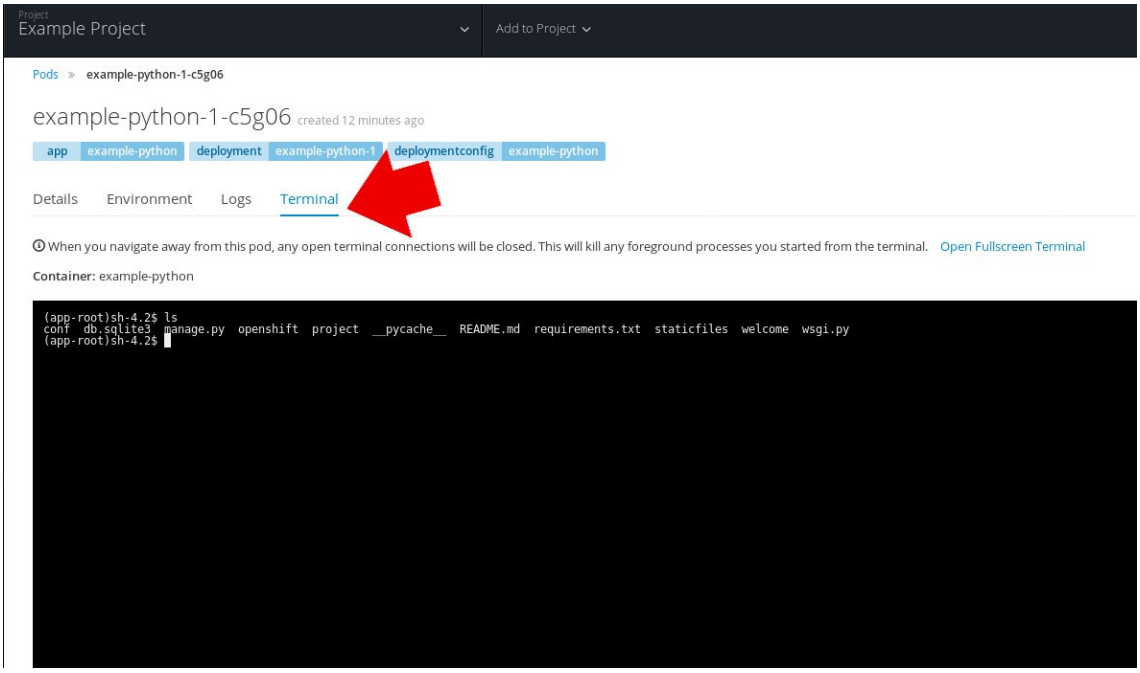
OpenShift web console

OpenShift 4 offers an open source toolchain for container creation and maintenance, including an interactive web console.



When you log into your web console, navigate to your project overview and click the **Applications** tab for a list of pods. Select a (running) pod to open the application's **Details** panel.

Click the **Terminal** tab at the top of the **Details** panel to open an interactive shell in your container.



If you prefer a browser-based experience for Kubernetes management, you can learn more through interactive lessons available at learn.openshift.com.

OpenShift oc

If you prefer a command-line interface experience, you can use the **oc** command to interact with containers from the terminal.

First, get a list of running pods (or refer to the web console for a list of active pods). To get that list, enter:

```
$ oc get pods
```

You can view the logs of a resource (a pod, build, or container). By default, **oc logs** returns the logs from the first container in the pod you specify. To select a single container, add the **--container** option:

```
$ oc logs --follow=true example-1-e1337 --container app
```

You can also view logs from all containers in a pod with:

```
$ oc logs --follow=true example-1-e1337 --all-containers
```

Execute commands

You can execute commands remotely with:

```
$ oc exec example-1-e1337 --container app hostname example.local
```

This is similar to running SSH non-interactively: you get to run the command you want to run without an interactive shell taking over your environment.

Remote shell

You can attach to a running container. This still does *not* open a shell in the container, but it does run commands directly. For example:

```
$ oc attach example-1-e1337 --container app
```

If you need a true interactive shell in a container, you can open a remote shell with the **oc rsh** command as long as the container includes a shell. By default, **oc rsh** launches **/bin/sh**:

```
$ oc rsh example-1-e1337 --container app
```

Kubernetes

If you're using Kubernetes directly, you can use the **kubetcl exec** command to run a Bash shell in your pod.

First, confirm that your pod is running:

```
$ kubectl get pods
```

As long as the pod containing your application is listed, you can use the **exec** command to launch a shell in the container. Using the name **example-pod** as the pod name, enter:

```
$ kubectl exec --stdin=false --tty=false example-pod -- /bin/bash
root@example.local:/# ls
bin  core etc   lib   oot  srv
boot  dev  home  lib64  sbin  tmp  var
```

Docker and Podman

The **docker** and **podman** commands are similar to **kubectl**. The Podman project doesn't require a daemon, while Docker requires **dockerd**. To get the name of a running container (you may have to use **sudo** to escalate privileges if you're not in the appropriate group), use either the **podman** or **docker** command (depending on which you've got installed):

```
$ podman ps
CONTAINER ID   IMAGE       COMMAND      NAME
678ac5cca78e    centos     "/bin/bash"   example-centos
```

Using the container name, you can run a command in the container:

```
$ docker exec example/centos cat /etc/os-release
CentOS Linux release 7.6
NAME="CentOS Linux"
VERSION="7"
ID="centos"
ID_LIKE="rhel fedora"
VERSION_ID="7"
[...]
```

Or you can launch a Bash shell for an interactive session:

```
$ podman exec -it example-centos /bin/bash
```

Containers and appliances

The important thing to remember when dealing with the cloud is that containers are essentially runtimes rather than virtual machines. While they have much in common with a Linux system (because they *are* a Linux system!), they rarely translate directly to the commands and workflow you may have developed on your Linux workstation. However, like appliances, containers have an interface to help you develop, maintain, and monitor them, so get familiar with the front-end commands and services until you're happily interacting with them just as easily as you interact with virtual (or bare-metal) machines. Soon, you'll wonder why everything isn't developed to be ephemeral.

Run containers on Linux without sudo in Podman

By Seth Kenlon

Containers are an important part of modern computing, and as the infrastructure around containers evolves, new and better tools have started to surface. It used to be that you could run containers with just LXC, and then Docker gained popularity, and things started getting more complex. Eventually, we got the container management system we all deserved with Podman, a daemonless container engine that makes containers and pods easy to build, run, and manage.

Containers interface directly with Linux kernel abilities like cgroups and namespaces, and they spawn lots of new processes within those namespaces. In short, running a container is literally running a Linux system *inside* a Linux system. From the operating system's viewpoint, it looks very much like an administrative and privileged activity. Normal users don't usually get to have free reign over system resources the way containers demand, so by default, root or `sudo` permissions are required to run Podman. However, that's only the default setting, and it's by no means the only setting available or intended. This article demonstrates how to configure your Linux system so that a normal user can run Podman without the use of `sudo` ("rootless").

Namespace user IDs

A kernel namespace is essentially an imaginary construct that helps Linux keep track of what processes belong together. It's the red queue ropes of Linux. There's not actually a difference between processes in one queue and another, but it's helpful to cordon them off from one another. Keeping them separate is the key to declaring one group of processes a "container" and the other group of processes your OS.

Linux tracks what user or group owns each process by User ID (UID) and Group ID (GID). Normally, a user has access to a thousand or so subordinate UIDs to assign to child processes in a namespace. Because Podman runs an entire subordinate operating system assigned to the user who started the container, you need a lot more than the default allotment of subuids and subgids.

You can grant a user more subuids and subgids with the `usermod` command. For example, to grant more subuids and subgids to the user `tux`, choose a suitably high UID that has no user assigned to it (such as 200,000) and increment it by several thousand:

```
$ sudo usermod \
--add-subuids 200000-265536 \
--add-subgids 200000-265536 \
tux
```

Namespace access

There are limits on namespaces, too. This usually gets set very high, but you can verify the user allotment of namespaces with `systctl`, the kernel parameter tool:

```
$ sysctl --all --pattern user_namespaces
user.max_user_namespaces = 28633
```

That's plenty of namespaces, and it's probably what your distribution has set by default. If your distribution doesn't have that property or has it set very low, then you can create it by entering this text into the file `/etc/sysctl.d/userns.conf`:

```
user.max_user_namespaces=28633
```

Load that setting:

```
$ sudo sysctl -p /etc/sysctl.d/userns.conf
```

Run a container without root

Once you've got your configuration set, reboot your computer to ensure that the changes to your user and kernel parameters are loaded and active.

After you reboot, try running a container image:

```
$ podman run -it busybox echo "hello"
hello
```

Containers like commands

Containers may feel mysterious if you're new to them, but actually, they're no different than your existing Linux system. They are literally processes running on your system, without the cost or barrier of an emulated environment or virtual machine. All that separates a container from your OS are kernel namespaces, so they're really just native processes with different labels on them. Podman makes this more evident than ever, and once you configure Podman to be a rootless command, containers feel more like commands than virtual environments. Podman makes containers and pods easy, so give it a try.

What is a container image?

By Nived Velayudhan

Containers are a critical part of today's IT operations. A *container image* contains a packaged application, along with its dependencies, and information on what processes it runs when launched.

You create container images by providing a set of specially formatted instructions, either as commits to a registry or as a Dockerfile. For example, this Dockerfile creates a container for a PHP web application:

```
FROM registry.access.redhat.com/ubi8/ubi:8.1
RUN yum --disableplugin=subscription-manager -y module enable php:7.3 \
  && yum --disableplugin=subscription-manager -y install httpd php \
 && yum --disableplugin=subscription-manager clean all
ADD index.php /var/www/html
RUN sed -i 's/Listen 80/Listen 8080/' /etc/httpd/conf/httpd.conf \
  && sed -i 's/listen.acl_users = apache,nginx/listen.acl_users =/' /etc/php-
fpm.d/www.conf \
  && mkdir /run/php-fpm \
  && chgrp -R 0 /var/log/httpd /var/run/httpd /run/php-fpm \
  && chmod -R g=u /var/log/httpd /var/run/httpd /run/php-fpm
EXPOSE 8080
USER 1001
CMD php-fpm & httpd -D FOREGROUND
```

Each instruction in this file adds a *layer* to the container image. Each layer only adds the difference from the layer below it, and then, all these layers are stacked together to form a read-only container image.

How does that work?

You need to know a few things about container images, and it's important to understand the concepts in this order:

1. Union file systems
2. Copy-on-Write
3. Overlay File Systems
4. Snapshotters

Union File Systems (Aufs)

The Union File System (UnionFS) is built into the Linux kernel, and it allows contents from one file system to be merged with the contents of another, while keeping the "physical" content separate. The result is a unified file system, even though the data is actually structured in branches.

The idea here is that if you have multiple images with some identical data, instead of having this data copied over again, it's shared by using something called a *layer*.

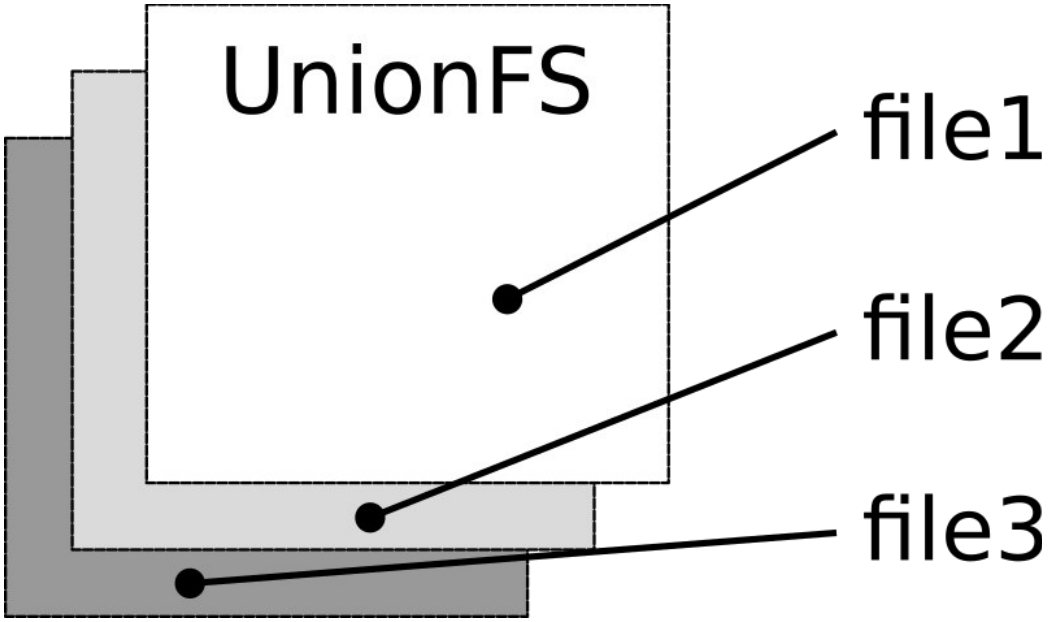


Image CC BY-SA opensource.com

Each layer is a file system that can be shared across multiple containers, e.g., The httpd base layer is the official Apache image and can be used across any number of containers. Imagine the disk space we just saved since we are using the same base layer for all our containers.

These image layers are always read-only, but when we create a new container from this image, we add a thin writable layer on top of it. This writable layer is where you create/modify/delete or make other changes required for each container.

Copy-on-write

When you start a container, it appears as if the container has an entire file system of its own. That means every container you run in the system needs its own copy of the file system. Wouldn't this take up a lot of disk space and also take a lot of time for the containers to boot? No—because every container does not need its own copy of the filesystem!

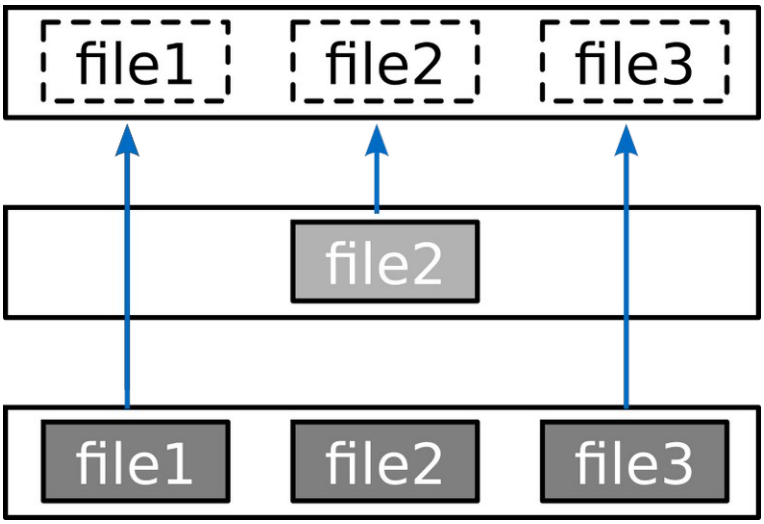
Containers and images use a copy-on-write mechanism to achieve this. Instead of copying files, the copy-on-write strategy shares the same instance of data to multiple processes and copies only when a process needs to modify or write data. All other processes would continue to use the original data. Before any write operation is performed in a running container, a copy of the file to be modified is placed on the writeable layer of the container. This is where the *write* takes place. Now you know why it's called *copy-on-write*.

This strategy optimizes both image disk space usage and the performance of container start times and works in conjunction with UnionFS.

Overlay File System

An overlay sits on top of an existing filesystem, combines an upper and lower directory tree, and presents them as a single directory. These directories are called *layers*. The lower layer remains unmodified. Each layer adds only the difference (the *diff*, in computing terminology) from the layer below it, and this unification process is referred to as a *union mount*.

The lowest directory or an Image layer is called *lowerdir*, and the upper directory is called *upperdir*. The final overlayed or unified layer is called *merged.*



Common terminology consists of these layer definitions:

- Base layer is where the files of your filesystem are located. In terms of container images, this layer would be your base image.
- Overlay layer is often called the *container layer*, as all the changes that are made to a running container, as adding, deleting, or modifying files, are written to this writable layer. All changes made to this layer are stored in the next layer, and is a *union* view of the Base and Diff layers.
- Diff layer contains all changes made in the Overlay layer. If you write something that's already in the Base layer, then the overlay file system copies the file to the Diff layer and makes the modifications you intended to write. This is called a *copy-on-write*.

Snapshotters

Containers can build, manage, and distribute changes as a part of their container filesystem using layers and graph drivers. But working with graph drivers is really complicated and is error-prone. SnapShotters are different from graph drivers, as they have no knowledge of images or containers.

Snapshotters work very similar to Git, such as the concept of having trees, and tracking changes to trees for each commit. A *snapshot* represents a filesystem state. Snapshots have parent-child relationships using a set of directories. A *diff can* be taken between a parent and its snapshot to create a layer.

The Snapshotter provides an API for allocating, snapshotting, and mounting abstract, layered file systems.

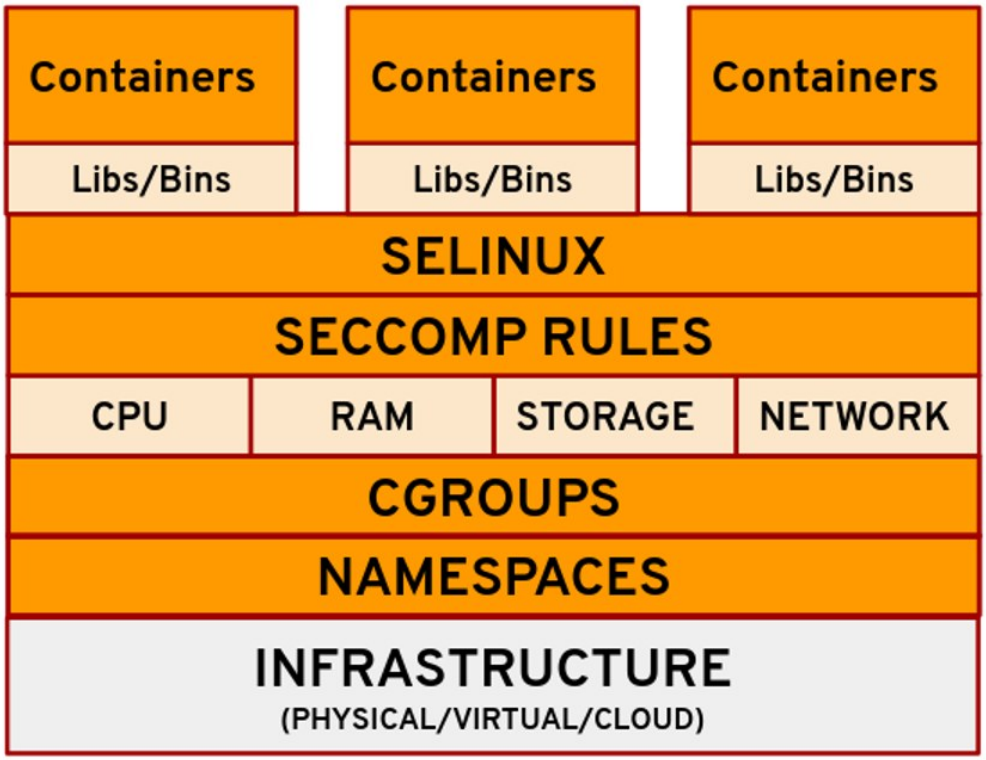
Wrap up

You now have a good sense of what container images are and how their layered approach makes containers portable. Next up, I'll cover container runtimes and internals.

4 Linux technologies fundamental to containers

By Nived Velayudhan

In previous articles, I have written about container images and runtimes. In this article, I look at how containers are made possible by a foundation of some special Linux technologies, including namespaces and control groups.



(Nived Velayudhan, CC BY-SA 4.0)

Linux technologies make up the foundations of building and running a container process on your system. Technologies include:

1. Namespaces
2. Control groups (cgroups)
3. Seccomp
4. SELinux

Namespaces

Namespaces provide a layer of isolation for the containers by giving the container a view of what appears to be its own Linux filesystem. This limits what a process can see and therefore restricts the resources available to it.

There are several namespaces in the Linux kernel that are used by Docker or Podman and others while creating a container:

```
$ docker container run alpine ping 8.8.8.8
$ sudo lsns -p 29413
        NS TYPE   NPROCS PID USER COMMAND
4026531835 cgroup   299   1  root /usr/lib/systemd/systemd --
switched...
4026533105 mnt 1 29413 root ping 8.8.8.8
4026533106 uts 1 29413 root ping 8.8.8.8
4026533105 ipc 1 29413 root ping 8.8.8.8
[...]
```

User

The user namespace isolates users and groups within a container. This is done by allowing containers to have a different view of UID and GID ranges compared to the host system. The user namespace enables the software to run inside the container as the root user. If an intruder attacks the container and then escapes to the host machine, they're confined to only a non-root identity.

Mnt

The mnt namespace allows the containers to have their own view of the system's file system hierarchy. You can find the mount points for each container process in the */proc/<PID>/mounts* location in your Linux system.

UTS

The Unix Timesharing System (UTS) namespace allows containers to have a unique hostname and domain name. When you run a container, a random ID is used as the hostname even when using the `--name` tag. You can use the `unshare` command to get an idea of how this works.

```
$ docker container run -it --name nived alpine sh
# hostname
9c9a5edabdd6
#
$ sudo unshare -u sh
# hostname isolated.hostname
# hostname
# exit
$ hostname
homelab.redhat.com
```

IPC

The Inter-Process Communication (IPC) namespace allows different container processes to communicate by accessing a shared range of memory or using a shared message queue.

```
# ipcmk -M 10M
Shared memory id: 0
# ipcmk -M 20M
Shared memory id: 1
# ipcs
---- Message Queues ----
key  msqid  owner  perms  used-bytes  messages
---- Shared Memory Segments
key        shmid owner perms bytes    nattch status
0xd1df416a 0     root  644   10485760 0
0xbd487a9d 1     root  644   20971520 0
[...]
```

PID

The Process ID (PID) namespace ensures that the processes running inside a container are isolated from the external world. When you run a `ps` command inside a container, you only see the processes running inside the container and not on the host machine because of this namespace.

Net

The network namespace allows the container to have its own view of network interface, IP addresses, routing tables, port numbers, and so on. How does a container able to communicate to the external world? All containers you create get attached to a special virtual network interface for communication.

Control groups (cgroups)

Cgroups are fundamental blocks of making a container. A cgroup allocates and limits resources such as CPU, memory, network I/O that are used by containers. The container engine automatically creates a cgroup filesystem of each type, and sets values for each container when the container is run.

SECCOMP

Seccomp basically stands for *secure computing*. It is a Linux feature used to restrict the set of system calls that an application is allowed to make. The default seccomp profile for Docker, for example, disables around 44 syscalls (over 300 are available).

The idea here is to provide containers access to only those resources which the container might need. For example, if you don't need the container to change the clock time on your host machine, you probably have no use for the *clock_adjtime* and *clock_settime* syscalls, and it makes sense to block them out. Similarly, you don't want the containers to change the kernel modules, so there is no need for them to make *create_module, delete_module* syscalls.

SELinux

SELinux stands for *security-enhanced Linux*. If you are running a Red Hat distribution on your hosts, then SELinux is enabled by default. SELinux lets you limit an application to have access only to its own files and prevent any other processes from accessing them. So, if an application is compromised, it would limit the number of files that it can affect or control. It does this by setting up contexts for files and processes and by defining policies that would enforce what a process can see and make changes to.

SELinux policies for containers are defined by the `container-selinux` package. By default, containers are run with the **container_t** label and are allowed to read (r) and execute

(x) under the */usr* directory and read most content from the */etc* directory. The label **container_var_lib_t** is common for files relating to containers.

Wrap up

Containers are a critical part of today's IT infrastructure and a pretty interesting technology, too. Even if your role doesn't involve containerization directly, understanding a few fundamental container concepts and approaches gives you an appreciation for how they can help your organization. The fact that containers are built on open source Linux technologies makes them even better!

What are container runtimes?

By Nived Velayudhan

In my examination of container images, I discussed container fundamentals, but now it's time to delve deeper into container runtimes so you can understand how container environments are built. The information in this article is in part extracted from the official documentation of the Open Container Initiative (OCI), the open standard for containers, so this information is relevant regardless of your container engine.

Container runtimes

So what really happens in the backend when you run a command like `podman run` or `docker run` command? Here is a step-by-step overview for you:

1. The image is pulled from an image registry if it not available locally
2. The image is extracted onto a copy-on-write filesystem, and all the container layers overlay each other to create a merged filesystem
3. A container mount point is prepared
4. Metadata is set from the container image, including settings like overriding CMD, ENTRYPOINT from user inputs, setting up SECCOMP rules, etc., to ensure container runs as expected
5. The kernel is alerted to assign some sort of isolation, such as process, networking, and filesystem, to this container (namespaces)
6. The kernel is also alerted to assign some resource limits like CPU or memory limits to this container (cgroups)
7. A system call (syscall) is passed to the kernel to start the container
8. SELinux/AppArmor is set up

Container runtimes take care of all of the above. When we think about container runtimes, the things that come to mind are probably runc, lxc, containerd, rkt, cri-o, and so on. Well, you are

not wrong. These are container engines and container runtimes, and each is built for different situations.

Container runtimes focus more on running containers, setting up namespace and cgroups for containers, and are also called lower-level container runtimes. Higher-level container runtimes or container engines focus on formats, unpacking, management, and image-sharing. They also provide APIs for developers.

Open Container Initiative (OCI)

The Open Container Initiative (OCI) is a Linux Foundation project. Its purpose is to design certain open standards or a structure around how to work with container runtimes and container image formats. It was established in June 2015 by Docker, rkt, CoreOS, and other industry leaders.

It does this using two specifications:

1. Image Specification (image-spec)

The goal of this specification is to enable the creation of interoperable tools for building, transporting, and preparing a container image to run.

The high-level components of the spec include:

- Image Manifest – a document describing the elements that make up a container image
- Image Index – an annotated index of image manifests
- Image Layout – a filesystem layout representing the contents of an image
- Filesystem Layer – a changeset that describes a container's filesystem
- Image Configuration – a document determining layer ordering and configuration of the image suitable for translation into a runtime bundle
- Conversion – a document explaining how this translation should occur
- Descriptor – a reference that describes the type, metadata, and content address of referenced content

2. Runtime specification (runtime-spec)

This specification aims to define the configuration, execution environment, and lifecycle of a container. The config.json file provides the container configuration for all supported platforms and details the field that enables the creation of a container. The execution environment is

detailed along with the common actions defined for a container's lifecycle to ensure that applications running inside a container have a consistent environment between runtimes.

The Linux container specification uses various kernel features, including namespaces, cgroups, capabilities, LSM, and filesystem jails to fulfill the spec.

Now you know

Container runtimes are managed by the OCI specifications to provide consistency and interoperability. Many people use containers without the need to understand how they work, but understanding containers is a valuable advantage when you need to troubleshoot or optimize how you use them.