Exercise CO.1 - Containerization as a DevOps Foundation

What’s a container?

At its core, a container is basically a glorified chroot. Chroot introduced way back in Version 7 Unix in 1979, and it let you change the root directory for a process. That means you could point a process to a different `/`, and it would think that’s the whole filesystem. Containers build on that idea with a bunch of extra isolation layers: namespaces (for things like network, PID, mount points) and cgroups (for resource limits like CPU and memory). So instead of just faking the root directory, you also fake the network stack, limit how much ram it can use, and so on. It’s like giving a process its own little sandbox that feels like a full system, but without the overhead of a fullblown vm.

Upsides of Containers vs VMS

VMs are still useful, but containers have some advantages:

- **Speed**: Spinning up a VM means booting an entire OS. Containers tart in seconds because they just launch a process with the bare minimum needed to runt that process, not a whole os.
- **Resource usage**: VMs each need their own kernel and OS. Containers share the host kernel, so they’re way lighter on ram and cpu.
- **Portability**: Moving a vm between systems is clunky. You’ve got to deal with hypervisor formats, disk images, etc. With containers, you just “docker pull” an image and “docker run” it.
- **Deployment**: Containers are made for ci/cd. You can build, test, and deploy in a clean environment every time. No “it works on my machine” nonsense.
- **Density**: You can run a ton of containers inside a single VM. That’s why people often use VMs as hosts and then run containers inside them.

That said, vms still win when you need full OS isolation or want to run different OSes (like Windows on a Linux host). But for most modern devops stuff, containers are the the way tot go.

Podman vs Docker

Both Docker and Podman let you build and run containers, but they go about it a bit differently.

- **Docker** uses a daemon that runs as root and manages containers for you. It uses containerd under the hood, which in turn uses runc to actually run containers.
- **Podman** is daemonless. When you run a container, it’s just a child process of your shell. No background service. It also uses runc (go language), or optionally crun (not by default though), which is faster and written in C.

Some differences:

- **Rootless by default**: Podman was designed with rootless containers in mind. Docker can do it too, but it’s not the default and takes more setup.
- **Pods**: Podman supports Kubernetes-style pods natively. You can define a group of containers that share networking and run them together just like in openshift or k8s.

- **System integration**: Podman can generate systemd unit files for containers, which is super useful for saving time on enterprise deployments where systemd is the way to go for non k8s docker deployments.
- **Ecosystem**: Docker has a bigger ecosystem dockerhub, compose, swarm, etc. Podman is catching up, especially in Red Hat and enterprise circles. (example if your company has a redhat license you get access to their images like ubi [universal base image] for stuff like jdk,node and so on)

In practice, they’re pretty compatible. You can alias docker to podman in you bash_aliases and most commands will just work. But if you care about rootless operation, tighter system integration, or avoiding daemons, podman’s a solid choice.

Sources:

https://www.youtube.com/watch?v=JOsWB50LmwQ

https://www.youtube.com/watch?v=sK5i-N34im8&t=2517s

work experience