Lesson #8 - Linux Containers

Advanced Linux Administration

Eduard Benes

Introduction

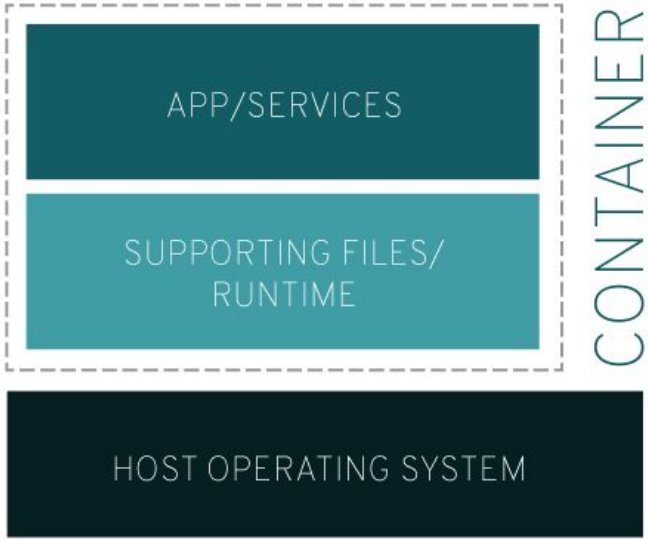
30 min

Containers on Linux

- Linux containers is technology that allow you to package and isolate applications with their entire runtime environment to have a small footprint and to be very lightweight.

Containers can help to address several problems

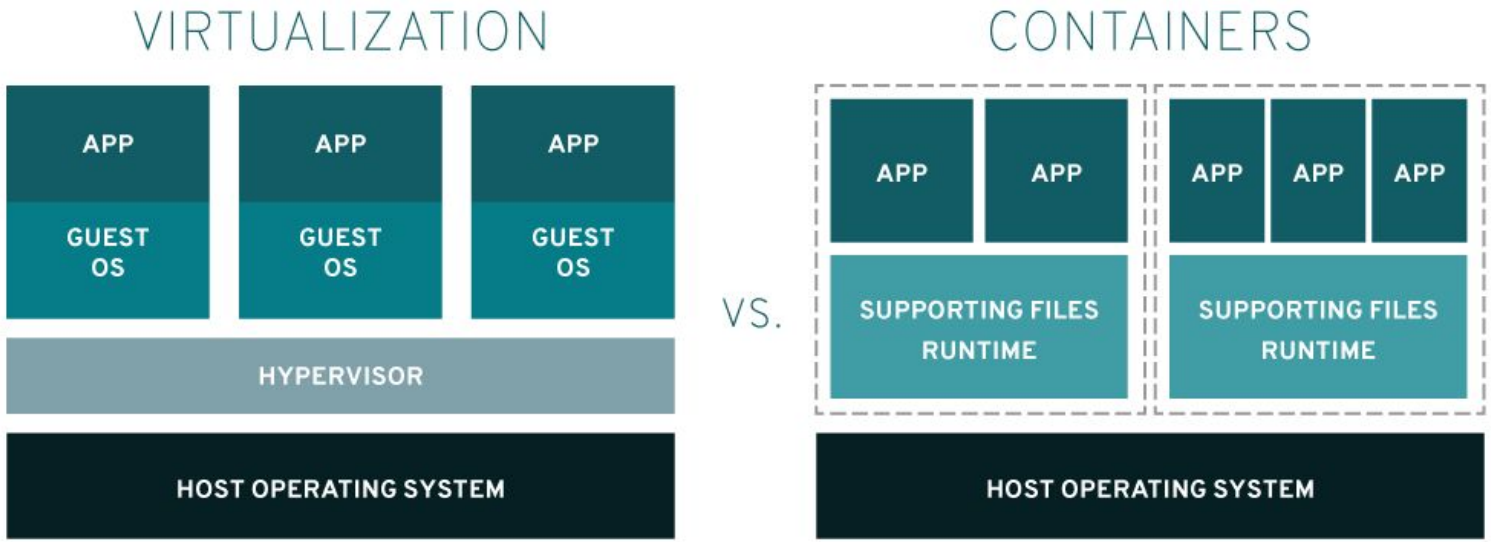
- *Reproducibility* - you can easily share your App+Runtime for debugging or from test to production (consistency)
- *Portability* - run your app across different distributions
- *Development Environment* - different library versions
- *Continuous integration / Testing* - quick usage and destroyal
- *Isolation* - don't trash or break your environment
- … can you come up with any other?



Note: In this course we will be using a container tool called `podman' instead of the more popular "docker".

Linux Containers Compared to Virtualization

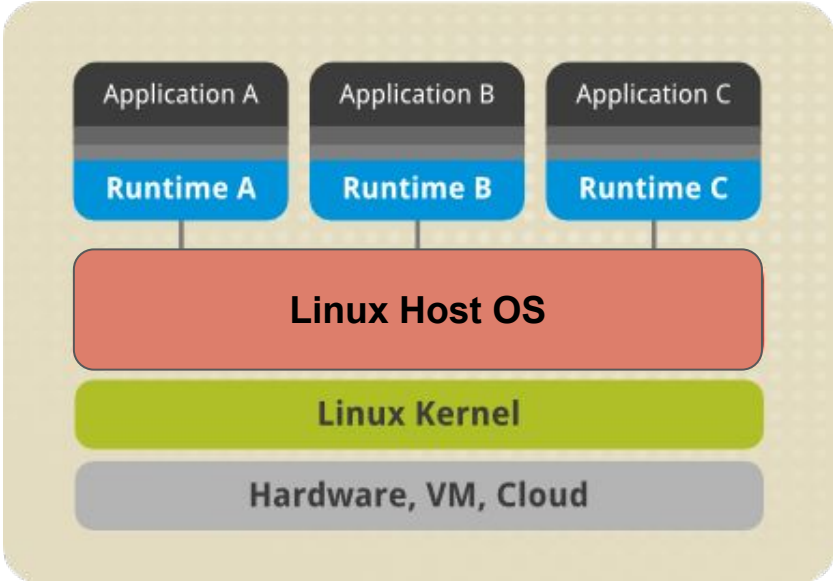
- Both have advantages and drawbacks that depend on the desired usage.
- Comparison by: HW, Kernel, isolation, resources, footprint, CPU, memory, storage, OS, security...
- Main difference —> Containers are meant to be LIGHTWEIGHT (Linux)



Overview of Container Host Environment

- Containers are isolated from each other, but share at least Kernel of a host system
- Management interface on a host system might differ depending on a specific Linux Containerization technology, e.g. Docker, LXC, Podman, …
- Each container has its own separate runtime environment

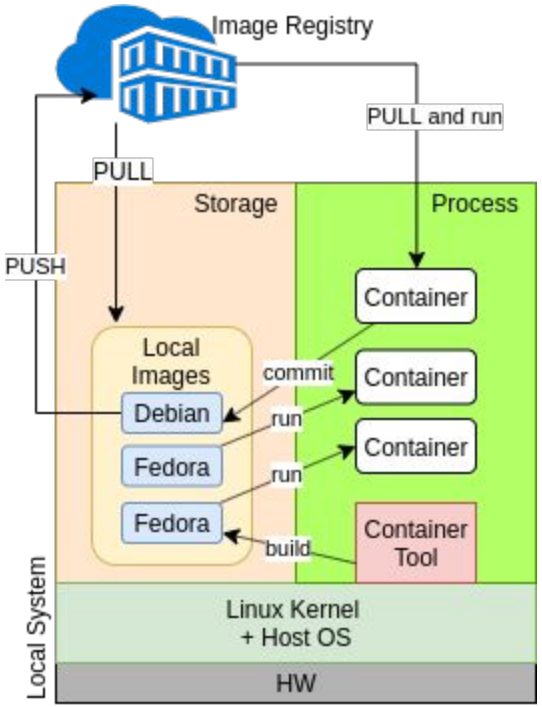
```
$ man podman ; man podman-run

$ podman create --rm --detach -ti fedora
$ podman start -l
$ podman attach -l    (Ctrl-P Ctrl-Q .. detach seq.)

$ podman run --detach --rm -ti fedora
$ podman exec -l uname -a

$ podman ps -a
```



Container, Image, and Registry

- **Image** - static read-only snapshot that is used for creation of a running container. It is never modified by running a container.
- **Container** - active instance of an image running some process inside, e.g. application or service. Changes are made just to a writable layer.
- **Registry -** are intended for distribution of images. They provide reliable, highly scalable, secured storage services for container images.
- “Dockerfile” - a recipe describing how a new image should be build.

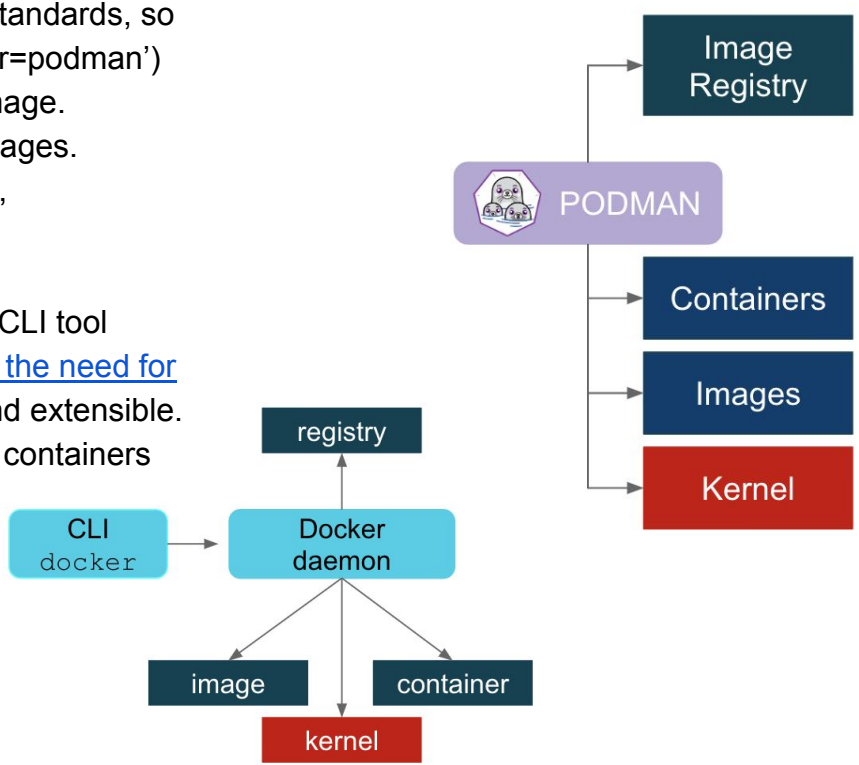
```
$ podman images
$ podman ps -a --ns

$ man podman-search
$ man podman-pull
$ podman pull alpine:latest
$ less /etc/containers/registries.conf

$ podman info
$ podman inspect -l
```



Podman VS Docker

- Both follow agreed OCI (Open Container Initiative) standards, so they are compatible (e.g. you could use `alias docker=podman')
  - image specification - describes a container image.
  - distribution specification - API to distribute images.
  - runtime specification - execution environment, configuration, and life-cycle of a container

- Docker has a daemon which is operated by `docker` CLI tool
- Podman approach is to interact directly by removing the need for a daemon, make the tooling and more lightweight and extensible.
- Images build by podman/buildah can be used to run containers using docker tool and vice-versa.

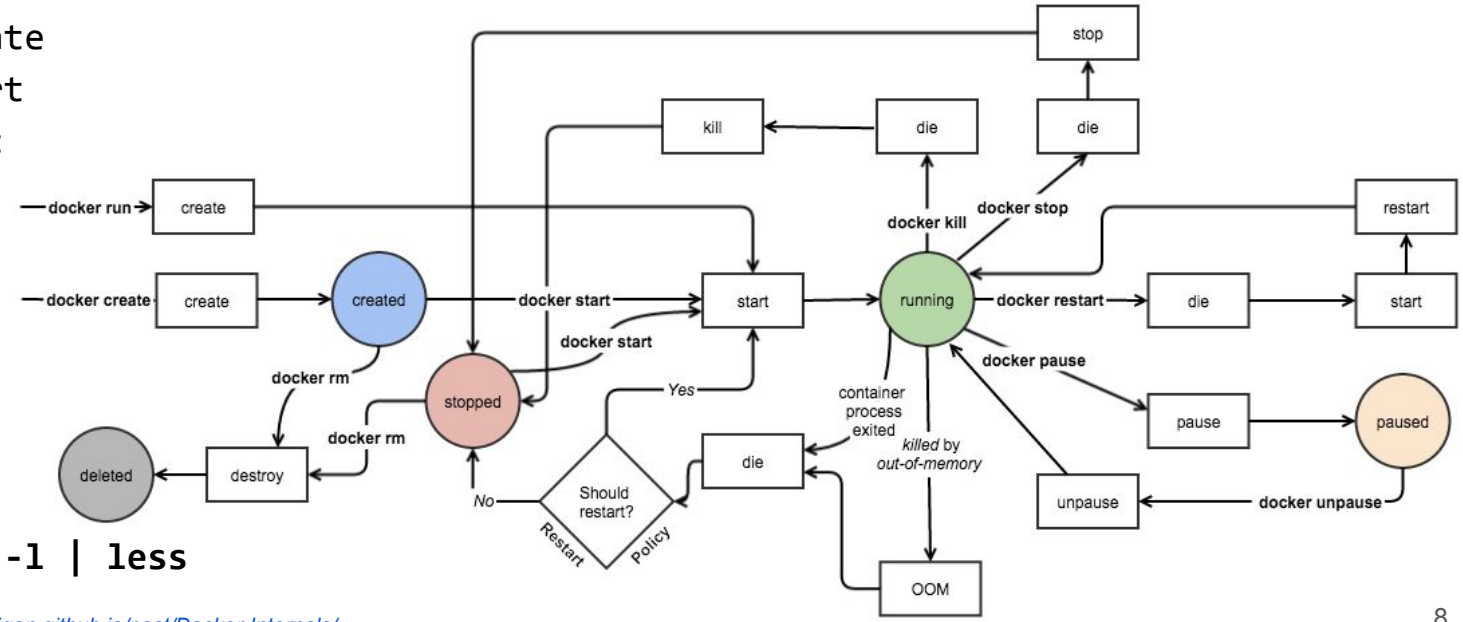



```
$ man podman ; podman tab-tab
$ man docker ; docker tab-tab
# systemctl status docker.service
```

Container Lifecycle (process states)

- Container (“created” from a static container image) can be in several states during its lifecycle.
- [Open Container Initiative runtime specification](http://docker-saigon.github.io/post/Docker-Internals/) makes sure to preserve compatibility.

```
$ man podman-create
$ man podman-start
$ man podman-exec
```



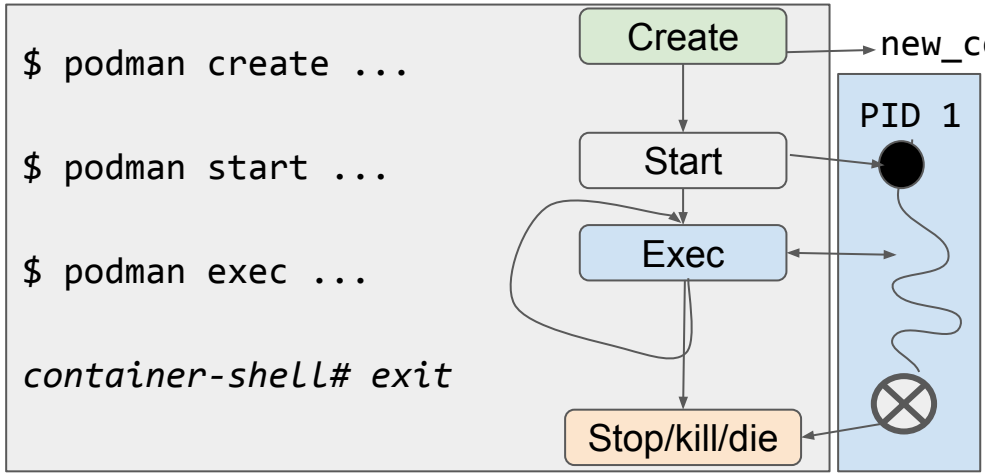

```
$ man podman-run
```

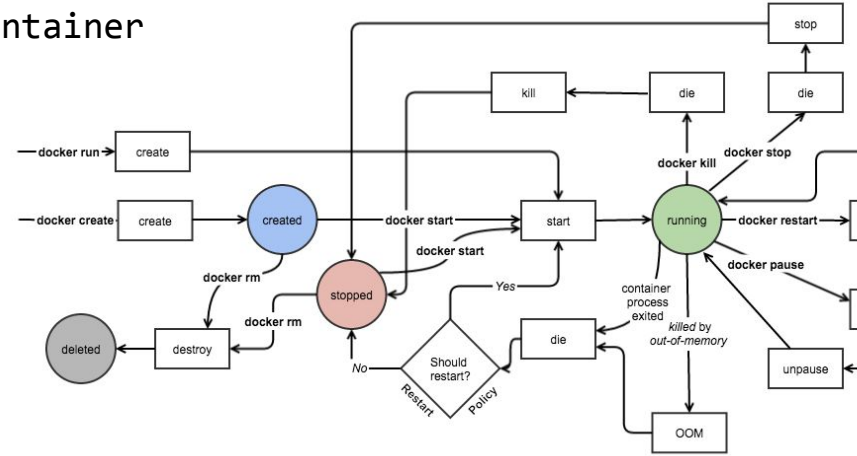
```
$ podman inspect -l | less
```

Image Source and Credits: http://docker-saigon.github.io/post/Docker-Internals/

Container Lifecycle (PID 1)

- Created container is not “running” yet, there is no process at host system.
- Command `podman start` will run a process representing PID 1 as configured (e.g. `/bin/bash`)
- Container process at host exits (representing PID 1) when the PID 1 in container exits.

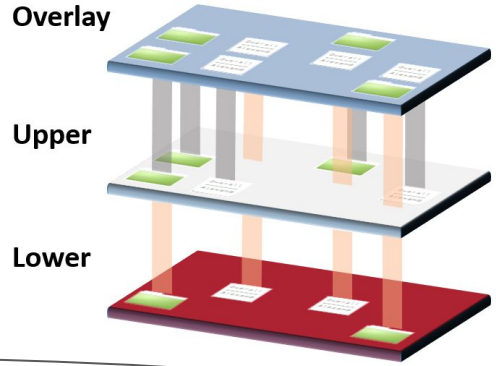


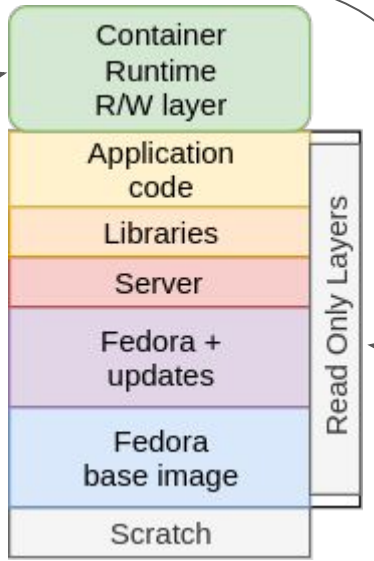


Example: When you exit a Bash process running in a container, the shell process end so also the container will stop and exit.

Anatomy of a Linux Container Image

- Container image usually consists of multiple image layers which are overlaid into a single coherent file system thanks to Union file system.
- When you create a new image from already existing container image, at least one new container image layer will be created.
- Image layers are *read-only* and are not changed by their running containers.
- Size of a final image can/should be optimized. For example by cleanup of data cached in a layer during build, or creating less layers by executing sequence of commands at once and thus create less image layers in final image.
- Creating a container from a container image mounts final (overlaid) root filesystem as *writable* allowing processes running in a container to make changes to this top layer.

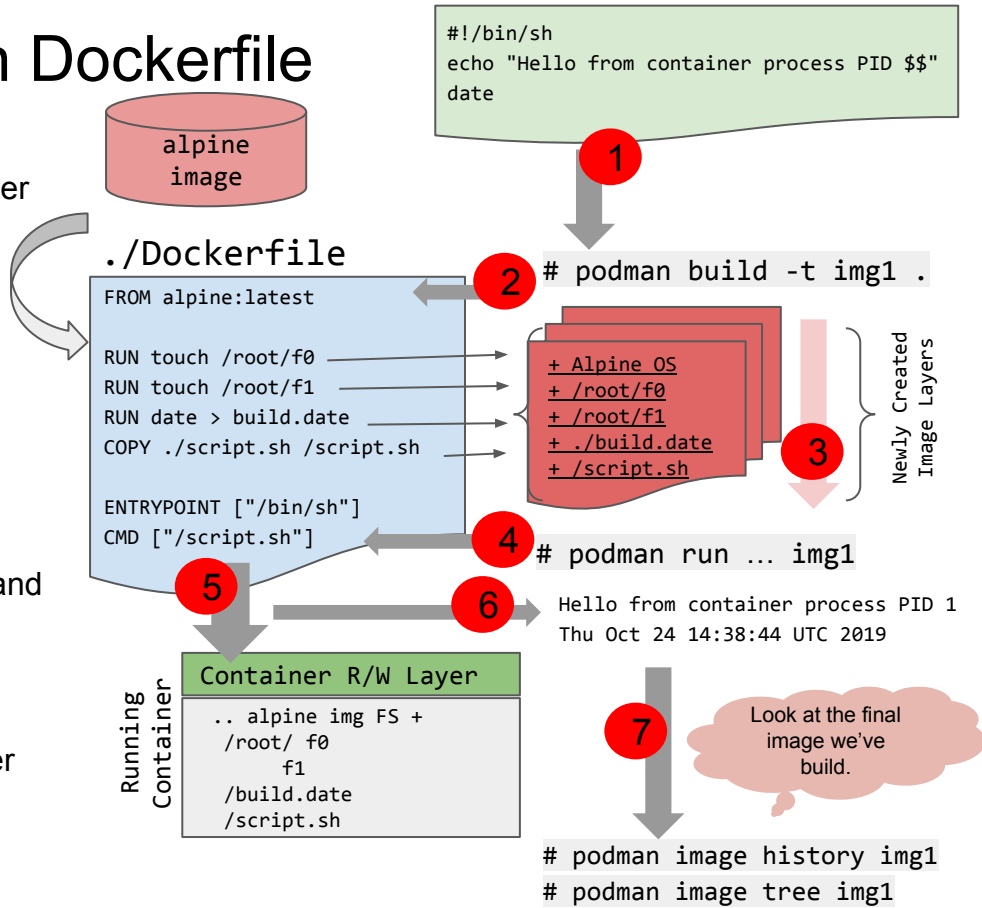





```
$ podman image history fedora
$ podman image tree fedora
$ podman image tree --whatrequires fedora

$ podman exec -l /bin/bash -c 'uname -a > /f1'
$ podman diff -l
```

Building container images with Dockerfile

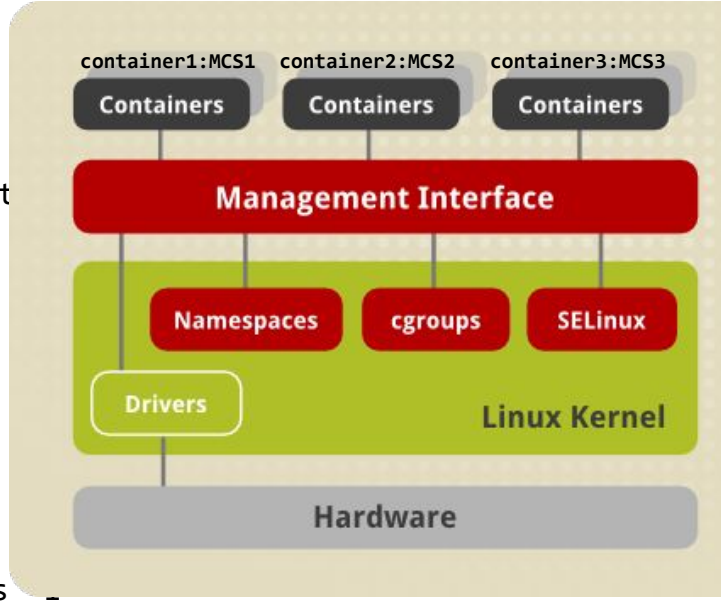
- Configuration file that automates creation of a container image is often still called Dockerfile
- Most common configuration instructions in Dockerfile:
  - `FROM, RUN, CMD, ENTRYPOINT`
- Other options useful options are:
  - `COPY, VOLUME, EXPOSE`
- `CMD vs ENTRYPOINT`
  - `CMD ..` will be executed only when you *run* container without specifying a command
  - `ENTRYPOINT ..` always executed, command and parameters are not ignored when Docker container runs with command line parameters
  - Shell VS exec form
- Note that some instructions will create a new img layer
  - `FROM, RUN, COPY, CMD`



Overview of Linux container primitives

Linux containers are created by using following technologies:

- Isolation - Linux Kernel Namespaces - Network, IPC, Mount, PID, UTS, User
- Constrained Resources - Control groups (cgroups)
- Restricted Capabilities
- Linux Kernel system call filtering - Seccomp
- Separation - Security Enhanced Linux (SELinux)
  - Types `container_t, container_file_t,` and `container_runt`
  - SELinux MCS category is generated for container process and its resources/files

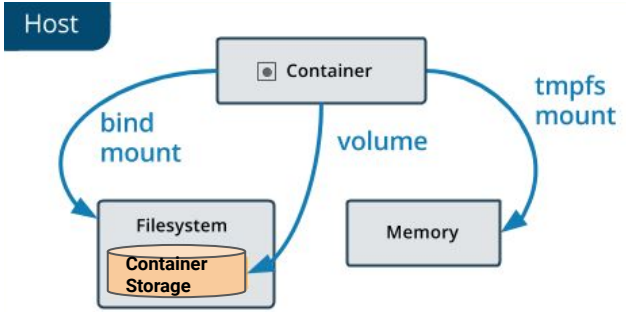



```
$ podman inspect -l | less  … search for {Pid, Caps, label}
$ podman ps -a --ns
$ pstree -pZ | grep container
$ podman info
$ ls -Z (on files found in output from inspect and info)
$ podman run --rm -ti alpine; podman exec -l ps ; podman ps --ns
```

Persistent Storage and Using Host Network Ports

- Containers don't have a persistent storage by default because they are meant to be lightweight!
  - Any changes made to the writable root filesystem of a running container (top RW layer) are lost when it exits.
- Container volumes are preferred solution for persistent data and shared data
  - **`--mount`** .. can attache a filesystem mount of type `{volume,bind,tmpfs}` to a container
  - **`--volume`** .. creates a bind mount of host dir to a container dir

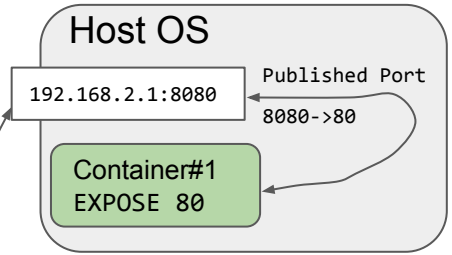
```
$ podman volume create my-new-volume
$ podman volume ls
```



- Exposing container ports with `EXPOSE` or **`--expose`** `PORT`
  - Tells podman to what port a container service can connect to.
  - It doesn't setup any network properties
  - Allows containers on the same network to talk to each other.
  - Expose doesn't publish the ports to the host system.
- Outside world ports with **`--publish`** `HOST:CONTAINER, --publish-all`
  - Ports are published by binding them to a host port, random or specified.
  - Can be used without a previously exposing a port in one step.

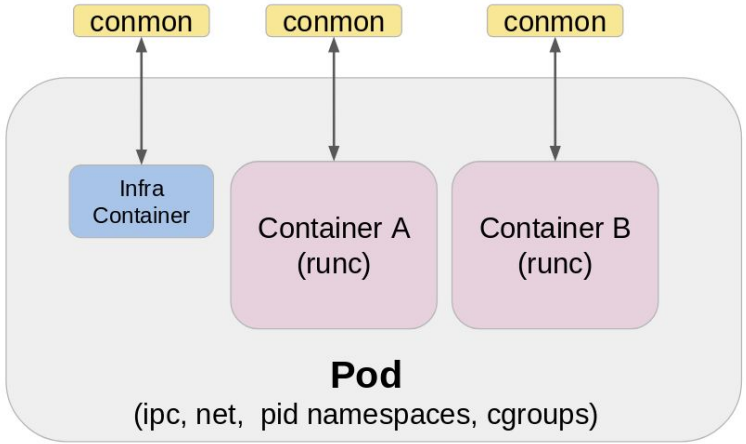
```
$ podman port -a
```

`192.168.2.1:8080`



Managing Group of Containers in a Podman Pods

- Podman pods are a group of several container sharing the same network, PID, and IPC namespaces.
- Using a `podman-pod` tool(command) you can manage such a group of containers together in a pod.
- Every podman pod has a unique container called “infra” container which holds shared namespaces associated with the pod and allow podman to connect to other containers in the pod.
- Other attributes assigned to the infra pod are:
  - Port bindings
  - Cgroup-parent values
  - Kernel namespaces (network, PID, IPC)




```
# man podman-pod
# podman pod ls
POD ID          NAME    STATUS   CREATED         # OF CONTAINERS   INFRA ID
61fbc1358d75    nc-pod   Running   6 days ago   3                 ea46fd70776b
```

```
# podman ps -a --pod
CONTAINER ID  IMAGE                                COMMAND  CREATED        STATUS          PORTS                    NAMES              POD
bbfc296fa538  docker.io/library/alpine:latest  /bin/sh    6 days ago  Up 6 days ago  0.0.0.0:8080->8080/tcp       client          61fbc1358d75
eacefe03782f  docker.io/library/alpine:latest  /bin/sh    6 days ago  Up 6 days ago  0.0.0.0:8080->8080/tcp       server          61fbc1358d75
ea46fd70776b  k8s.gcr.io/pause:3.1                        6 days ago  Up 6 days ago  0.0.0.0:8080->8080/tcp    61fbc1358d75-infra  61fbc1358d75
```

Workshop

60 min

Workshop labs

- In the following next slides there are 7 labs total, no bonus lab today ;-)
- Each lab has a time estimate how much time should you spend on it if everything goes well
- We encourage you to help each other or raise your hand to get help from lecturers
- If you couldn’t finish all labs during this class, you should complete them on your own later because learned skills will be used in following lectures or during a final practical exam.
- *HINT: focus on the lab content and leave any exploration or deep dive desires as a self-study for later*

Labs

1. Preparing work environment (5 min)
2. Obtaining images and running containers (10 min)
3. Temporary changes in a container (5min)
4. Commiting image of a modified container (10 min)
5. Exploring container isolation primitives (5 min)
6. Building a container image from Dockerfile (15 min)
7. Inspecting images, layers, diffs, logs (10 min)

Other labs are today optional and most likely will be used in some other lesson.

Lab 1 - Preparation (5 min)

- As root install package `podman`

```
# dnf -y install podman
# podman info
```

- Try to pull some images to have them locally

```
# podman pull fedora:latest
# podman pull alpine
```

- Lets run your first container as a root

```
# podman run --rm -ti fedora
container# uname -r
```

Try the Ctrl-P ctrl-Q sequence to detach

```
# podman attach -l
```

… type `exit` to quit container shell

When you run podman tool as a normal user, it is called “rootless”. It has its purpose, but because some actions are not allowed for normal user for security reasons, we are NOT going to use it in our labs.

*For the following labs, please make sure to run it **as root user.***

$ man podman-$COMMAND

*HINT: Sequence to **detach** from an attached container:*

Ctrl-P Ctrl-Q

Lab 2 - Obtaining images, running containers (10 min)

Purpose of this lab is to get familiar with `podman` tool, how to create and run application in a container.

- Search for an image: `# podman search alpine`
- Pull some image, e.g.: `# podman pull alpine:latest`
- Try other searching and filters options: `# man podman search`

- Create, start, and execute a new container named `new_toy` running command `` `uname -a` ''

```
# podman create --name new_toy -ti fedora
# podman start new_toy
# podman exec new_toy uname -a
```

- Kill the container and make sure `new_toy` is removed when finished. If not, fix it! :)

```
# podman kill new_toy ; podman rm new_toy ; podman ps -a
```

- Do the very same task again with just one-liner command:

```
# podman run --rm -ti --name new_toy alpine uname -a
```

Note: Observe that the "`uname -a`" is being passed to a shell process running in the `new_toy` container.

Lab 3 - Temporary changes in a container (5min)

Containers are lightweight and by default any changes made in a container are lost when it exits.

- Start a new container from image `fedora:latest (run --rm -ti)` and create in it a few new files `/root/file{1,2,3}` containing string `hello` and exit the container (so it stops and is removed).

```
# podman run --rm -ti --name c1 fedora
```

`container-shell#` … do the changes to it by creating some new files

Detach from the container using sequence **Ctrl-p Ctrl-q**

```
# podman diff c1
```

- Now attach again to the container and exit the shell

`# podman attach c1` … and inside of it in the shell type `exit`

```
container-shell# exit
```

Lab 4 - Commiting image of a modified container (10min)

After you make some changes to a running container which you don't want to lose (e.g. configuration) you can commit (save) the whole container as a new container image so you can use it to run new containers later.

- Run a new `fedora:latest` based container named `c1` again. Are there the 3 files created before? Hopefully not :)
- Install a new `psmisc` package into the container: `# dnf -y install psmisc`
- Demonstrate psmisc is present by listing processes in the container with `# pstree -p`
- **Detach** from the container (**ctrl-P ctrl-Q**), and make sure to KEEP the container RUNNING
- Confirm that the c1 container is still running with `# podman ps -a`
- Now we will pause and commit the container as a new container image named `pstree-img` so we can use it later

- Make sure the `c1` is running and that you are executing this a as a root at your container host system.

`# podman pause c1 ; podman ps -l`

`# podman commit c1 pstree-img` → later you can try to use also some other available options, which might be useful

`# podman images` → your newly created image should be listed

`# podman unpause c1`

- Finally run a fresh-new container from the commited new image `pstree-img` and demonstrate the package (`psmisc` RPM) is installed and your changes are also still present in the newly started container.

Lab 5 - Exploring container isolation (5 min)

Start few detached containers from `pstree-img` image on background so we can explore them.

- On host system list IDs of Linux kernel namespaces for the running containers and note which are unique and which are shared? What is the PID of the running container on the host system?

```
# podman ps --ns -a
```

- On host system observe SELinux isolation represented as different MCS labels of the running containers for the process running as PID 1:

```
# pstree -Zp | grep -1 container_t
```

- Let's look at differences of the same process in and outside of a container from `c1-img`

```
# pstree -SapZ | grep $PID_OF_CONTAINER
...
container# $ pstree -SapZ
```

→ Note: if you haven't used the `pstree-img to create this container,` you might need to install `psmisc` packages to get `pstree` command

- Now try to check whether and how is SELinux used inside of a `pstree-img` based container

```
container# ls -lZ /
```

- On a host you can try to list context of some file used by a container and list SELinux context:

```
# ls -lZ `podman inspect -l --format "{{.HostnamePath}}"`
```

- By exploring output of the inspect command you can also look at other values, for example
  - `MountLabel, ProcesLabel`
  - `EffectiveCaps`

Lab 6 - Building a container image from Dockerfile (15min)

- Create a new Dockerfile with following content, built it as `my_image`

```
# cat Dockerfile
    FROM fedora:latest
    RUN touch /f0 ; touch /f1
    RUN dnf -y install nmap-ncat
    # && dnf clean all
    RUN date > build.date
    CMD ["/bin/sh"]
```

→ This will be different to default CMD target in `fedora:latest` base image

```
# podman build -t my_image -f ./Dockerfile
```

- Check it is really there as a new image and run it while printing build time from `/build.date`

```
# podman images
# podman run --rm -ti my_image cat /build.date
```

- Try passing an environment variables to the container with --env option (or ENV in Dockerfile)

```
# podman run --rm -ti --env MY_ENV=hello fedora printenv MY_ENV
```

… now you can try and spend few minutes experimenting with `Dockerfile` commands on your own.

Lab 7 - Inspecting images, layers, diffs, logs (10min)

- First look at size of an official image fedora:latest you've pulled from registry

```
# podman image tree fedora:latest
# podman history fedora:latest
# podman image tree --whatrequires fedora:latest
```

- Explore the custom build image `my_image` from previous lab and its size

```
# podman image tree my_image ; # podman history my_image
```

- Now compare final size of both `my_image` and `fedora:latest` , the custom image is larger.
- We can do better. For example cleaning cached content and/or creating less layers.
- Try to optimize size of the final image by cleaning DNF cache (e.g. `dnf clean all`) when installing new packages in `Dockerfile` from previous lab, re-build it and confirm it has as a smaller image.
- Look at content of changes in a running container or image layer compared to its parent layer

```
# podman diff --format json some_container_or_image_name
```

- Run a container and execute some commands in its shell, look at command history in logs

```
# podman logs container
```

End of Lesson #8 - Linux Containers

Unfortunately we don’t have enough time to cover all topics today.

Following extra labs will be most likely used in a separate lesson related again to Linux containers.

You can start experimenting with this labs already and start mastering Linux containers already now!

~~Lab 8 - Volume mounts to share and preserve data ½ (10min)~~

Container are meant to be lightweight. Persistent storage, sharing, or preserving data can be done using volume mounts.

- Clone some git repository and make it available in a container

```
# mkdir -p /data/path/to/git/ ; cd /data/path/to/git
# git clone https://github.com/opencontainers/runtime-spec.git
# podman run -ti --rm --volume=/data/path/to/git/:/shares/:rw,z fedora
```

    - Change some files in the git from the container; Is it changed also on the host system?
- Next create two container volumes

```
# podman volume create
# podman volume create myvol
```

- List available volumes on your system

```
# podman volume ls
```

~~Lab 8 - Volume mounts to share and preserve data 2/2 (10m)~~

- Now, lets ty to write directly to the volume from host and access it from container:

`# podman run --rm -ti --volume=myvol:/shares/ fedora`

`container# ls /shares` → do some changes to the folder, e.g. create some new files

`# podman volume inspect myvol`

- … copy the location of volume data (“`Mountpoint`”) and create some file in the path on host system

`# date > ..your..path..to..the..file.../f1`

`# podman run --rm -ti --volume=myvol:/shares/ fedora`

`container# cat /shares/f1` → have you find a correct date in it?

~~Lab 9 - Expose containers to the network (5min)~~

- We will create a simple chat-like scenario using command `nc` between two `alpine` containers
- Run container listening as server at port 8080 and on host bound to some high free port 8080 or 8088

```
# podman run --rm -ti -p 8080:8080 -d --name server alpine
# podman run --rm -ti -d --name client alpine
```

- Attach to the server and get its IP address (`# ip addr`) and try to ping the server IP from client
- Attach to the server and get its IP address (`# ip addr`) and try to ping the server IP from client

```
server# nc -l -p 8080
client# ping server.ip.address
client# nc server.ip.address 8080
```

… and start chatting ….

~~Lab 10 - Running multiple containers in a pod 1/2 (10min)~~

Create a podman pod with 4 containers in the pod demonstrating usage and features of a pod.

- Create a new pod named `nc-pod` and observe what has been created.

```
# podman pod create -n nc-pod
```

`# podman ps -a --pod` → you should see a container named `XXXXX-infra` running in the pod?

```
# podman pod ls
```

`# podman inspect 61fbc1358d75-infra | less` → inspect content of a Network section, is there an IPAddr address value? Shouldn't be

- Run 3 containers to act as netcat servers

```
# podman run --rm -ti -d --pod nc-pod --name server1 alpine nc -l -p 8080
# podman run --rm -ti -d --pod nc-pod --name server2 alpine nc -l -p 8888
# podman run --rm -ti -d --pod nc-pod --name server3 alpine nc -l -p 9999
```

- Start 3rd container which you will use as a client

```
# podman run --rm -ti -d --pod nc-pod --name client alpine
```

~~Lab 10 - Running multiple containers in a pod 2/2 (10min)~~

Now check what we have running in the pod and what is the IP address of the pod.

```
# podman ps -a --pod
# podman inspect 61fbc1358d75-infra | less → check value of IPAddress, should be like 10.88.0.xx
```

Now establish connection from the client container in the nc-pod over localhost and from host system using IPAddress of infra container.

```
client# nc localhost 8888
client# nc 10.88.0.xx 9999
host# nc 10.88.0.xx 8080
```

Stop containers and pod, make sure to properly clean up after the pods:

```
# podman pod stop nc-pod
# podman pod rm nc-pod
```

~~Lab #11- Pod containers bind to a host ports (5min)~~

BONUS: Recreate a new pod named `nc-public` which would allow you to connect to the `server1` and `server2` running in the pod `nc-public` from host system using IP address of the host system (or localhost), because it'll be binded to host ports `8080` and `8888`

Test #1

```
host# nc host.ip.address 8080
```

Test #2

```
host# nc host.ip.address 8888
```

Hint: `# podman port -a`

Misc Links and Resources

- Libpod github
- Managing podman pods
- Container development kit
- Docs for Containes in RHEL 7 Atomic Host
- RHEL 8 container docs
- Minishift, some workshop
- Quay.io
- Containers_101_with_Podman_on_Fedora29
- Docker internals
- Run Red Hat Enterprise Linux 8 in a container on RHEL 7
- Managing containerized system services with Podman
- Using systemd with podman containers
- Podman and user namespace