표준 오픈소스 컨테이너 101

STD CONTAINER

소개

Podman

Buildah

Skopeo

설치

2023-05-30

:^)<-<

포드만

현재 오픈소스 컨테이너 엔진은 포드만으로 전환 중. docker-ee, docker-ce는 여전히 사용이 가능하나, 오픈소스 표준 엔진 및 런타임을 제공하는 포드만을 권장. 포드만은 다음과 같은 표준형식을 제공한다.

- OCI
- CNI
- CRI

도커는 도커에서 사용하는 관리 방법 및 구조를 사용하고 있기 때문에, OCI형식으로 이미지는 제공하지만, 네트워크 및 런타임 인터페이스는 도커 자체 구조를 사용한다. 이 부분에 대해서는 뒤에서 더 다루도록 한다.

포드만

포드만 사용 이유는 다음과 같다.

1. 쿠버네티스에서 사용하는 이미지 빌드
2. 표준 런타임 기반으로 컨테이너 애플리케이션 테스트
3. Pod기반으로 구성하여 쿠버네티스에서 동작이 가능한지 확인
4. systemd의 service형태로 컨테이너 구현을 원하는 경우

기존 도커 런타임은 위와 같은 기능을 제공하지 않는다.

빌더(BUILDAH)

빌더(Buildah)는 오픈소스 표준 이미지 빌드 도구.

기존에는 'docker build'를 사용하여 구성 및 구현하였지만, 이미지 빌드 용도(전용)로 사용하기에는 너무 무겁기 때문에, 이미지 빌드 도구로 나옴. 기존 **Dockerfile**과 **Containerfile**를 둘 다 지원한다.

CI/CD에서 이미지 빌드 구성 및 생성 용도로 많이 사용하는 단일 도구이다. **Buildah**의 발음은 **빌더**이며, **"빌드-아"** 라고 읽는 경우도 있다.

SKOPEO

도커에서 이미지 검색 부분을 분리하여 독립적인 도구로 만든 명령어. 이미지 검색은 oci, docker프로토콜을 지원한다. 현재는 docker프로토콜만 원격 HTTP/V2를 제공한다.

1. oci
2. oc-archive
3. docker-daemon
4. docker-archive
5. docker
6. dir
7. container-storages

사용 방법은 뒤에서 다루도록 한다.

PODMAN

설치

설치

포드만은 기본적으로 대다수 리눅스 배포판에서 제공하고 있다. 다만, 레드햇 계열 경우에는 확장 기능을 사용하기 위해서 다음과 같은 저장소를 구성한다.

- epel-release

```
# dnf search epel-release
# dnf install epel-release
# dnf repolist
> epel
# dnf search podman
> podman-tui
> podman-remote
> podman-compose
> podman-docker
```

설치

API서비스가 필요한 경우, "podman.service"시작이 필요하며, 그렇지 않는 경우 'podman'명령어만 사용하여 된다.

```
# systemctl enable --now podman.service
# podman ps
```

컨테이너 기술

기술설명

루트리스/루트풀

기반기술

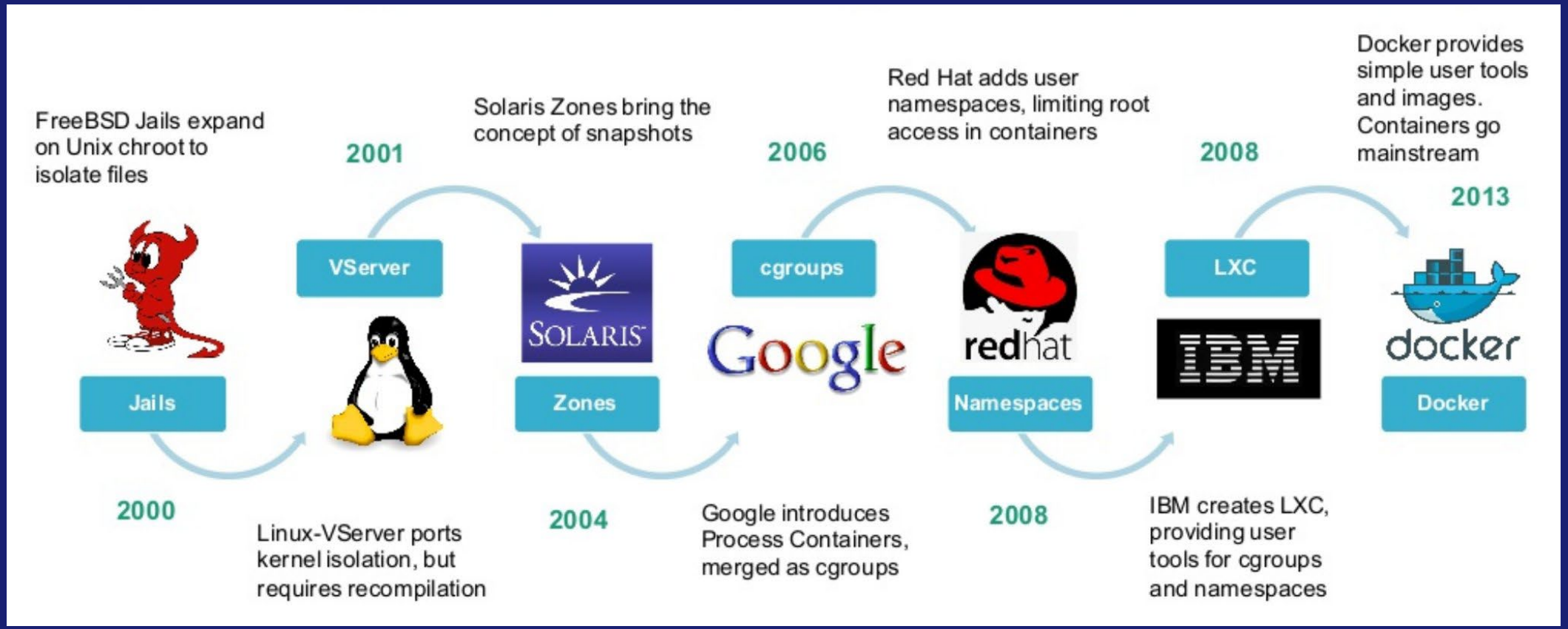
런타임 엔진

컨테이너 기술

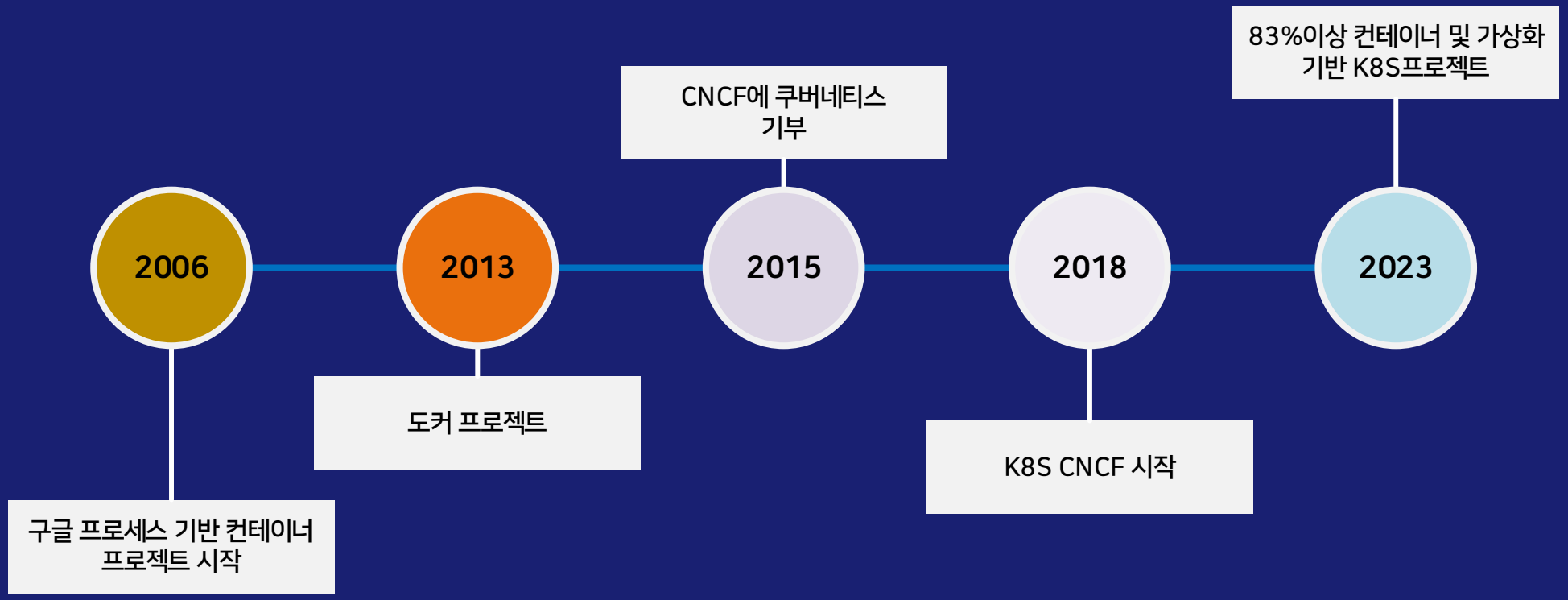
기술설명

2023-05-30

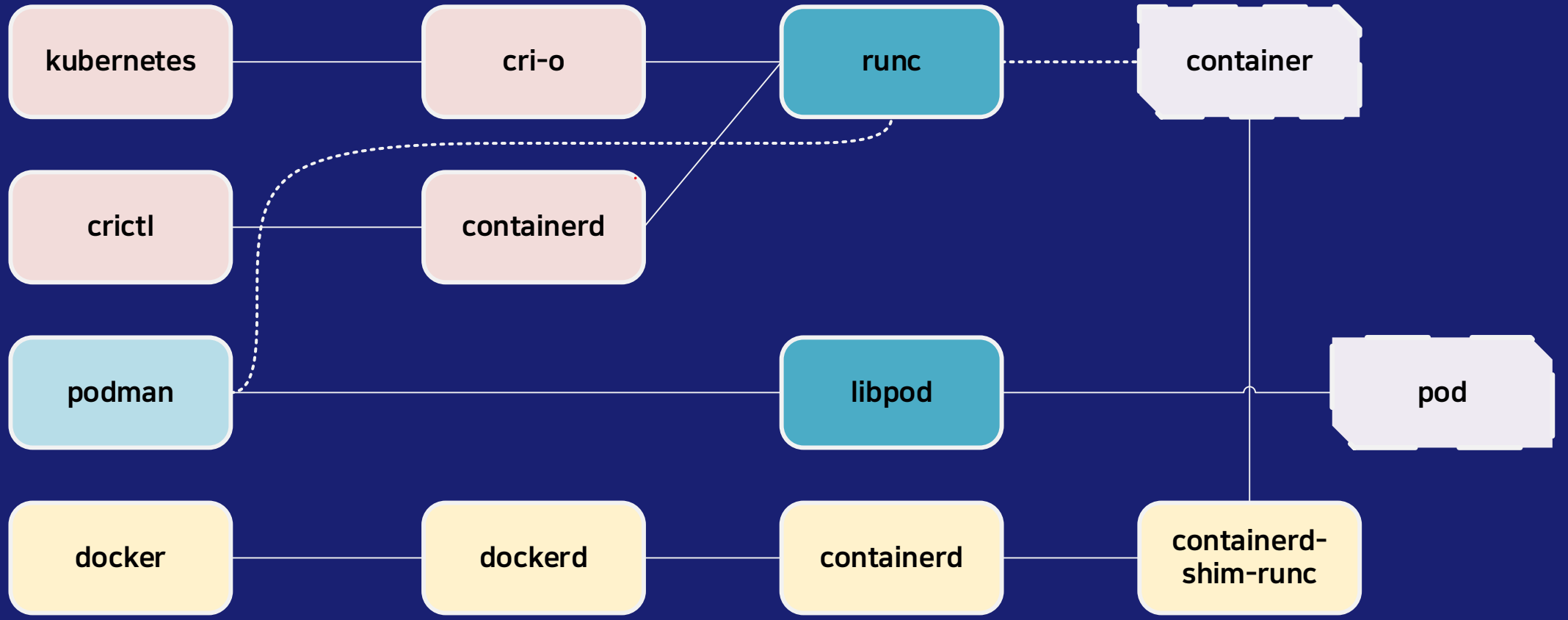
RUNTIME HISTORY



KUBERNETES HISTORY



LOW/HIGH RUNTIME



기반기술

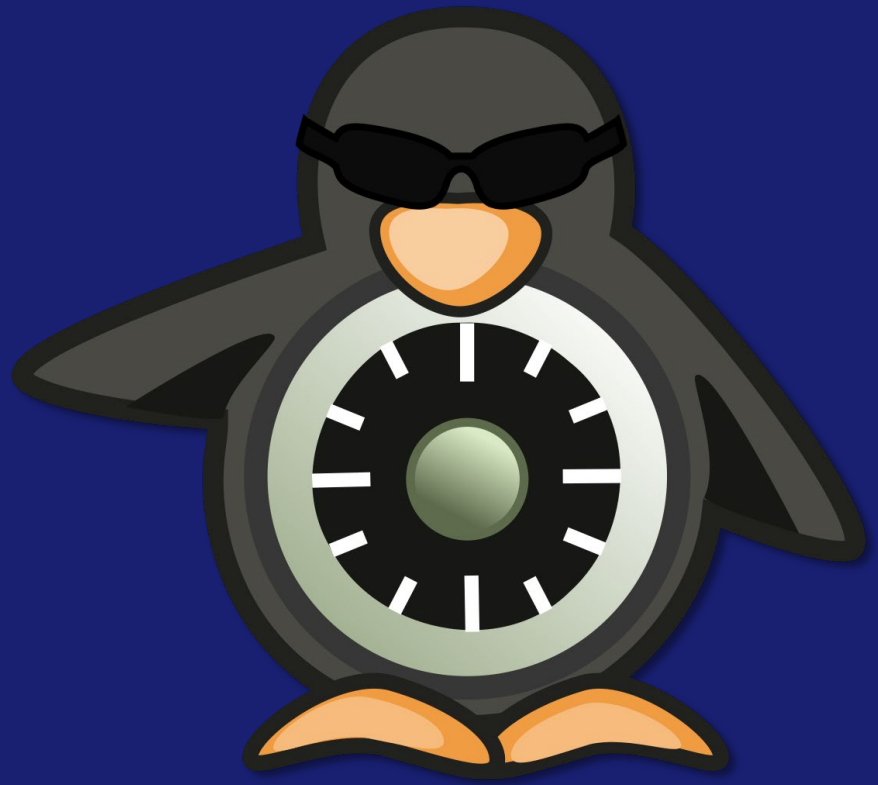
컨테이너를 구성 및 구현하기 위해서 다음과 같은 기술이 커널에서 필요하다.

1. Linux Capabilities
2. Namespace(USER)

추가적으로 다음과 같은 기술이 제공이 되면 컨테이너 운영 시 많은 도움이 된다.

1. SELinux
2. AppArmor
3. SECComp

SELINUX



SELINUX

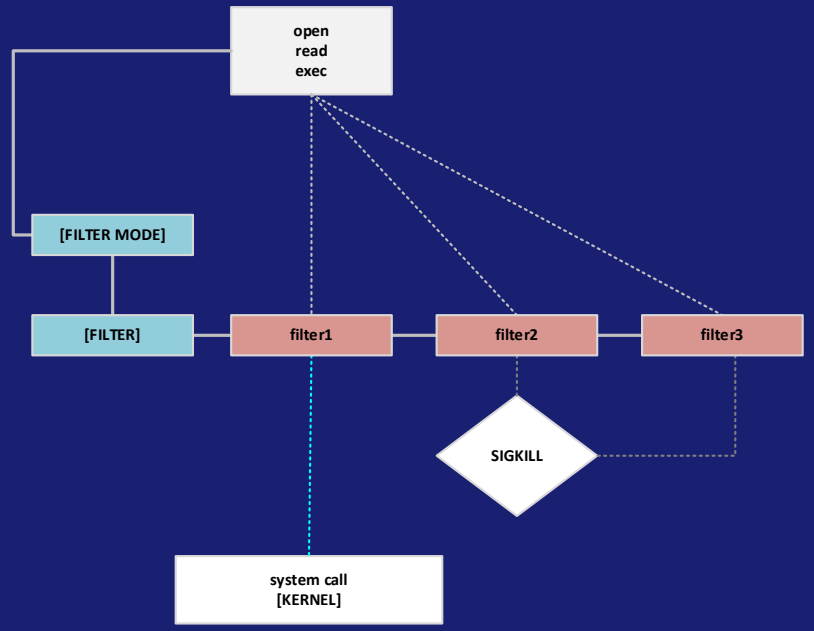
SELinux는 필수적인 컨테이너 구성원은 아니다. 하지만, SELinux를 사용하는 경우 좀 더 안정적으로 운영이 가능하다. SELinux는 다음과 같은 대표적인 구성 요소로 되어있다.

- fcontext
- port
- boolean

SELinux는 모든 배포판에서 지원하지만, 실질적으로 쿠버네티스에 적용은 매우 어려운 부분이다. 이러한 이유로 많은 사용자가 보통 SELinux, AppArmor를 꺼두고 사용한다.

SECCOMP

SECCOMP는 필수 구성원이다. 특정 시스템 콜을 SECCOMP그룹에 넣어서 콜을 제한한다. 이를 통해서 프로세스를 통해서 위험한 시스템 콜을 차단 할 수 있도록 한다. 아래 그림은 SECCOMP가 커널에서 동작하는 메커니즘 설명이다. 간단하게 필터를 통해서 특정 시스템 콜만 커널에서 수행하도록 필터링한다.



링크:SECCOMP

NAMESPACE

네임스페이스는 가상화 기술 중 하나이다. 이 기술은 본래 리눅스 'vServer Project'의 결과물 중 하나다. 네임스페이스는 기능은 프로세스에서 사용하는 각기 다른 자원들을 NAMESPACE(이하 NS)영역으로 분리 및 격리한다.

이를 통해서 얻는 장점은 **오버헤드를 최소화** 한다.

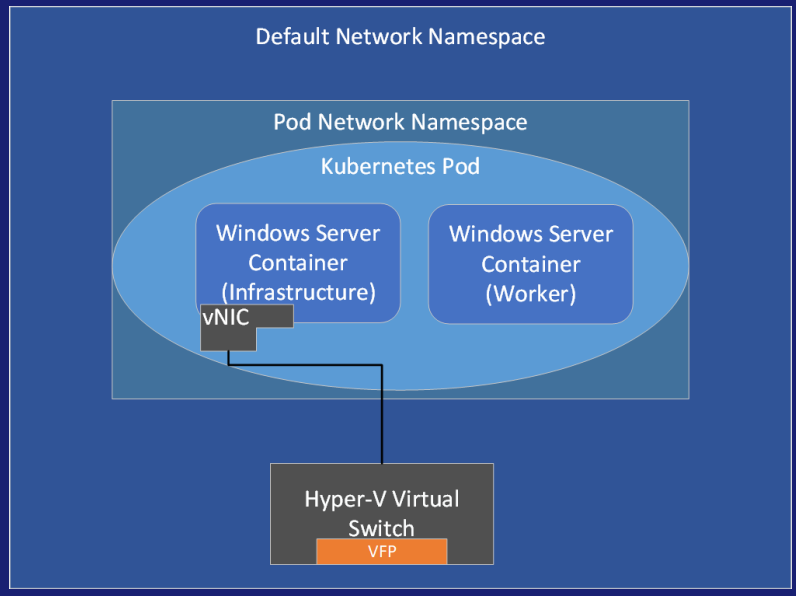
1. 네트워크 장치를 네임스페이스를 통해서 가상화 장치로 제공한다.
2. 파일 시스템에 마운트가 되는 부분을 네임스페이스를 통해서 컨테이너에 제공한다.
3. "pid" 네임스페이스는 컨테이너에서 동작하는 프로세스가 호스트 영역의 프로세스 정보를 확인 할 수 없도록 한다.

컨테이너 및 네임스페이스

이 기술은 유닉스 계열 시스템에서는 하드웨어 혹은 논리적으로 1970년도에 구현이 되었다. 리눅스는 1999년도 후반, 2000년도 초에 컨테이너 구현에 필요한 기능들이 구현이 되었다.

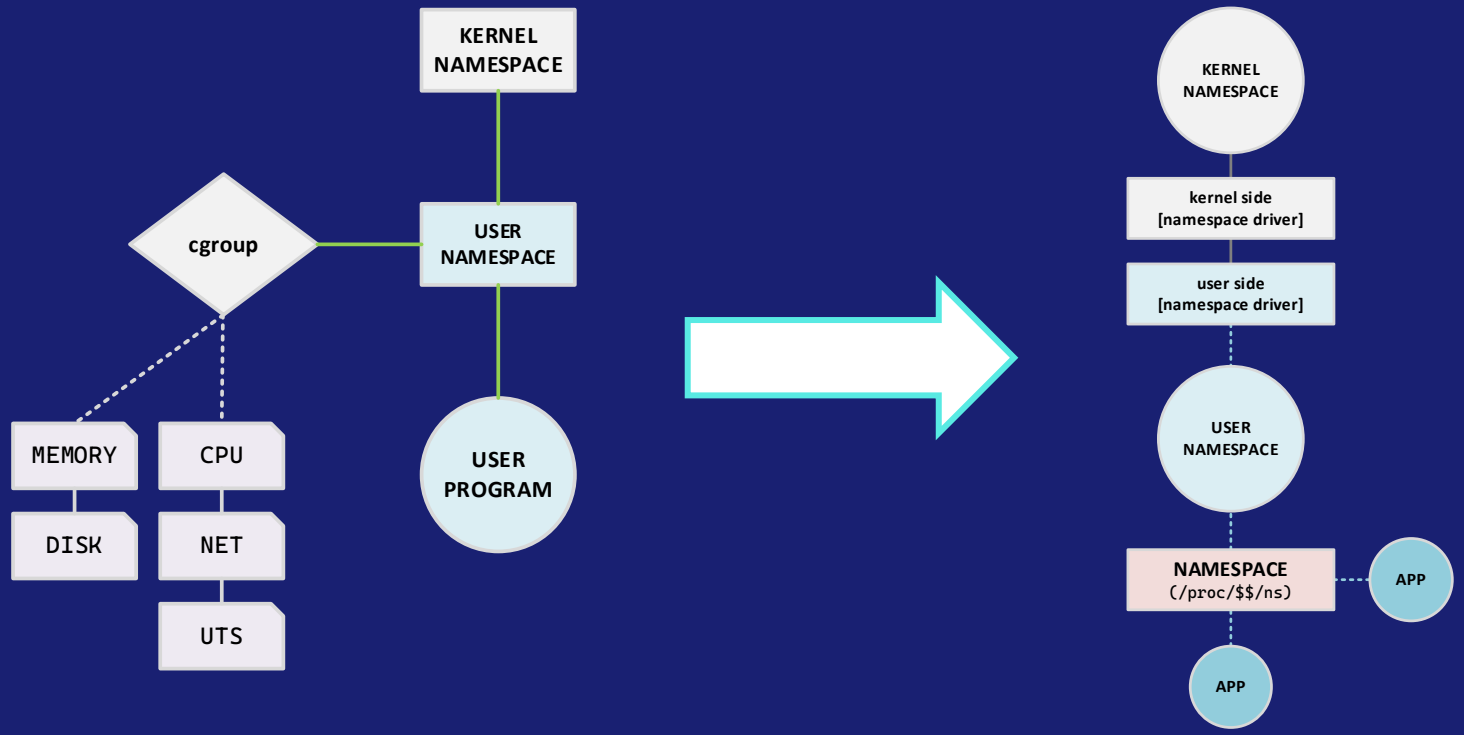
윈도우 서버도 컨테이너 구현을 위한 리눅스와 호환되는 네임스페이스 레이어를 가지고 있다. 이러한 이유로, 리눅스 및 윈도우에서 도커 및 포드만 사용이 가능하다.

https://learn.microsoft.com/en-us/virtualization/windowscontainers/container-networking/network-isolation-security

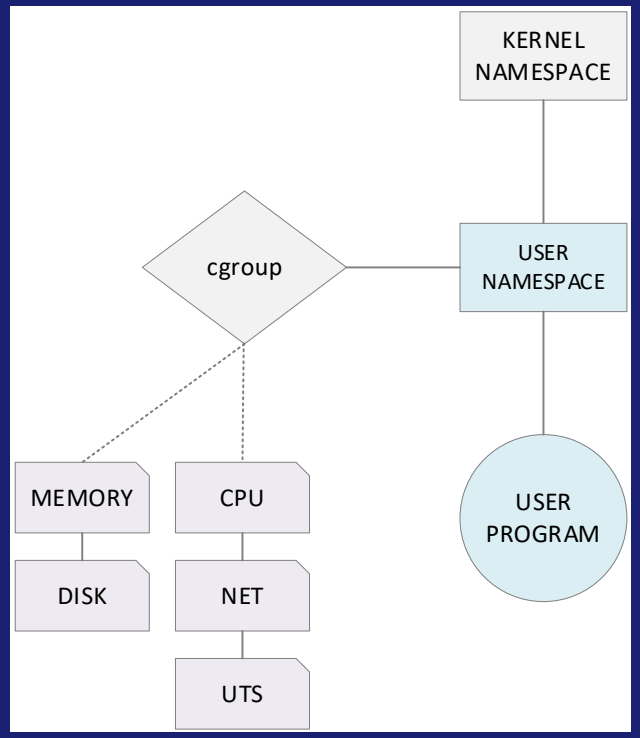


NAMESPACE

컨테이너는 "CGROUP"만 있어도 프로세스 격리가 가능하다. 하지만, CGROUP은 자원 격리보다는 추적/감사 그리고 제한이 주요 목적이기 때문에, 완전한 격리가 어렵다. 레드햇은 프로세스에 완벽한 격리를 제공하기 위해서 커널에 네임스페이스(NAMESPACE)를 추가하였고, 현재 모든 리눅스 컨테이너는 네임스페이스를 통해서 자원 제공 및 공유를 구현하고 있다.

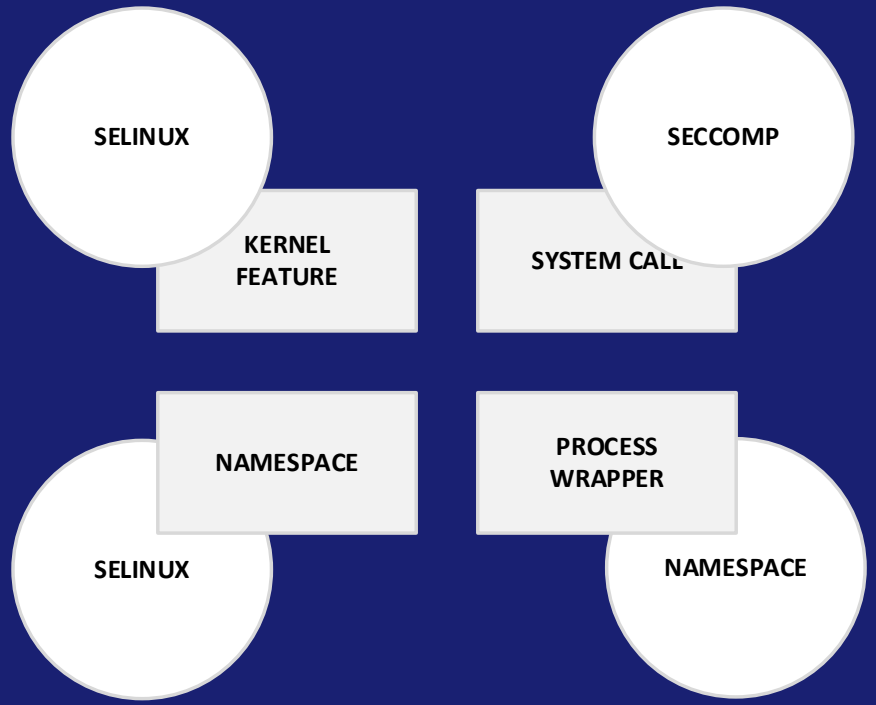


NAMESPACE


KERNEL NAMESPACE
cgroup
USER NAMESPACE
MEMORY
CPU
USER PROGRAM
DISK
NET
UTS

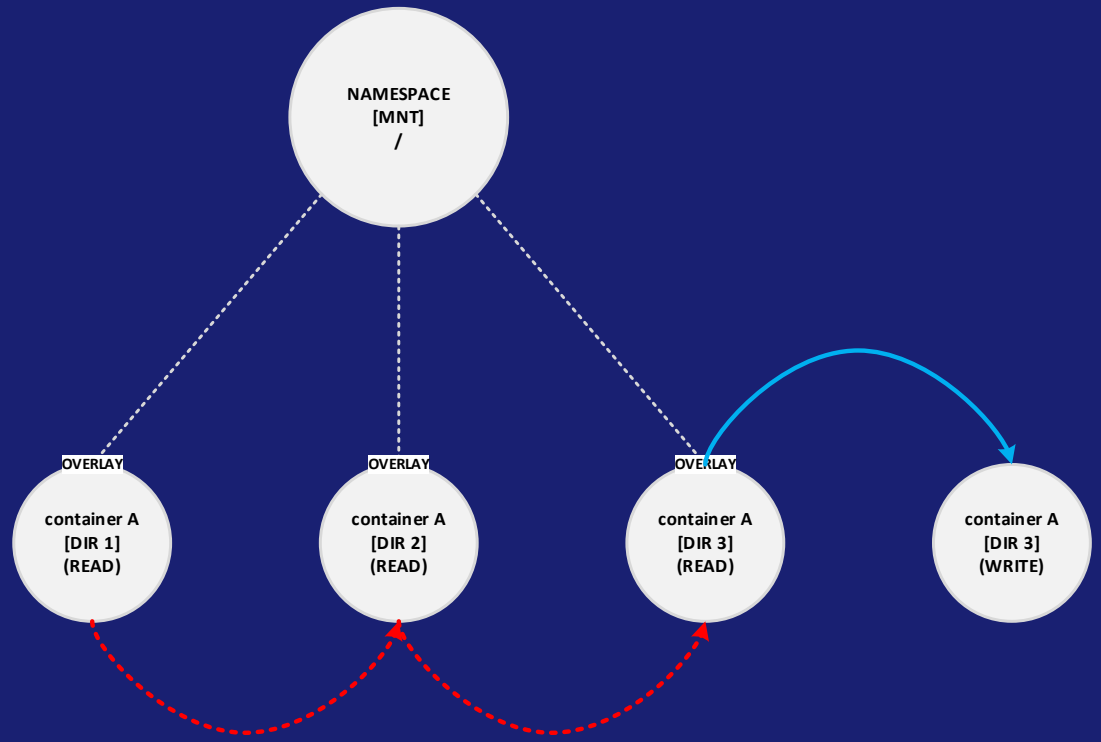
네임스페이스와 구성원 관계

커널 및 운영체제에서 위의 기능들을 구현하여 컨테이너 생성 및 관리 할 수 있는 런타임 환경(RUNTIME ENVIRONMENT)를 제공한다.



NAMESPACE

컨테이너 디스크 보통 이미지라고 부르지만, 시스템에는 디렉터리 생성 및 구성이 되어 있다. 컨테이너 런타임 동작 시 이미지 정보를 확인하여 오버레이 장치로 바인딩 및 마운트 하여 일반 블록장치처럼 보이게 한다.



컨테이너 기술

루트리스/루트풀

루트리스 설명

루트리스 컨테이너는 무엇인가? 도커 컨테이너 엔진 및 런타임이 나오면서 루트리스 컨테이너 개념이 나오기 시작했다. 이전 컨테이너는 루트리스가 아닌, 루트 시스템이 존재하는 형태로 구성이 되었다. 이러한 이유로 컨테이너와 가상머신의 차이가 크게 없었다.

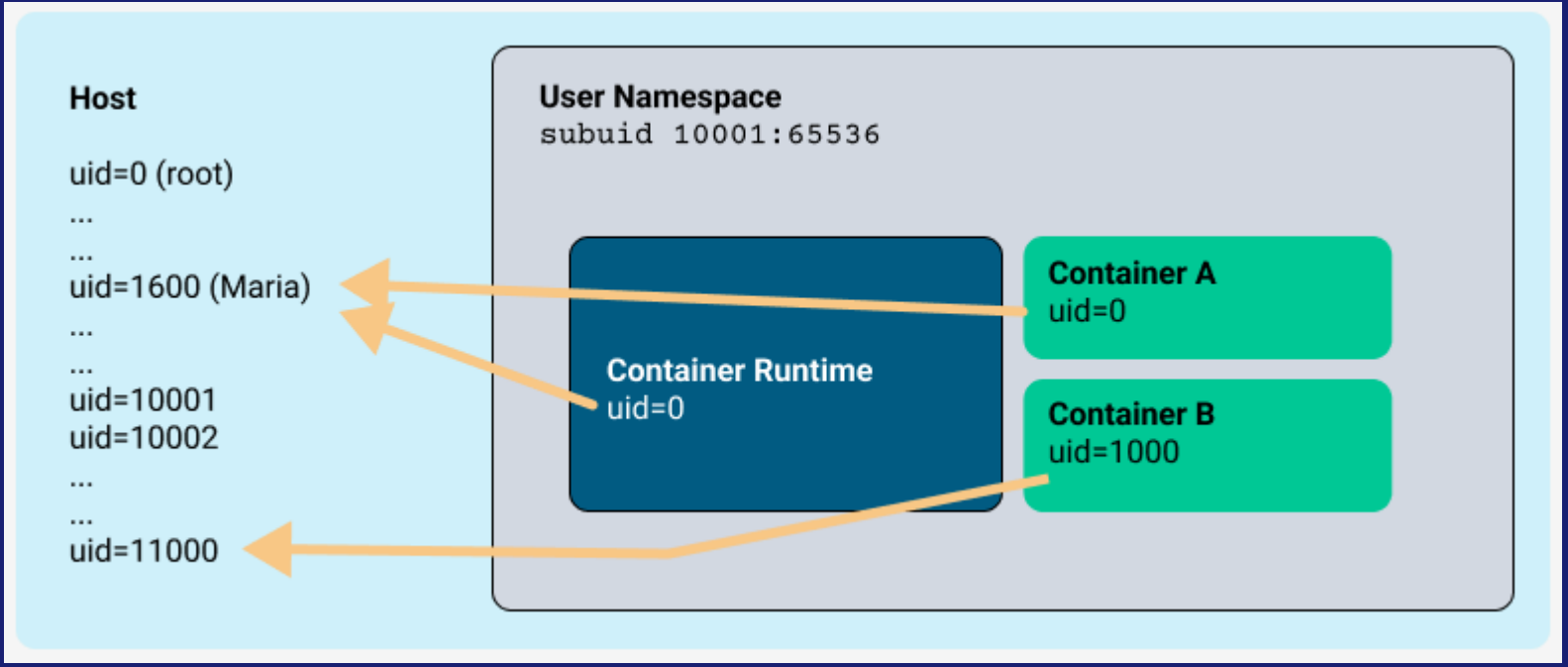
물론, 컨테이너와 가상머신은 링 구조 및 드라이버 구조상 다를 수 밖에 없지만, 루트리스가 아닌 경우 가상머신과 거의 동일한 구조를 가지고 있다. 대표적인 루트 시스템이 있는 컨테이너 시스템은 LXC 컨테이너 런타임이다.

루트리스 비교

도커 및 포드만은 둘 다 루트리스 컨테이너 환경을 제공한다. 하지만, 기본적으로 도커는 루트 영역에서 docker 및 dockerd 대몬이 동작해야 한다.

	도커	포드만
rootless	지원	지원
daemonless	미지원	지원
namespace	자제관리 및 지원	운영체제
cgroup	자체관리 및 지원	운영체제

UID/GID


Host
uid=0 (root)
...
...
uid=1600 (Maria)
...
...
uid=10001
uid=10002
...
...
uid=11000
User Namespace
subuid 10001:65536
Container Runtime
uid=0
Container A
uid=0
Container B
uid=1000

UID/GID

UID/GID

루트에서 작업하는 경우, 네임스페이스 맵핑에 대해서 확인이 어렵다. 루트에서 일반 사용자로, 'su', 'sudo'명령어로 전환하지 말고, 일반 사용자로 ssh나 혹은 콘솔로 직접 로그인 후 사용해야 한다.

- **/proc/self/loginuid**: 사용자 로그인 시 사용하는 UID 확인. 네임스페이스 장치 사용 시 맵핑이 된다.
- **/proc/self/gid_map**: 사용자 로그인 시 사용하는 GID확인. 네임스페이스 장치 사용 시 맵핑이 된다.

포드만에서 생성하는 컨테이너는 기본적으로 "ROOTLESS"형태로 생성 및 구성이 된다.

ROOTLESS

- **/etc/subgid**

OCI사양의 런타임 구성파일. 사용자 gid를 subgid에 명시된 값으로 다시 매핑한다. 이전 도커에서 사용하는 /etc/group 맵핑과 비슷하지만 별도 파일로 선언한다.

- **/etc/subuid**

OCI사양의 런타임 구성파일. 사용자 uid를 subuid에 명시된 값으로 다시 매핑한다. 이전 도커에서 사용하는 /etc/group 맵핑과 비슷하지만 별도 파일로 선언한다. 모든 프로세서는 시작 시, kABI자원 /proc/self/loginuid 를 생성한다. 컨테이너는 이 영역 정보를 아래처럼 변경해서 컨테이너 구성 및 생성한다.

ROOTLESS CONTAINER

"/proc/self/loginuid" 위의 정보 확인하기.

```
$ podman run --rm quay.io/centos/centos:stream9-minimal cat
/proc/self/loginuid
```

ROOTFUL CONTAINER

"rootful"컨테이너를 구성 및 생성하기 위해서 간단하게 다음과 같이 Podman에서 실행 및 구성하면 된다.
rootful에서 사용할 가상머신을 생성 후, 아래와 같은 명령어로 확인이 가능하다.

```
$ sudo dnf install qemu-kvm qemu-img -y
$ podman machine init
$ podman machine list
$ podman machine info
$ podman machine start
$ podman machine os
$ podman machine set --rootful
$ podman machine set --rootful rootful-vm
```

이 과정에서는 rootful기반으로 하지 않기 때문에 위의 옵션을 사용하지 않아도 된다.

ROOTFUL

위의 명령어로 간단하 컨테이너를 실험적으로 동작한다. 위의 컨테이너는 **권한**(privileged)를 주었기 때문에, 호스트의 자원에 직접적으로 접근이 가능하다.

```
$ podman run -it --name hacker --privileged -v /:/host ubi8 chroot /host
```

기반기술

네임스페이스

설명

네임스페이스에 대한 이해를 위해서 명령어로 직접 수동 컨테이너를 생성한다. 다시 말하지만, 이 명령어는 컨테이너 생성 명령어가 아니라 어떠한 방식으로 동작하는지 테스트 하는 확인 명령어이다.

UNSHARE(ipc)

런타임을 통하지 않고, IPC를 네임스페이스로 구성하는 경우, 아래와 같이 생성이 가능하다.

```
# unshare -i --mount-proc /bin/bash
```

UNSHARE(uts)

readlink, 링크가 걸려있는 본래 파일 확인. "uts"는 프로세서가 사용하는 시간을 제공하는 자원.

```
# unshare --fork --pid --mount-proc readlink /proc/self
1
# unshare --map-root-user --user sh -c whoami
root
# touch /root/uts-ns
# nsenter --uts=/root/uts-ns hostname
FOO
# umount /root/uts-ns/
```

UNSHARE(mnt)

컨테이너 생성시, OverlayFS를 다음과 같은 형태로 만든다. "mnt"자원은 컨테이너에 파일 시스템을 생성 및 제공하며, 이는 바인딩(bind) 및 오버레이(overlay2) 모듈을 통해서 제공한다.

```
# mkdir /root/mnt-ns/
# mount --bind /root/mnt-ns/ /root/mnt-ns/
# mount --make-private /root/mnt-ns
# touch /root/mnt-ns/mnt
# unshare --mount=/root/mnt-ns/mnt
```

UNSHARE(net)

컨테이너에서 사용하는 네트워크는 아래와 같이 생성한다. 네임스페이스에서 네트워크 임시 장치를 생성 후 가상 장치를 연결 및 구성한다.

```
# touch /root/net-ns
# unshare --uts=/root/uts-ns hostname opensource
# unshare --net=/root/net-ns true
# ip link add veth0 type veth peer name veth1
# nsenter --net=/root/net-ns sleep 30 & pid=$!
# ip link set veth1 netns $pid
# ip -br link
```

UNSHARE(net)

구성된 네트워크 장치에 아래와 같이 장치 설정 및 인터페이스 연결을 합니다.

```
# mount -t tmpfs --make-rshared tmpfs /run/netns
# mount -t tmpfs tmpfs /run
# ip netns add net-ns
# ip link add name veth0 type veth peer netns net-ns name veth1
# ip -br link
# ip -n net-ns -br link
```

UNSHARE(loopback device)

루프백 장치는 아래와 같이 생성이 가능하다. 컨테이너 프로세서도 가상머신처럼 루프백 장치는 꼭 필요하다.

```
# unshare -n bash
# echo $$
# ip link
# touch /run/netns/new_namespace
# mount -o bind /proc/$$/ns/net /run/netns/new_namespace
# ip netns exec new_namespace bash
# ip link set lo up
# ip a l
```

UNSHARE(namespace isolate)

```
# nsenter -t <PID> -n -S 100024 -G 100024
```

SHPPING(container vs virtual machine)

컨테이너 기반으로 애플리케이션을 동작하기 위해서는 애플리케이션을 어딘가 **'담아야'**한다. 현재는 이걸 두 가지 형태로 나누어서 구현한다.

1. 가상머신
2. 컨테이너

가상머신은 기본적으로 하드웨어를 **100% 에뮬레이터 구성 및 생성**하기 때문에, 일반 컴퓨터와 동일하게 구성 및 구현하는 효과가 있지만, 무겁다는 단점이 있다.

또한, 가상머신으로 구성되는 경우, 모든 가상머신들은 "rootful", "daemonful"한 상태를 구성하게 된다. 이러한 이유로 서비스 시작이 빠르지 않다.

컨테이너는 이러한 단점을 해결하기 위해서 **완전한 격리**가 아닌, **논리적 격리 방식**을 선택. 초창기 컨테이너는 "rootful"한 행태로 구성하였다가, 지금은 "rootless" 및 "daemonless"한 형태로 구성한다.

CONTAINER SHPPING

CONTAINER IMAGE

CONTAINER PACKAGING

컨테이너 런타임에서 사용하는 컨테이너는 다음과 같은 조건으로 애플리케이션 및 라이브러리를 보관한다.

1. tar기반의 이미지 파일
2. 디렉터리 기반으로 레이어 분류
3. 애플리케이션 사용시 필요한 바이너리리 및 애플리케이션

컨테이너 기술

런타임 엔진

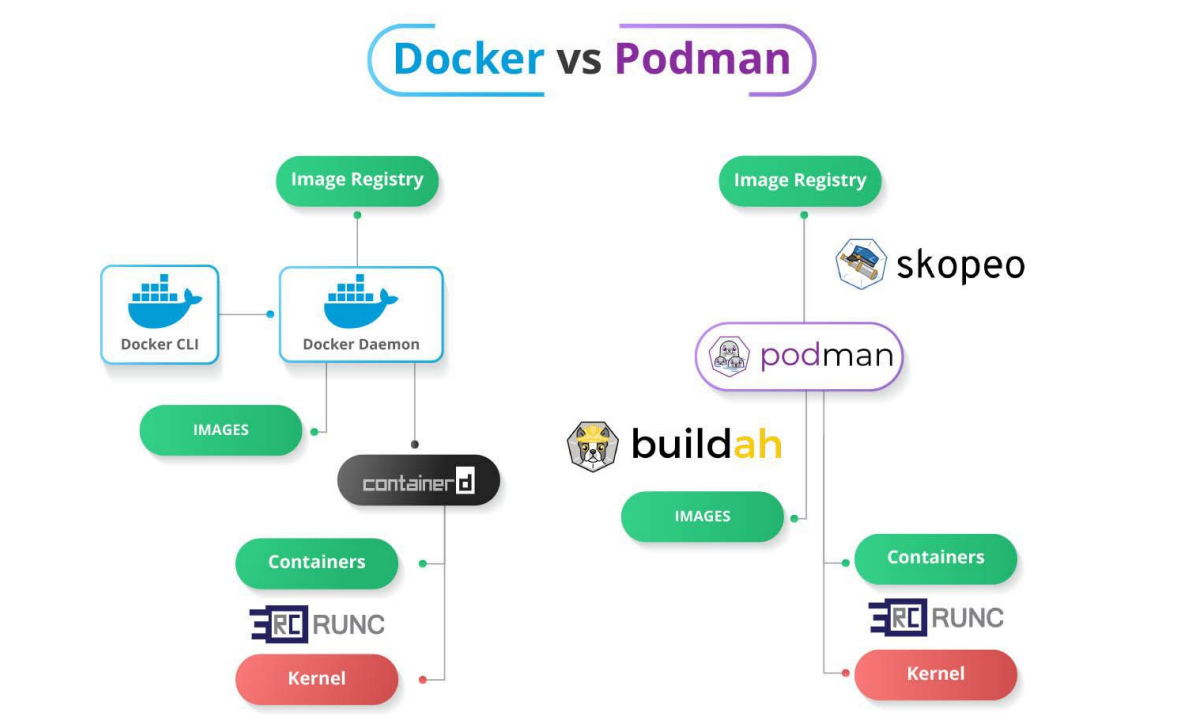
컨테이너 엔진/컨테이너 런타임

컨테이너 엔진과 컨테이너 런타임은 기능적으로 다르다. 컨테이너 엔진은 보통 컨테이너 데몬 및 API를 다루는 역할을 하며, 대표적인 예제는 **docker, dockerd**이다. docker명령어를 통해서 조회를 하면, dockerd는 해당 API를 받아서 **저-수준 컨테이너 런타임**에 동작중인 런타임 목록을 화면에 출력한다.

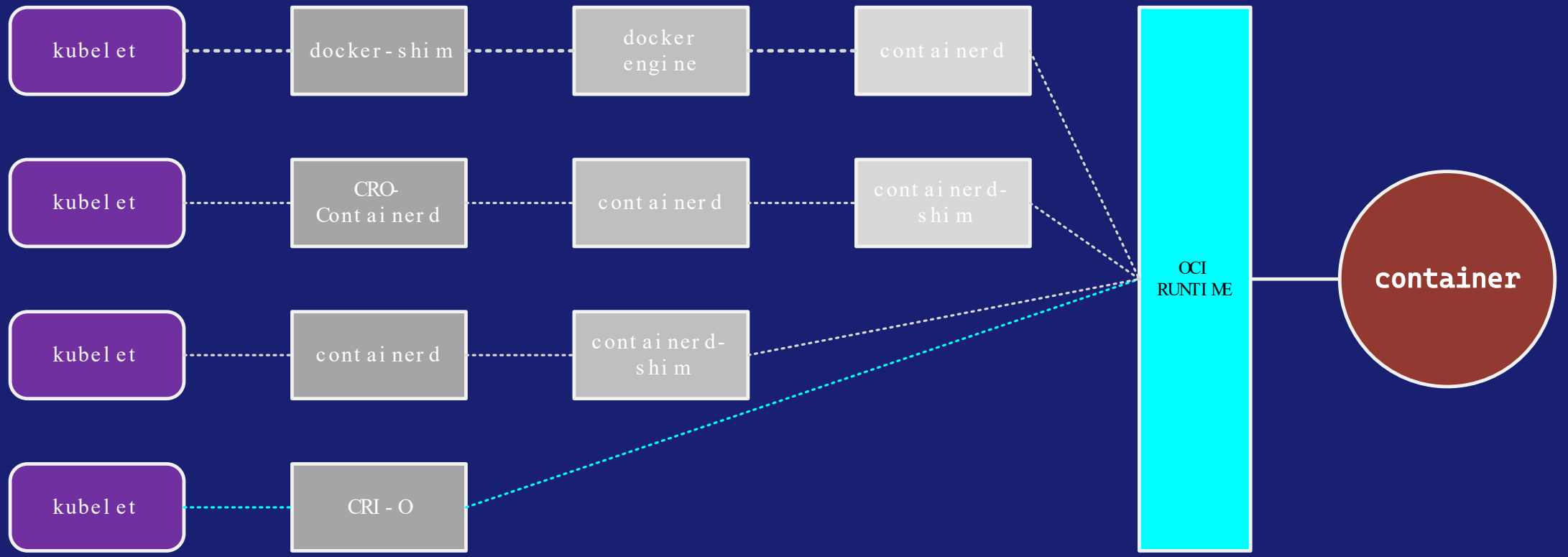
이러한 이유로, 쿠버네티스는 저-수준의 컨테이너 런타임만 사용하며, 도커 및 포드만은 보통 상위 컨테이너 엔진 기반으로 컨테이너 관리 및 생성을 한다.

독립적인 컨테이너 프로그램들은 일반적으로 엔진 기반으로 구성하며, 여러 게 컨테이너를 클러스터 단위로 구성하는 경우, 런타임 기반으로 구성한다.

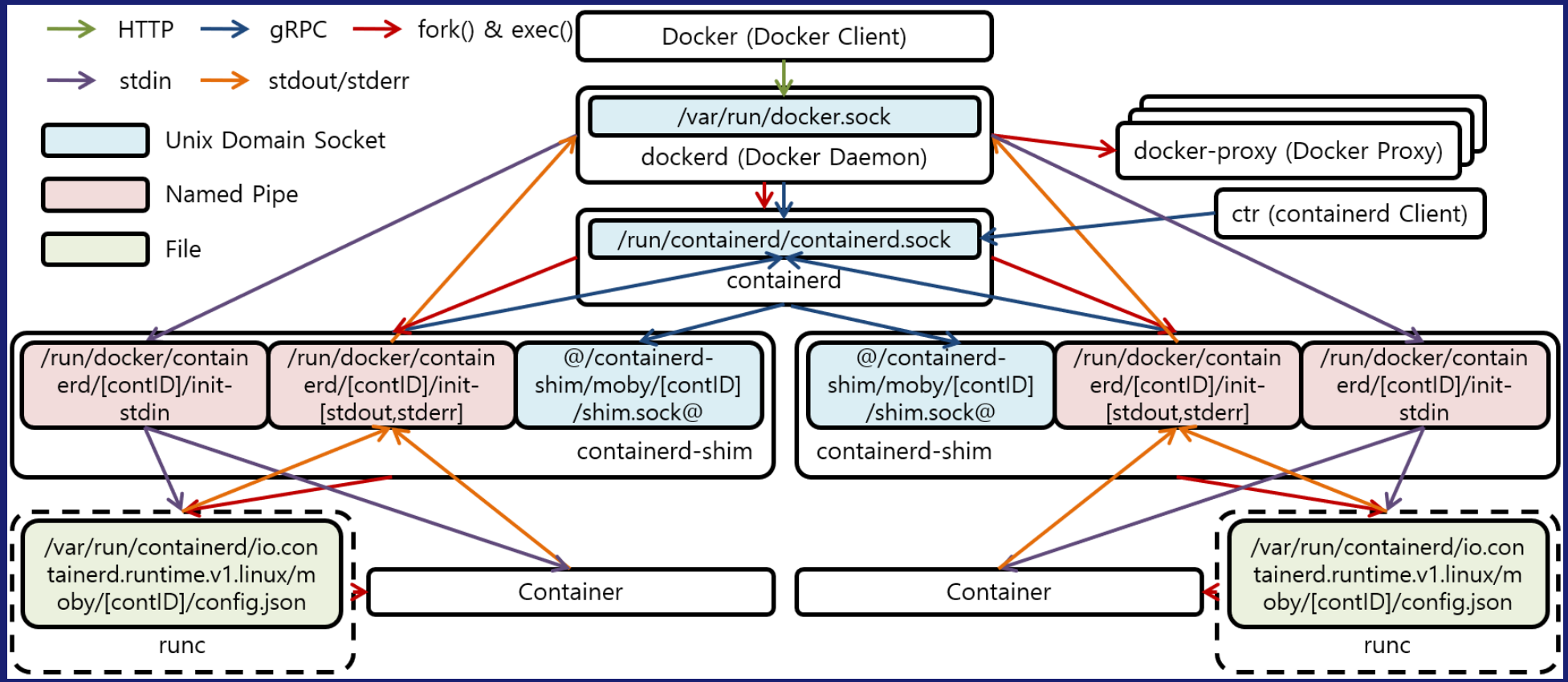
DOCKER VS PODMAN



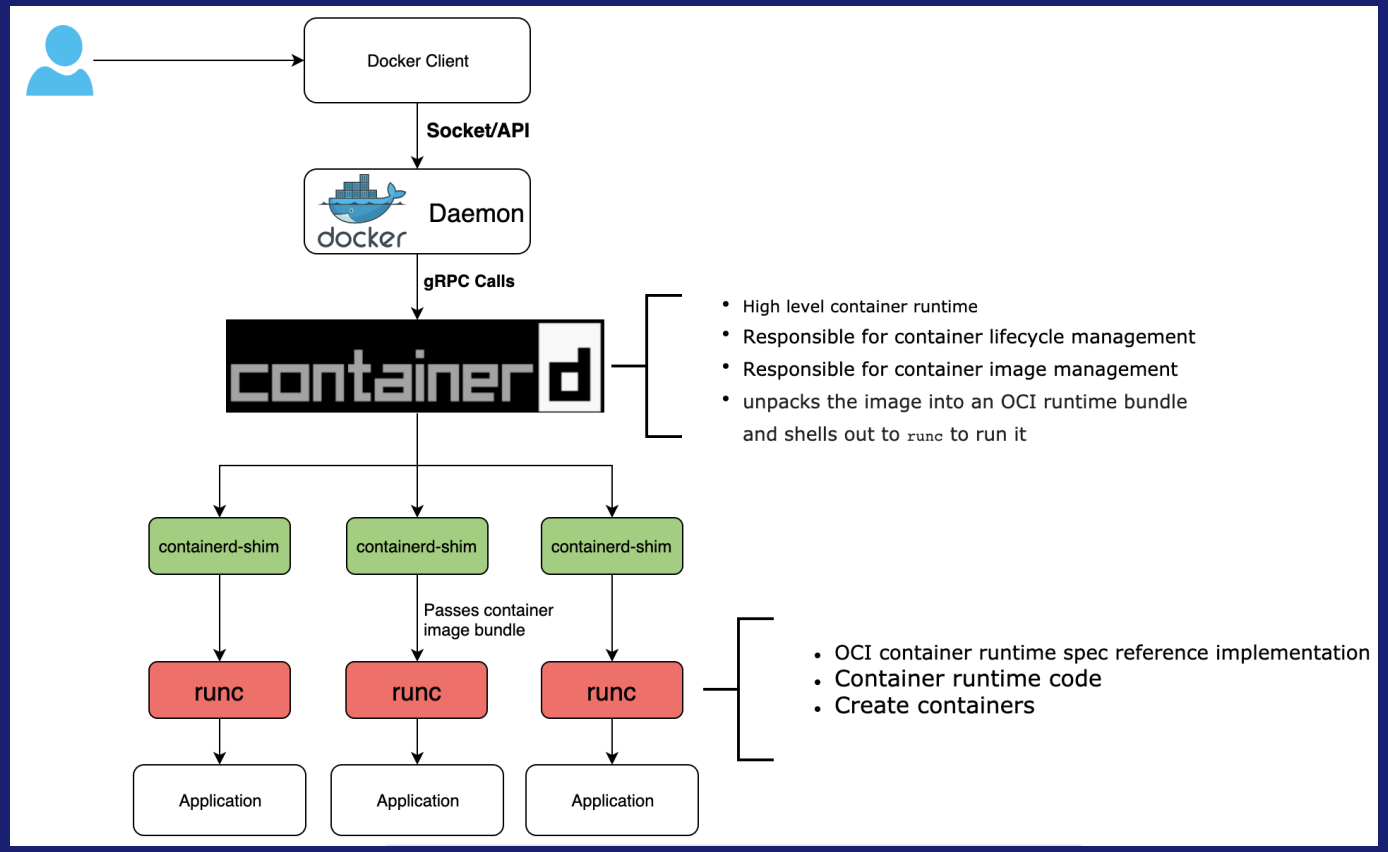
CONTAINER RUNTIMES(OCI/CRI)



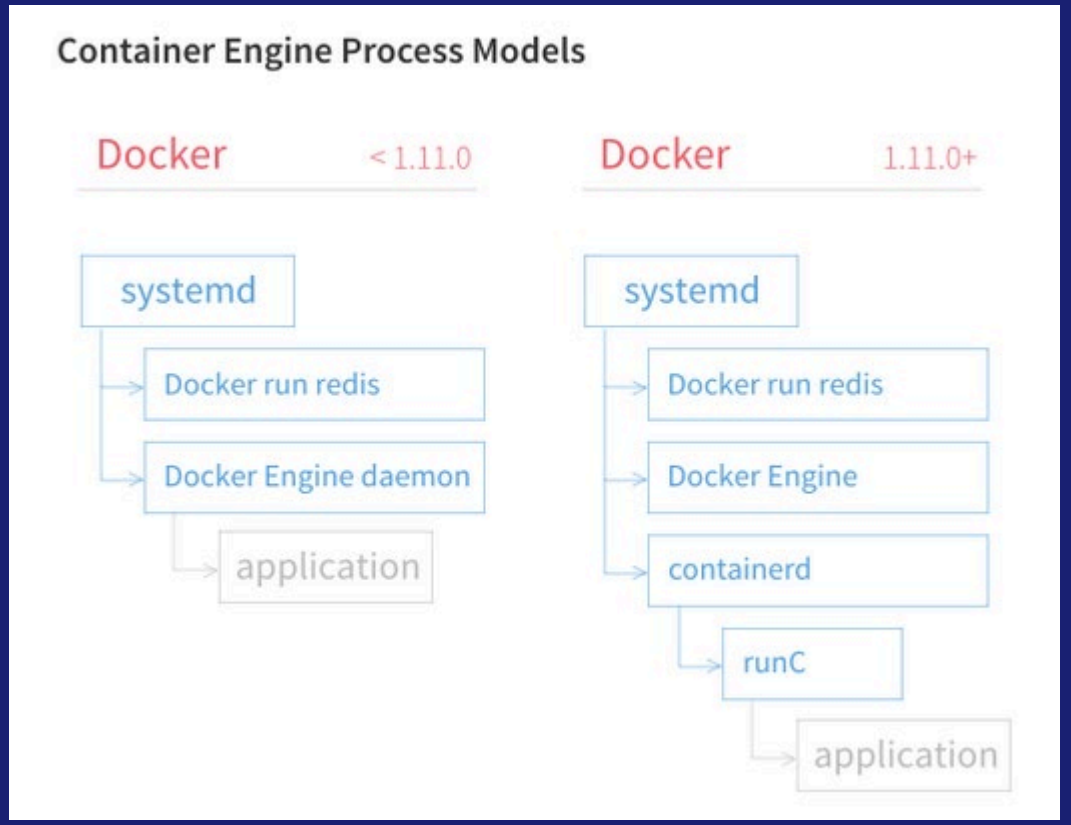
CONTAINER INTERACTIVE(docker/containerd)



DOCKER OCI

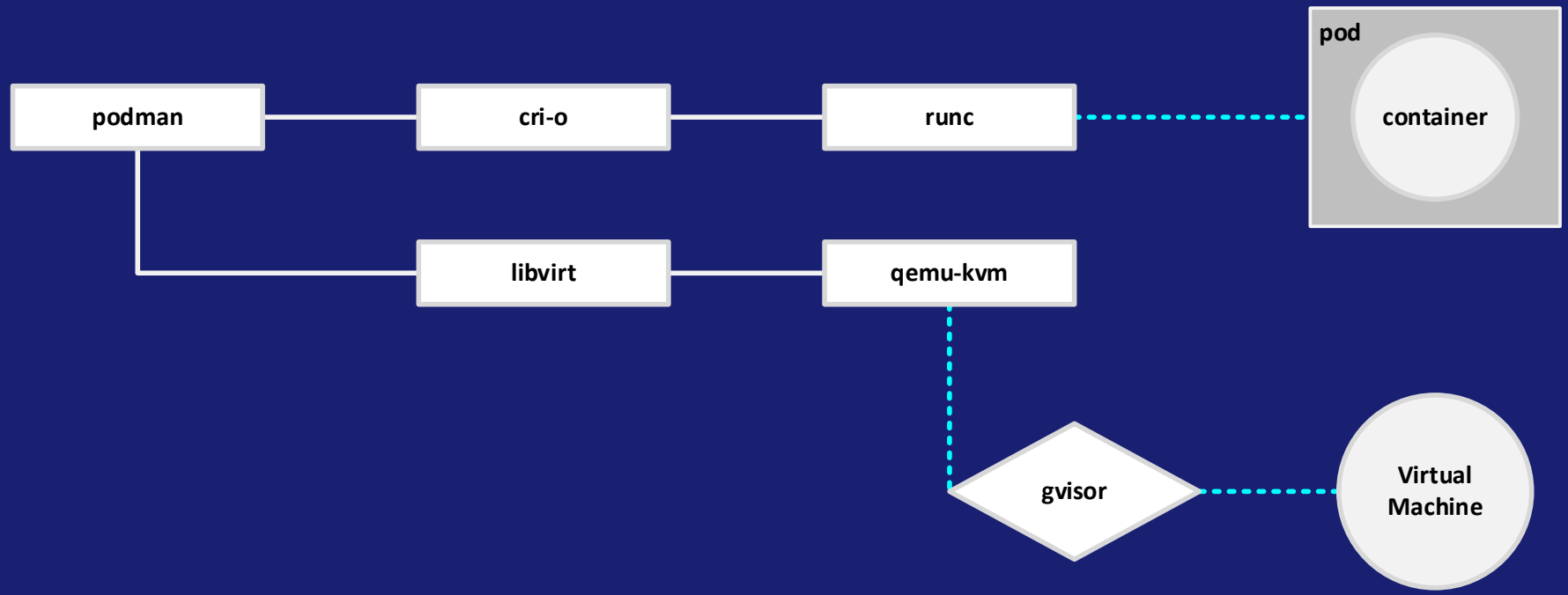

Docker Client
Socket/API
Daemon
docker
gRPC Calls
containerd
High level container runtime
Responsible for container lifecycle management
Responsible for container image management
unpacks the image into an OCI runtime bundle
and shells out to runc to run it
containerd-shim
containerd-shim
containerd-shim
Passes container image bundle
runc
runc
runc
OCI container runtime spec reference implementation
Container runtime code
Create containers
Application
Application
Application

도커 엔진 모델(OCI 비호환/호환)



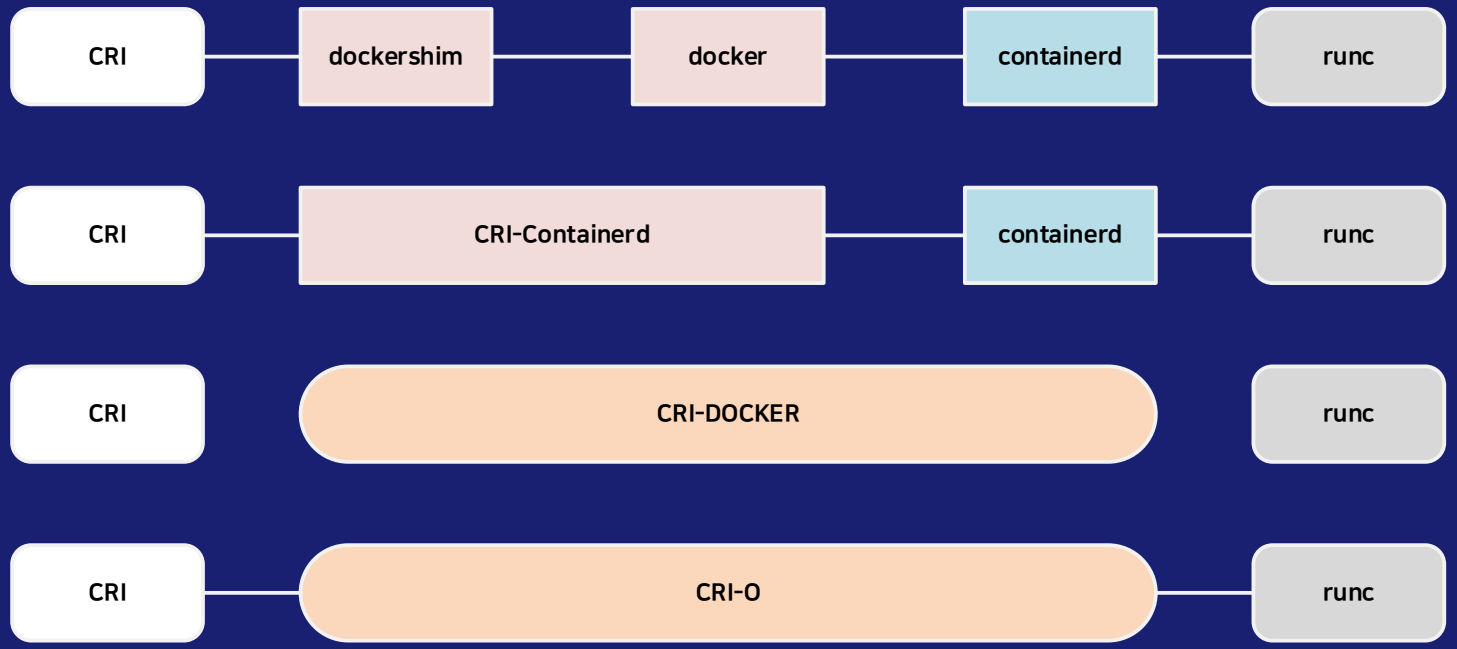
포드만 컨테이너 및 가상화

포드만은 컨테이너 및 가상머신을 둘 다 지원한다. 가상머신은 기존과 동일하게 libvirt를 사용하지만, "gvisor"를 통해서 가상머신 생성이 된다. Pod 생성 시, 사용하는 'pause'와 동일한 역할을 한다.



CRI 컨테이너 인터페이스 레이어(혹은 어뎁터)

현재 컨테이너 런타임은 CRI인터페이스 호환이 되도록 구성이 되어있다. 다만, 도커는 사용환경 및 설정에 따라서 다를 수 있다.



DOCKER VS PODMAN

기능	PODMAN	DOCKER	설명
OCI 및 DOCKER 이미지 지원	지원	지원	
OCI 컨테이너 엔진 지원	지원	지원	
간단한 명령어 지원	지원	지원	
시스템 블록 통합(systemd)	지원	미지원	docker는 자체적으로 namespace, cgroup관리.
Fork/Exec 지원	지원	지원	
클라이언트 서버(API)	지원	지원	도커는 EE버전으로 지원하며 CE에서는 지원하지 않음.
docker-compose	지원	지원	

DOCKER VS PODMAN

기능	PODMAN	DOCKER	
레지스트리 수정	지원	미지원	podman은 보안, 저장소 및 네트워크 수정 가능
기본구성 설정 변경	지원	지원	
맥 지원	지원	지원	
윈도우 지원	지원	지원	WSL2 기반으로 지원
리눅스 지원	지원	지원	
업데이트 시 중지 필요	해당사항 없음	중지 필요	

정리

PODMAN

명령어

PODMAN

명령어

명령어

이미 도커를 사용한 경험이 있는 사용자는, 기본 명령어 부분은 크게 차이가 없기 때문에 아래 내용에 대해서 학습은 따로 필요 없다. 하지만, 몇몇 부분에서 다르게 동작하기 때문에 해당 부분에 대해서 한번 더 확인한다.

현재 표준 컨테이너는 다음과 같은 패키지로 나누어 진다.

1. 표준 컨테이너 런타임 엔진 포드만
2. 표준 컨테이너 도구(skopeo, buildah…)

INSTALLATION PODMAN

포드만 설치는 RHEL기반의 클론 버전인 ROCKY LINUX에서 설치한다. 가상머신은 이미 oVirt시스템에 리눅스 설치 및 구성이 되어 있기 때문에 바로 설치만 진행하면 된다.

```
$ dnf search container-tools podman
$ sudo dnf install podman epel-release -y
$ podman images
$ podman pods ls
$ podman containers ls
```

command:ps

둘 다 동일한 컨테이너 목록을 출력. 권장은 'podman container ls'명령어 사용을 권장.

```
$ podman container ls
$ podman ps
```

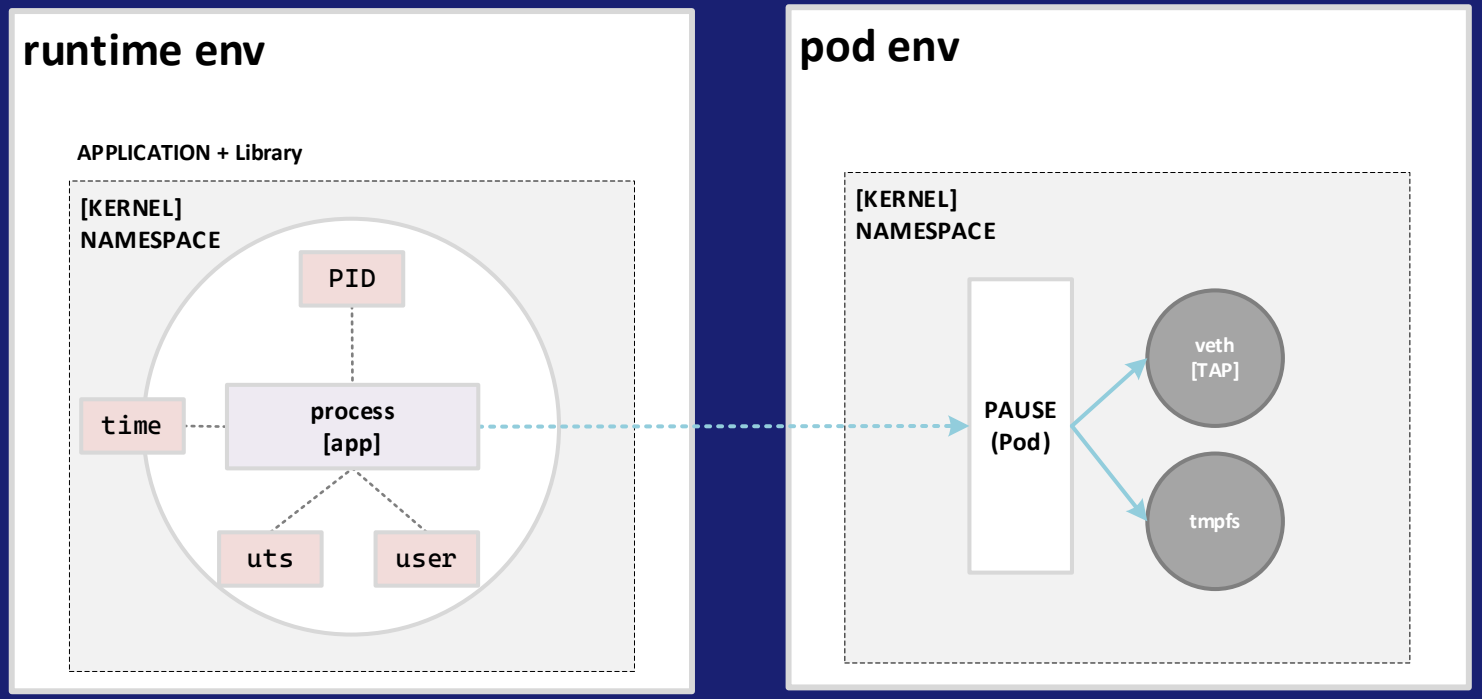
Podman POD

Pod는 배포판 조금씩 다를 수 있다. 여기서 말하는 POD 혹은 Pod는 쿠버네티스에서 말하는 개념이다. 레드햇 계열은 RHLE 7이후부터는 포드만에서 **"kata"라는 Pod 애플리케이션을** 사용하고 있으며, 이전 버전들은 쿠버네티스에서 사용하는 "pause"기반으로 사용하고 있다.

"Pod"는 쿠버네티스에서 제시한 개념이며, "pause"는 "Pod"를 구현하기 위해서 사용하는 **컨테이너 애플리케이션**이다.

Pause프로그램이 동작하는 컨테이너 자원은 "인프라 컨테이너(infra-container)"라고 불리기도 한다. 이 목차에서는 간단하게 생성 및 실행만 하고, 컨테이너와 연동해서 활용하는 부분은 후에 진행하도록 한다.

POD 개념


runtime env
APPLICATION + Library
[KERNEL]
NAMESPACE
PID
time
process
[app]
uts
user
pod env
[KERNEL]
NAMESPACE
PAUSE
(Pod)
veth
[TAP]
tmpfs

command:pod

생성 및 동작중인 Pod를 확인한다. 일반적으로 사용하는 Pod는 쿠버네티스에서 사용하는 'pause'기반으로 POD를 구성한다. 아래 명령어로 간단하게 Pod생성 및 구성이 가능하다.

```
$ podman pod create
$ podman pod ls
$ podman pod create test
```

command:pod

생성한 Pod는 아래와 같이 실행한다. Pod는 애플리케이션 컨테이너가 연결이 되지 않으면, 시작하지 않는다.

```
$ podman pod start <ID> or <NAME>
POD ID        NAME                STATUS      CREATED         INFRA ID      #
OF CONTAINERS
5de2fc474c1c  gallant_mahavira  Running     2 minutes ago   e223bc8ef0b7  1
```

command:pod

실행된 Pod는 아래와 프로세스 확인이 가능하다. 또한, 이미지를 파일로 저장해서 어떻게 구성이 되어 있는지 내부 구성원을 확인한다.

```
$ ps -ef | grep -i pause
> pause
$ podman save <POD_CONTAINER_IMAGE> -o pod.tar
$ mkdir kata
$ tar xf pod.tar -C kata/
$ tar xf <FILENAME>.tar
```

command:run

컨테이너 런타임에서 컨테이너를 생성한다.

`--rm`	컨테이너가 중지가 되면, 컨테이너를 제거한다.
-v	디렉터리를 바인딩 하는 옵션. 일반적으로 컨테이너 내부의 프로그램이 저장하는 외부 장치에 이 옵션을 같이 사용한다. 혹은 런타임에서 사용하는 볼륨을 사용하기도 한다.
`-d`	detach 옵션은 사용자 영역에서 실행하는 프로그램을 사용자 영역(userspace)에서 분리한다.
`-i`	interactive 옵션은 사용자가 대화형으로 stdout/stderr를 podman 명령어를 통해서 주고 받는다.
`-t`	컨테이너에 tty장치를 생성해서 터미널을 구성한다. 여기에서 생성하는 tty는 pseudo-tty장치이며, 실제로 존재하지 않는 가상의 터미널이다.
`-p`	컨테이너 애플리케이션이 사용하는 포트를 내부에서 외부로 바인딩하는 옵션.

명령어 및 옵션은 아래처럼 사용한다.

```
$ podman run -ti --rm centos:stream9-minimal bash
```

command:run

컨테이너를 실행하는 명령어. 컨테이너 런타임을 통해서 컨테이너 생성 및 실행한다. 컨테이너 실행 시, 백 그라운드로 실행 혹은 포그라운드로 실행이 가능하다. 아래 명령어는 컨테이너 실행 후, TTY를 통해서 연결한다.

```
$ podman run -d centos:stream9-minimal
$ podman run -it --rm centos:stream9-minimal bash
[root@9bab61b80300 /]# ip link
bash: ip: command not found
[root@9bab61b80300 /]# ls /bin | wc -l
185
[root@9bab61b80300 /]# cd /dev/
[root@9bab61b80300 dev]# ls
console  core  fd  full  mqueue  null  ptmx  pts  random  shm  stderr  stdin  stdout
tty  urandom  zero
[root@9bab61b80300 dev]#
```

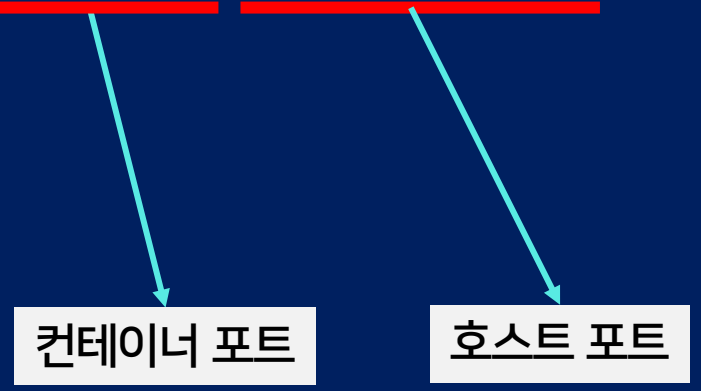
command:port

컨테이너에서 사용하는 포트를 호스트로 바인딩한다. 왼쪽은 '**컨테이너 포트**', 오른쪽은 '**호스트 포트**'를 명시한다.
도커는 docker-proxy를 통해서 컨테이너 포트를 구성한다.

```
$ podman run -d -p 80:8080 --rm --name centos-httpd centos7/httpd-24-centos7
$ podman run -d -p 8080:80 --rm --name centos-httpd centos7/httpd-24-centos7
```

RUN(PORT)

```
# podman port -l or port <CONTAINER ID/NAME>
80/tcp -> 0.0.0.0:8080
```



STOP

컨테이너를 중지하기 위해서 'stop'명령어를 실행한다.

```
$ podman stop <CONTAINER ID> <CONTAINER NAME>
$ podman container stop
```

중지가 잘 되었는지 'ps'명령어를 통해서 확인한다.

```
$ podman ps
$ podman container ps
```

STOP

실행중인 모든 컨테이너를 중지하기 위해서는 '--all'명령어를 통해서 중지가 가능하다.

```
$ podman container stop --all
```

올바르게 컨테이너가 중지되지 않으면 직접적으로 중지 신호를 명시 할 수 있다.

```
$ podman container stop -t <SIGKILL NUMBER> <CONTAINER ID OR NAME>
```

START

중지된 컨테이너를 다시 실행하기 위해서 위의 명령어로 실행한다.

```
$ podman container start <ID> <CONTAINER NAME>
```

INSPECT

- 동작중인 pod/container의 정보를 확인한다.
- 정보는 JSON형태로 제공한다.
- 이미지도 역시 inspect명령어로 확인이 가능하다.

컨테이너 런타임별로 동작 방법이 조금 다를 수 있으니 주의가 필요하다.

```
$ podman image inspect <ID> <NAME>
          container == podman ps
          pod       == podman ps -a
```

도커 경우에는 사용하는 버전 및 런타임 수준에 따라서 명령어가 다를 수 있다.

INSPECT

자주 사용하는 옵션은 다음과 같다.

옵션	설명
--latest(-l)	마지막에 실행된 컨테이너의 정보를 확인한다. 이름 및 컨테이너 아이디 명시가 필요 없다.
--format	JSON에서 특정 데이터 딕셔너리를 선택하여 화면에 출력한다.
--size	컨테이너가 사용하는 공간 크기를 같이 검사한다. image, pod에서는 이 옵션은 동작하지 않음.

INSPECT

JSON에서 특정 딕 및 리스트에서 값을 찾아서 화면에 출력한다.

```
$ podman container inspect centos-httpd --format '{{ .Config.Cmd }}'
[/usr/bin/run-httpd]
$ podman container inspect centos-httpd --format '{{ .Config.StopSignal }}'
15
$ podman container inspect centos-httpd --format '{{ .NetworkSettings.Ports }}'
map[8080/tcp:[{ 80}] 8443/tcp:[]]
```

PS

컨테이너화 되어서 동작하는 프로세스 목록을 확인한다.

```
$ podman container ps
a98f2a616cfa  quay.io/xinick/containerlab/httpd:latest  /bin/sh -c httpd ...  2 hours ago     Up
2 hours ago     0.0.0.0:8080->80/tcp  gallant_mahavira
```

PS

'ps' 명령어는 'process'의 약자이며, 컨테이너에서 동작중인 프로세스 목록을 확인한다.

옵션	설명
--size	현재 사용중인 컨테이너의 크기를 확인한다. 다만, 시간이 좀 걸린다.
--sort	정렬 기준의 필드를 선택합니다.
--noheading	맨 위의 제목 필드를 출력하지 않는다.
--latest(-l)	마지막에 생성된 컨테이너를 출력한다.
--watch(-w)	실시간으로 프로세스 목록을 명시된 초만큼 갱신한다.

RM

컨테이너를 제거한다. 다만, 컨테이너 제거 시, 컨테이너에서 사용하는 프로세스는 반드시 중지가 되어 있어야 한다.

```
$ podman container rm centos-httpd
          pod
```

옵션	설명
- - a l l	모든 컨테이너를 제거한다.
- - f or c e	컨테이너를 강제로 제거한다. 다만, 사용 중이면, 컨테이너가 중지가 될 때 제거가 된다.

RM

런타임 엔진에서 관리하는 이미지를 제거한다.

```
$ podman rmi <IMAGE NAME OR ID>
```

옵션	설명
- - a l l	모든 이미지를 한번에 제거한다. 다만, 사용중인 경우 제거가 되지 않을 수 있으며, 사용 중에 강제로 제거하면 컨테이너가 올바르지 않는 동작을 한다.

EXEC

컨테이너 내부에 명령어 실행. 컨테이너에서 생성된 파일이나 혹은 바인딩된 디렉터리를 확인 시 사용한다.

```
$ podman exec -i centos-httpd bash -c 'cat > /var/www/html/index.html' << _EOF
Hello World
_EOF
```

EXEC

대화형 모드는 반드시 '-i', '-t'옵션을 통해서 접근이 가능하다.

```
$ podman exec centos-httpd cat /var/www/html/index.html
$ curl localhost:8080
```

RUN(EXEC)

컨테이너 실행자마다 다르지만, 컨테이너 실행자는 'tty', 'pts'를 지원하지 않는다.

```
$ podman exec -it <POD_ID> -- /bin/bash
```

COMMIT

현재 사용중인 컨테이너를 이미지로 저장한다.

```
$ podman stop centos-httpd
$ podman commit centos-httpd commit-httpd
Getting image source signatures
Copying blob 53498d66ad83 skipped: already exists
Copying blob e41246bc487f skipped: already exists
Copying blob d320f5da4f70 skipped: already exists
```

COMMIT

실행중인 컨테이너를 이미지로 다시 커밋한다. 하지만, 이 방식은 실무에서 사용하지 않는다.

```
$ podman run -d --name my-httpd -p8080:80 commit-httpd
```

PODMAN CONTAINER

컨테이너 로그를 확인한다. 컨테이너가 시작하면 표준 출력/오류를 런타임으로 전달한다.

```
# podman container logs centos-httpd
=> sourcing 10-set-mpm.sh ...
=> sourcing 20-copy-config.sh ...
=> sourcing 40-ssl-certs.sh ...
---> Generating SSL key pair for httpd...
AH00558: httpd: Could not reliably determine the server's fully qualified domain
name, using 10.88.0.19. Set the 'ServerName' directive globally to suppress this
message
[Sun Feb 05 06:47:31.399659 2023] [ssl:warn] [pid 1] AH01909: 10.88.0.19:8443:0
server certificate does NOT include an ID which matches the server name
```

PODMAN CONTAINER

컨테이너에 마운트 혹은 바인딩(mount --bind)이 된 자원을 확인한다.

```
$ podman container mount
77da2c1e24e8
/var/lib/containers/storage/overlay/1b003f63da8767dce7dacdcc431045ef46264a003047efeb
9c032061e1e7506c/merged
a98f2a616cfa
/var/lib/containers/storage/overlay/9e781c2758890d0d7e9328cd6c69674cec7a1c7840ed06e3
79d847637507abc1/merged
```

PODMAN CONTAINER

동작중인 컨테이너를 일시적으로 중지한다.

```
$ podman container pause centos-httpd
77da2c1e24e84283547e6b7defcdf80231b2833d28c18fa19bf5a0b222c88e85

$ podman container ls
a98f2a616cfa  quay.io/xinick/containerlab/httpd:latest  /bin/sh -c httpd ...  3
hours ago  Up 3 hours ago  0.0.0.0:8080->80/tcp  gallant_mahavira
```

PODMAN CONTAINER

일시 중지된 컨테이너를 다시 시작 시 사용한다.

```
$ podman container unpause
```

PODMAN CONTAINER

사용하지 않는 컨테이너를 모두 제거한다.

```
$ podman container prune
WARNING! This will remove all non running containers.
Are you sure you want to continue? [y/N]
```

PODMAN CONTAINER

컨테이너 이름을 변경한다.

```
$ podman container rename centos-httpd centos-apache
$ podman container ps
> /bin/sh -c httpd ...  29 seconds ago  Up 29 seconds ago  0.0.0.0:8080->80/tcp
centos-apache
```

PODMAN CONTAINER

동작중인 컨테이너를 중지 후 다시 시작한다.

```
$ podman container restart centos-apache
842293d27f7d4bfe62c433fb50bb1a477944335cd65a8d94ee97742be46b1b2b
```

PODMAN CONTAINER

현재 상태를 체크포인트로 저장 합니다.

```
$ podman container checkpoint --keep centos-httpd
1dcb10cfe96a65ffe63f2a8fba97d93ad7a1133b09372e5f4d347c4a5cc737bf
```

PODMAN CONTAINER

특정 시점으로 컨테이너 상태를 변경한다. 이미지나 혹은 동작중인 컨테이너에 적용이 가능하며, 적용 전 'checkpoint'명령어로 적용된 목록을 확인한다.

```
$ podman container restore centos-httpd
1dcb10cfe96a65ffe63f2a8fba97d93ad7a1133b09372e5f4d347c4a5cc737bf
```

command:stats

컨테이너 상태를 확인 합니다. 이 정보는 systemd에서 cgroup으로 모니터링한다. 매 5초마다 정보를 갱신한다.

```
$ podman container stats
```

```
ID            NAME          CPU %       MEM USAGE / LIMIT  MEM %       NET IO          BLOCK IO      PIDS        CPU TIME    AVG CPU %
1dcb10cfe96a  centos-httpd  0.04%       32.48MB / 2.728GB  1.19%       796B / 1.836kB  32.77kB / 0B  213         179.744ms   0.09%
```

command:top

네임스페이스에서 동작하는 프로세스의 CPU 사용률 및 동작 시간을 확인한다.

```
$ podman container top <CONTAINER_ID> <NAME>
```

```
[root@container ~]# podman container top centos-httpd
USER        PID         PPID        %CPU        ELAPSED          TTY         TIME        COMMAND
root        1           0           0.000       7m2.357571898s   ?           0s          httpd -DFOREGROUND
apache      2           1           0.000       7m2.357666755s   ?           0s          httpd -DFOREGROUND
apache      3           1           0.000       7m2.357688937s   ?           0s          httpd -DFOREGROUND
apache      4           1           0.000       7m2.35770685s    ?           0s          httpd -DFOREGROUND
apache      5           1           0.000       7m2.357724764s   ?           0s          httpd -DFOREGROUND
```

command:mount

바인딩된 루트 **파일시스템**(root filesystem, /var/lib/containers/storage)디렉터리를 컨테이너에서 해제한다.
컨테이너 재시작은 필요 없으며, 네임스페이스 공간을 통해서 연결이 해제된다.

```
$ podman container unmount
$ podman inspect centos-httpd-volume --format '{{ .HostConfig.Binds }}'
[htdocs:/var/www/html/:rw,rprivate,nosuid,nodev,rbind]
$ podman container unmount centos-httpd-volume
```

command:wait

컨테이너의 실행 상태를 코트로 화면에 출력한다. 시스템에서 사용하는 시스템 리턴 코드와 동일하게 0,1 그리고 125와 같은 숫자로 표현한다.

```
$ podman container wait
$ podman container wait --condition=running centos-httpd
-1
```

연습문제

INFRA CONTAINER

POD

POD

포드만의 **"Pod Manager"**의 줄인 이름이다. 도커와 다르게 포드만 Pod기반으로 단독 컨테이너 생성할 수 있도록 환경을 제공한다. 또한, 포드만은 **대몬 리스**로 Pod를 관리한다.

포드만에서 Pod는 "pause"라는 애플리케이션을 사용하며, 이를 "pause container"라고 부르기도 하며, 혹은 "infra container"라고 부르기도 한다. 이 부분은 앞서 이야기 했던 부분.

포드 컨테이너 혹은 인프라 컨테이너 동작 조건은 아래와 같다.

1. 인프라 컨테이너는 "컨테이너 모니터 프로세스"를 가지고 있다. 모든 Pod는 "conmon"이라는 프로그램을 통해서 라이프사이클을 관리한다.
2. conmon은 OCI런타임을 통해서 실행이 되며, OCI 사양 파일 기반으로 레이어를 생성한다. 이를 통해서 rootfs를 구성한다.
3. conmon를 통해서 컨테이너를 모니터링하고 종료 시, 종료 값를 반환한다.
4. 사용자가 conmon를 통해서, 컨테이너에 연결 시, 발생되는 표준 출력 및 표준 오류 내용을 전달한다.
5. 표준 출력 및 표준 오류로 발생한 내용은 컨테이너 로그로 남긴다.

포드 명령어

하위 명령어	설명
`create`	새로운 Pod를 생성한다.
`exists`	Pod가 존재 하는지 확인한다.
`inspect`	Pod의 정보를 JSON형태로 런타임 정보를 출력한다.
`kill`	Pod에서 사용하는 "pause"를 종료한다. 별도로 값을 명시하지 않으면 15번.
`list`	Pod의 목록을 출력한다.
`logs`	Pod에서 발생한 기록을 확인한다.
`pause`	동작중인 Pod를 잠시 정지한다.
`prune`	중지된 Pod 및 Container를 제거한다.
`restart`	Pod를 재시작 한다.

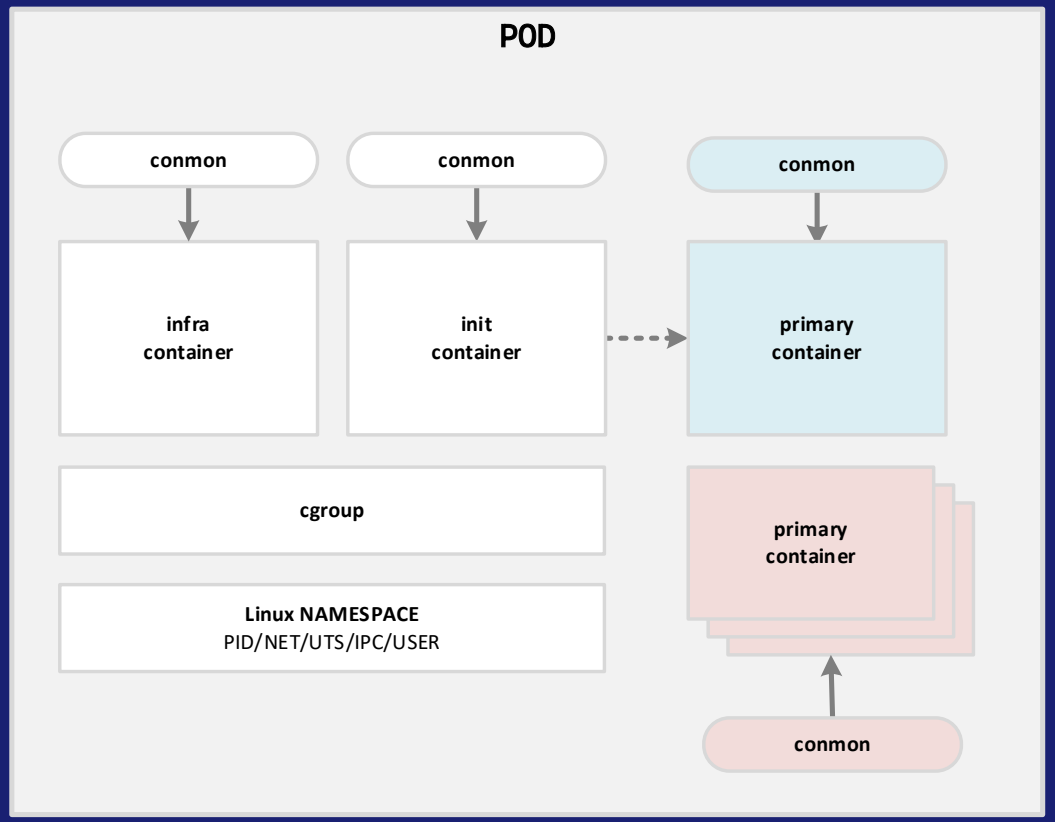
포드 명령어

하위 명령어	설명
`rm`	Pod를 제거한다.
`state`	Pod상태 정보를 사용자가 보기 편하게 렌더링하여 출력한다.
`start`	Pod 시작.
`stop`	Pod 중지.
`top`	Pod의 자원 사용 상태를 본다.
`unpause`	일시 정지된 Pod를 다시 시작한다.

POD동작 방식

POD 동작 방식은 다음과 같다.

1. "infra container" 실행
2. "init container" 실행
3. "primary container" 실행
4. "side car container"를 통한 모니터링

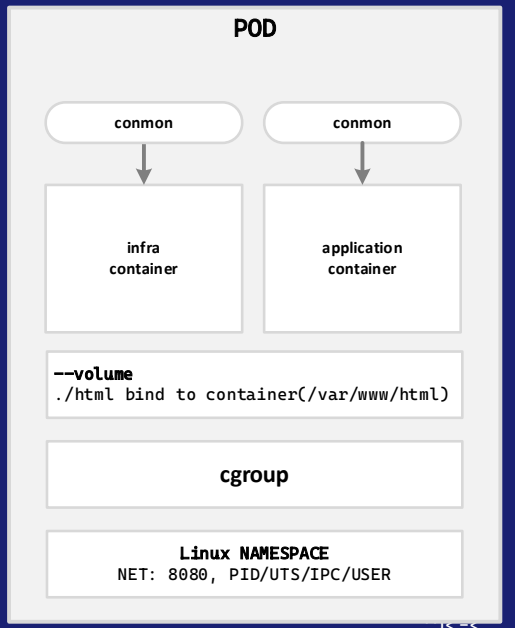


POD 컨테이너 생성

포드만에서 인프라 컨테이너와 함께 컨테이너 생성하는 방법은 다음과 같다.

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

일반 컨테이너 생성 명령어와 별반 차이가 없지만, Pod를 생성 후, 볼륨 및 네트워크 포트를 네임스페이스에 pause 에 전달한다.



POD+CONTAINER 생성

Pod에 컨테이너를 같이 연결해서 실행한다.

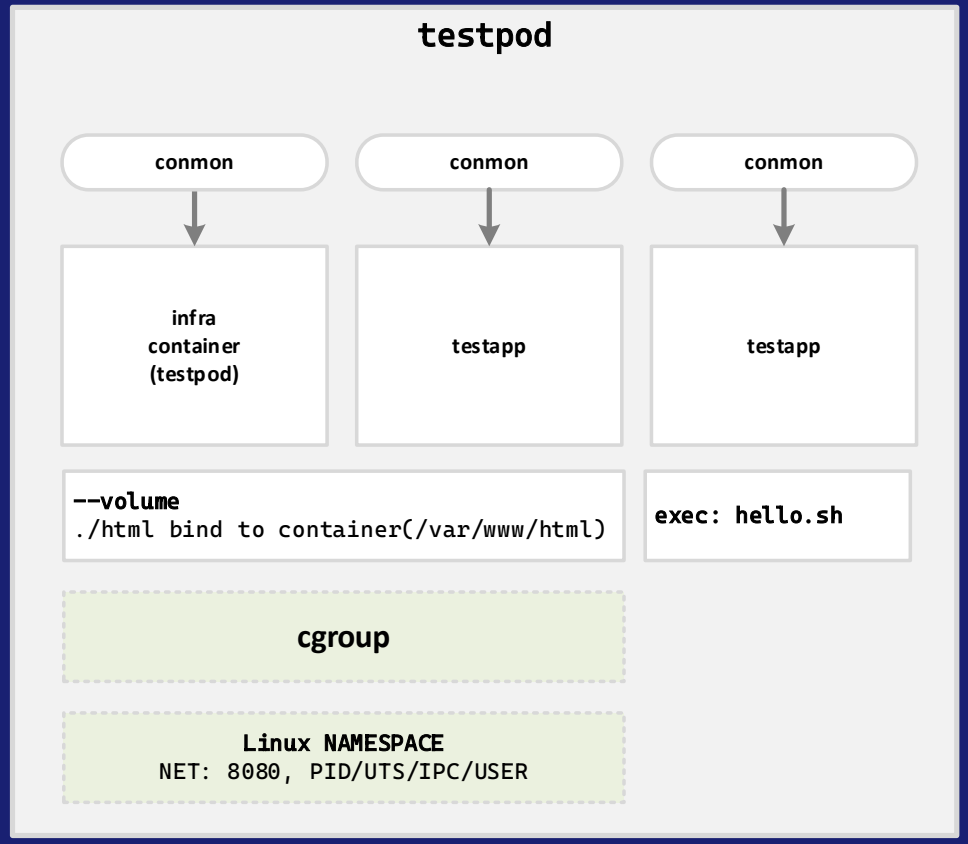
```
$ podman create --pod testpod --name testapp --rm --name testapp httpd
```

현재 컨테이너는 매우 기본적인 Pod생성 후 애플리케이션 컨테이너와 연결이 된다.

```
$ podman create --pod testpod --name testapp --workdir /var/www/html httpd hello.sh
```

위와 같이 실행이 되는 경우 다음과 같은 방식으로 동작한다. 임의로 "/var/www/html"에 'hello.sh'라는 파일을 생성 및 실행 한다. "hello.sh"가 실행이 될 때, 사이드 카(side car)로 컨테이너 이미지 "workdir"에 있는 'hello.sh'를 실행한다.

POD+CONTAINER 생성



POD+CONTAINER

Pod에 컨테이너를 같이 연결해서 실행한다.

```
$ podman create --pod testpod --name testapp --rm --name testapp httpd
```

현재 컨테이너는 매우 기본적인 Pod생성 후 애플리케이션 컨테이너와 연결이 된다.

```
$ podman create --pod testpod --name testapp --workdir /var/www/html httpd hello.sh
```

위와 같이 실행이 되는 경우 다음과 같은 방식으로 동작한다. 임의로 "/var/www/html"에 'hello.sh'라는 파일을 생성 및 실행 한다. "hello.sh"가 실행이 될 때, 사이드 카(side car)로 컨테이너 이미지 "workdir"에 있는 'hello.sh'를 실행한다.

POD Network

포드에 네트워크를 생성 및 구성하려면 다음과 같이 실행한다.

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

Pod에 컨테이너를 연결하기 위해서는 다음과 같이 명령어를 실행한다.

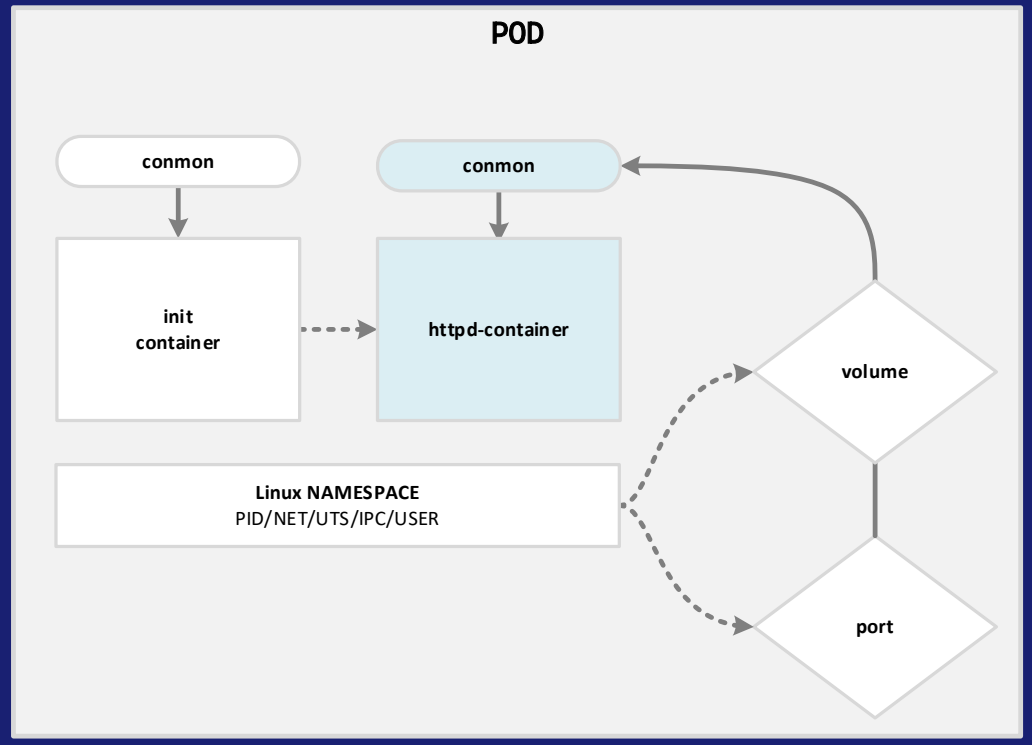
```
# podman run -d --pod httpd-pod --name httpd-container --rm quay.io/fedora/httpd-24
```

실행이 완료가 되면, 올바르게 포트 및 볼륨이 연결 및 구성이 되었는지 명령어로 확인한다.

```
# podman container ls
bfdab1daf26a  localhost/podman-pause:4.6.1-1699237644                       About a
minute ago  Up 5 seconds  0.0.0.0:8080->8080/tcp  31781a5b32e6-infra
528ada0faa9f  quay.io/fedora/httpd-24:latest           /usr/bin/run-http...  5
seconds ago       Up 5 seconds  0.0.0.0:8080->8080/tcp  httpd-container
```

POD NETWORK/VOLUME

포드만에서 Pod를 사용해서 쿠버네티스와 동일하게 Pod기반으로 생성 및 구성이 가능하다.



2023-05-30

Pod List

Pod의 목록 출력은 Container출력 방법과 동일하다.

```
$ podman pod ls
```

출력 시, "format"옵션을 통해서 특정한 필드만 선택적으로 확인이 가능하다. 'ps'명령어만 사용하는 경우, Pod, Container 둘 다 목록에 출력한다.

```
$ podman ps --all "{{.ID}} {{.Image}} {{.Pod}}"
```

포드만에서 Pod관련된 명령어는 'pod'하위 명령어로 지원하기 때문에, Pod 관련된 작업이 필요한 경우, 아래 표를 참조하여 Pod 자원을 관리한다.

Pod는 포드만에서 일반 컨테이너로 취급하기 때문에 애플리케이션/포드 컨테이너를 같은 컨테이너로 취급한다.

```
POD ID        NAME        STATUS      CREATED          INFRA ID      # OF CONTAINERS
31781a5b32e6  httpd-pod   Running     19 minutes ago   bfdab1daf26a  2
```

연습문제

APP CONTAINER

APPLICATION CONTAINER

애플리케이션 컨테이너

앞에서 Pod컨테이너에 대해서 학습하였다. 애플리케이션 컨테이너는 도커에서 이미지 기반으로 생성 및 실행하였던 일반 컨테이너이다.

이전에 학습하였던 컨테이너 관리 명령어를 통해서 컨테이너를 생성한다. 컨테이너 이미지 기반으로 애플리케이션을 생성하면 다음과 같이 화면에 출력이 된다.

```
Trying to pull quay.io/centos7/httpd-24-centos7:20230719...
Getting image source signatures
Copying blob 8f001c8d7e00 done   |
Copying blob c61d16cfe03e done   |
Copying blob 06c7e4737942 done   |
Copying config d7af31210b done   |
Writing manifest to image destination
d7af31210b288164c319bae740ca1281528390a3c5cee657e95f243670b49e6a
```

레지스트리 서버

이미지 레이어 다운로드

컨테이너 생성

컨테이너 생성은 앞에서 학습한 명령어로 생성이 가능하다.

1. create
2. run

위의 두 명령의 차이는 "create"명령어는 컨테이너 자원 생성, "run"은 생성 및 실행을 수행한다.

```
# podman create quay.io/centos7/httpd-24-centos7:20230719
# podman container ls -a
> 03540beab50e  quay.io/fedora/httpd-24:latest  /usr/bin/run-http...  44 seconds ago
Exited (0) 39 seconds ago   tender_hopper
# podman run quay.io/centos7/httpd-24-centos7:20230719
=> sourcing 10-set-mpm.sh ...
=> sourcing 20-copy-config.sh ...
=> sourcing 40-ssl-certs.sh ... # podman container ls
```

컨테이너 실행

모든 컨테이너는 표준 출력/입력 오류를 통해서 대화형 모드로 사용이 가능하다. 각 컨테이너는 실행이 되면, 네임스페이스 안에서 다음과 같은 자원을 생성하여 호스트와 공유가 가능하다. 아래 명령어로 컨테이너 생성 후 내부 자원도 확인한다.

```
# cd /proc/$$/self
# ls -l
total 0
lrwx------. 1 root root 64 Apr 14 18:46 0 -> /dev/pts/0
lrwx------. 1 root root 64 Apr 14 18:46 1 -> /dev/pts/0
lrwx------. 1 root root 64 Apr 14 18:46 2 -> /dev/pts/0
lrwx------. 1 root root 64 Apr 14 18:46 255 -> /dev/pts/0
# podman run -d quay.io/centos7/httpd-24-centos7:20230719
# podman ps
# podman exec -it doring_haibt /bin/bash
> ls -l /proc/$$/slef
```

컨테이너 실행

생성된 컨테이너 내부에 들어오면 가상머신과 다르게 프로세서는 매우 간결하게 동작한다. 루트 리스 컨테이너는 모든 장치에 접근이 불가능하다.

```
# ps -ef
default         1       0  0 09:49 ?        00:00:00 httpd -D FOREGROUND
default        52       1  0 09:49 ?        00:00:00 /usr/bin/cat
default        53       1  0 09:49 ?        00:00:00 /usr/bin/cat
default        54       1  0 09:49 ?        00:00:00 /usr/bin/cat
default        55       1  0 09:49 ?        00:00:00 /usr/bin/cat
# ls /sys/
block  bus  class  dev  devices  firmware  fs  hypervisor  kernel  module  power
```

하지만, 컨테이너는 기본적으로 자원 접근에 대해서 제한이 있기 때문에, 런타임에서 자원을 외부에서 접근 할 수 있도록 네임스페이스를 외부에 노출해야 한다.

컨테이너 자원 노출

컨테이너에 자원을 노출하기 위해서 다음과 같은 옵션을 제공한다.

1. Pod를 통한 자원 공유
2. Container에 네임스페이스를 통해서 직접적인 자원 공유

루트리스 컨테이너는 보통 포트를 통해서 네트워크 자원을 바인딩하기 때문에, 아래 명령어로 확인이 가능하다.

```
# podman port <CONTAINER_NAME> <CONTAINER_ID>
```

앞서 사용한 명령어는 포트를 외부로 바인딩하지 않았기 때문에, 위의 명령어로 포트번호가 출력되지 않는다. 위에서 실행한 명령어에 다음과 같은 옵션을 사용하여 다시 실행한다.

```
# podman container ls
# podman stop adoring_haibt
# podmam rm adoring_haibt
# podman run -d --port 8080:8080 quay.io/fedora/httpd-24
# podman container ls
# curl localhost:8080
```

컨테이너 자원 노출

포트 노출하는 방법은 컨테이너 종류, 버전 혹은 위치에 따라서 다르다.

1. gssproxy
2. nftables(iptables)
3. docker-proxy
4. Linux Bridge

위의 도구를 통해서 호스트에 특정 포트로 들어오는 트래픽을 컨테이너 애플리케이션으로 전달한다. 기본적으로 런타임에서 명시한 포트는 아래와 같이 라우팅이 구성된다.

컨테이너 포트

컨테이너는 가상 장치를 생성 후 리눅스 브릿지를 통해서 라우팅 및 포트 관리를 한다. 아래와 같이 명령어를 통해서 확인이 가능하다.

```
# ip link sh podman0
15: podman0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP mode
DEFAULT group default qlen 1000
    link/ether 86:d9:b0:33:35:16 brd ff:ff:ff:ff:ff:ff
# ip link show veth0
16: veth0@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master
podman0 state UP mode DEFAULT group default qlen 1000
    link/ether 86:d9:b0:33:35:16 brd ff:ff:ff:ff:ff:ff link-netns netns-6aa57521-
1d54-fb5a-b042-e8cee5122ebf
```

컨테이너가 생성이 되면서 런타임은 자동적으로 가상장치(veth)를 생성하며, 이 장치는 네임스페이스 영역에서 생성 및 구성이 된다. 생성된 이 장치는 다시 리눅스 브릿지를 통해서 nftables로 데이터를 전달 후 라우팅을 구성한다

컨테이너 네트워크(브릿지)

포드만은 기본적으로 리눅스 브릿지 기반으로 컨테이너 포트를 제어한다.

```
# brctl show
bridge name     bridge id               STP enabled     interfaces
podman0         8000.86d9b0333516       no              veth0
# brctl showmacs podman0
port no mac addr                is local?       ageing timer
  1     86:d9:b0:33:35:16       yes                0.00
  1     86:d9:b0:33:35:16       yes                0.00
# brctl showstp podman0
veth0 (1)
 port id                8001                    state                forwarding
 designated root        8000.86d9b0333516       path cost                 2
 designated bridge      8000.86d9b0333516       message age timer         0.00
 designated port        8001                    forward delay timer       0.00
 designated cost           0                    hold timer                0.00
 flags
# podman network ls
NETWORK ID    NAME        DRIVER
2f259bab93aa  podman      bridge
```

컨테이너 네트워크(라우팅)

네임스페이스로 포트 구성 및 생성이 되면, 해당 포트는 노출된 포트, 즉, "8080/TCP"에 대해서 아래와 같이 라우팅 정보를 구성 및 생성한다. 이를 통해서 "SOURCE", "DESTINATION"정보를 확인하여 컨테이너 애플리케이션에 데이터를 전달한다.

```
# nft list table nat | grep dport
# Warning: table ip nat is managed by iptables-nft, do not touch!
                tcp dport 8080  counter packets 0 bytes 0 jump NETAVARK-DN-
1D8721804F16F
                ip saddr 10.88.0.0/16 tcp dport 8080 counter packets 0 bytes 0 jump
NETAVARK-HOSTPORT-SETMARK
                ip saddr 127.0.0.1 tcp dport 8080 counter packets 0 bytes 0 jump
NETAVARK-HOSTPORT-SETMARK
                tcp dport 8080 counter packets 0 bytes 0 dnat to 10.88.0.8:8080
```

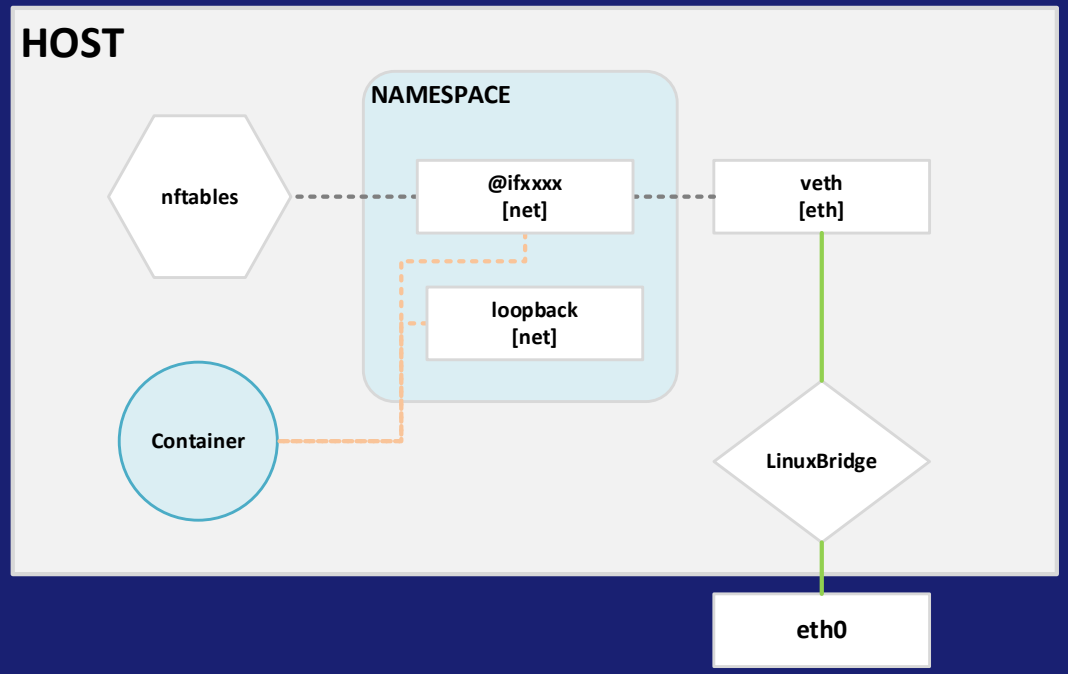
컨테이너 네트워크(내부정보)

컨테이너 내부에서는 다음처럼 네트워크를 설정 및 구성한다. 이를 확인하기 위해서 "centos"기반으로 컨테이너를 새로 만들어서 구성한다.

```
# podman run -it --name centos-network --rm -p8082:80 quay.io/centos/centos:stream9 /bin/bash
> dnf install ip-route
> ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default
qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0@if21: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP mode DEFAULT
group default qlen 1000
    link/ether 4e:d7:39:88:f3:3b brd ff:ff:ff:ff:ff:ff link-netnsid 0
> bridge link
> nft list tables
Operation not permitted (you must be root)
netlink: Error: cache initialization failed: Operation not permitted
```

컨테이너 네트워크

위의 내용을 그림으로 표현하면, 아래처럼 구성이 된다.



컨테이너 실행 옵션

컨테이너 실행 시, 다음과 같은 옵션을 많이 사용한다.

옵션	설명	예제
`--rm`	컨테이너가 중지가 되면 제거가 된다.	podman --rm
`--it`	대화형 모드로 진입한다.	podman -it
`-d`	"--detach"의 약자. 백그라운드로 시작한다.	podman run -d
`run`	컨테이너를 실행한다.	podman run
`--all`	중지된 컨테이너까지 화면에 출력한다.	podman container ls --all
`stop`	컨테이너를 중지한다. "-t"옵션을 통해서 타임아웃 시간 명시가 가능하다. 값이 "0"이면 기본적으로 SIGKILL로 들어간다.	podman stop -t 10 testapp
`--user`	컨테이너 접근 시, 사용할 사용자.	podman run --user apache
`-p`	컨테이너 외부 및 내부 포트 맵핑.	podman run -p 8081:8080
`--size`	컨테이너의 실제 크기를 확인한다. 여기서는 메타 정보의 크기는 제외된다.	podman container ls --size

컨테이너 구성 및 생성

컨테이너를 좀 더 복합적으로 구성 및 생성한다. 우리는 이 학습에서 앞으로 사용할 쿠버네티스를 고려해야 한다. 그러므로, 단순하게 애플리케이션만 컨테이너로 실행하는 것은 만족스럽지 않다. Pod와 함께 컨테이너를 생성한다.

아래 명령어를 보면, "--userns"라는 옵션을 사용하며, 여기에 "auto"라는 값을 명시 하였다. 컨테이너가 생성 및 동작이 되면 자동으로 포드의 네임스페이스에 사용자 영역을 구성한다.

```
$ podman run --pod new:pod_httpd -p port 8084:8080 quay.io/fedora/httpd-24
--volume /var/www/html --userns auto
```

위의 옵션은 컨테이너가 Pod와 함께 생성 및 구성이 되었을 때 사용이 가능하며, 컨테이너 자체는 위의 기능을 지원하지 않는다.

아래 페이지에서 컨테이너 생성 옵션에 대해서 설명한다.

컨테이너 유저 맵핑

컨테이너 사용자 매핑이 필요하면, 아래와 같이 설정한다.

키	사용자	컨테이너 사용자
auto	**$UID**	nil, 사용자의 UID를 컨테이너에 매핑하지 않습니다.
host	**$UID**	0, 기본 사용자는 컨테이너 루트 계정과 매핑 됩니다.
keep-id	**$UID**	$UID, 컨테이너와 호스트 사용자의 UID를 동일하게 매핑한다.
keep-id:uid=200,gid=210	**$UID**	200:210, 명시한 UID/GID를 컨테이너 사용자 및 그룹과 매핑한다.
nomap	**$UID**	nil, 호스트 사용자의 UID를 컨테이너에 매핑하지 않는다.

컨테이너 볼륨

컨테이너 볼륨 마운트 시, 다음과 같은 옵션 사용이 가능합니다.

값	설명
r w| r o	컨테이너에 쓰기 및 읽기로 볼륨 디스크를 설정한다.
z| Z	SELinux 컨텍스트를 적용 및 허용한다.
[O]	오버레이 모듈을 통해서 컨테이너에 마운트 한다. 컨테이너 생성시 이 방식으로 구성 및 생성한다.
[U]	이 옵션을 사용하면 호스트 기반의 UID/GID변경을 합니다. 컨테이너가 많을수록 컨테이너 생성 속도가 느려 집니다.
[no] copy	컨테이너 내부 볼륨에 내용을 복사 혹은 복사하지 않는다.
[no] dev	컨테이너 프로세스는 볼륨 내용에 접근이 가능하다. 기본값은 "nodev"로 된다.

컨테이너 볼륨

`[no]exec`	"noexe"마운트가 된 볼륨은 컨테이너 프로세스가 사용할 수 없다.
`[no]suid`	"nosuid"옵션이 활성화 되면, 컨테이너는 "nosuid" 권한을 변경할 수 없습니다.
`[r]bind`	"rbind"옵션이 활성화 되면, 하위 디렉터리까지 재귀적으로 연결이 됩니다. 기본적으로 이 옵션은 사용하지 않습니다.

`[r]shared`
`[r]slave`
`[r]private`
`[r]unbindable`
	컨테이너에 마운트는 기본적으로 private으로 된다. 볼륨이 공유(share)으로 필요한 경우 적절한 옵션을 선택해서 사용한다.

`idmap[=options]`	idmap=uids=0-1-10#10-11-10;gids=0-100-10, 세세하게 매핑이 필요한 경우 사용한다.

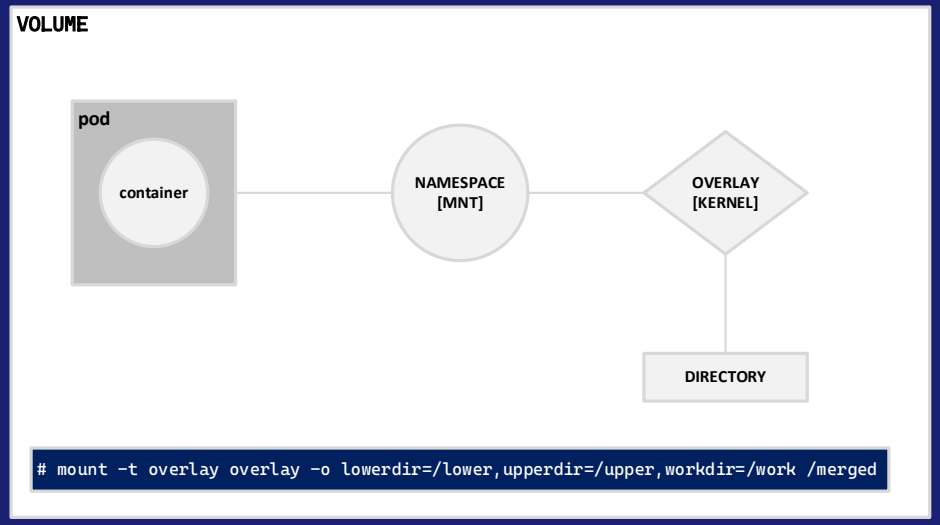
VOLUME

디스크

디스크

포드만에서 보통 **볼륨**이라고 표현하며, 다른 표현은 **컨테이너 디스크**라고 표현하기도 한다. 다만, 컨테이너에서는 디스크라는 개념이 없기 때문에 **컨테이너 볼륨**이라고 말하는 올바른 호칭이다.(보통 편의상 컨테이너 디스크라고 부른다.)

컨테이너에서 블록장치 및 볼륨 연결은 다음과 같은 방법으로 지원한다. 오버레이 장치는 일반적으로 사용하는 'mount'명령어로 구현이 가능하다.



컨테이너에서 사용하는 장치는 커널에서 오버레이 모듈을 통해서 특정 디렉터리 위치에서 마운트 및 바인딩을 한다. 맨 처음 생성되는 컨테이너는 오버레이 기반으로 구성이 되며, 이후 연결되는 디렉터리는 바인딩이 된다.

VOLUME VS BINDING

도커와 그리고 포드만 CLI형태로 오버레이 디스크에 마운트를 제공한다.

- 사용자 디렉터리를 컨테이너 내부로 마운트 바인딩 하는 방법.
  - namespace[mnt] + rbind
- 런타임에서 볼륨 생성 후, 컨테이너에 전달하는 방법.
  - /var/lib/containers/storage/volumes
  - /var/lib/containers/overlay

포드만은 'podman volume'명령어를 통해서 생성 및 관리가 가능하다. "/var/lib/containers/overlay"는 컨테이너에서 사용하는 오버레이 디렉터리들이 존재한다. 순수 데이터 저장용도로 사용하는 디렉터리는 아니다.

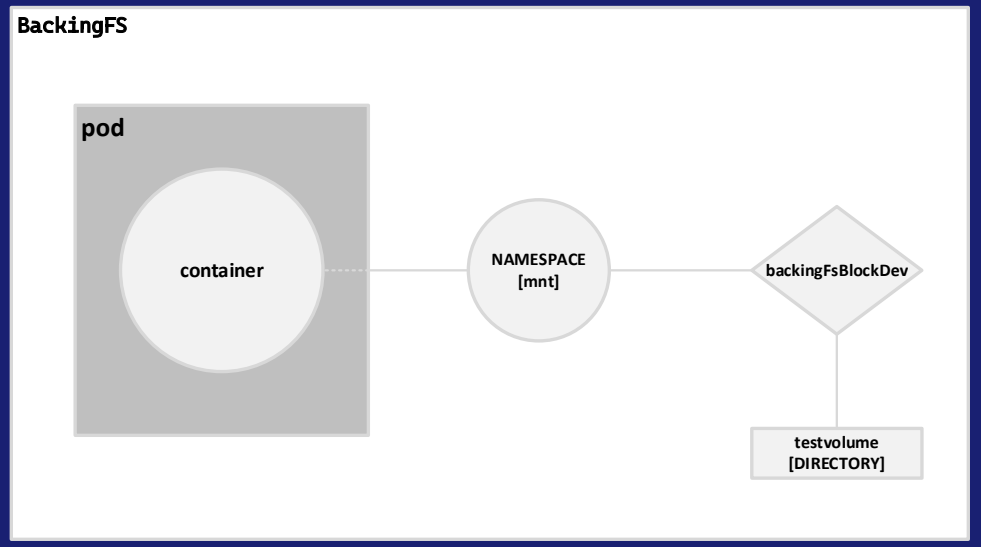
데이터는 일반적으로 "/var/lib/containers/storage/volumes"에 생성 및 저장이 된다.

ROOT DISK VOLUME

포드만은 도커와 동일하게 볼륨 디스크를 명령어로 생성이 가능하다. 볼륨은 컨테이너 엔진에서 관리하며 생성되는 위치는 다음과

```
# podman volume create testvolume
# podman volume ls
# ls -l /var/lib/containers/storage/volumes
```

생성된 위치에 "backingFsBlockDev"의 특별한 장치가 있는데, 이 장치는 컨테이너에 볼륨 생성 및 구성 시 사용하는 장치다. 이 장치를 통해서 오버레이 모듈로 구성된 컨테이너 디스크에 추가적으로 볼륨을 연결한다.



일반 볼륨

컨테이너에 특정 디렉터리 바인딩 시, 다음과 같이 명령어를 실행한다.

```
# mkdir htdocs
# podman run -d -v htdocs:/var/www/html -p 8082:8080 --rm --name volume-httpd
quay.io/fedora/httpd-24
# podman container inspect volume-httpd | grep -e volume -e htdocs
```

위의 명령어를 실행하면, 바인딩이 된 컨테이너 볼륨 정보가 화면에 출력된다.

```
# podman container inspect volume-httpd | grep -e volume -e htdocs
          "Name": "volume-httpd",
                    "Type": "volume",
                    "Name": "htdocs",
                    "Source": "/var/lib/containers/storage/volumes/htdocs/_data",
                    "htdocs:/var/www/html",
                    "volume-httpd",
                    "htdocs:/var/www/html:rw,rprivate,nosuid,nodev,rbind"
```

일반 볼륨 확인

호스트에서 생성된 디렉터리는 컨테이너 내부에 연결이 될 때, 기존의 디렉터리 자원을 네임스페이스 할당 및 분리를 한다. 옵션에 보면 다음과 같은 옵션이 컨테이너 디렉터리로 바인딩 되면서 구성 되었다.

옵션	설명
`rprivate`	컨테이너에 마운트는 기본적으로 private으로 된다. 볼륨이 공유(share)으로 필요한 경우 적절한 옵션을 선택해서 사용한다.
`rbind`	하위 디렉터리가 존재하면, 하위 디렉터리까지 컨테이너로 바인딩 합니다.
`nodev`	"noexe"마운트가 된 볼륨은 컨테이너 프로세스가 사용할 수 없다.
`nosuid`	"nosuid"옵션이 활성화 되면, 컨테이너는 "nosuid" 권한을 변경할 수 없습니다.

엔진 기반 볼륨

런타임 엔진에서 볼륨을 생성 후, 컨테이너에 전달이 가능하다. 포드만 명령어로 볼륨 생성 후, 컨테이너에 전달한다.

```
# podman container stop --all
# podman volume create test-volume
# podman run -d -v test-volume:/var/www/html -p 8082:8080 --rm --name
volume-httpd quay.io/fedora/httpd-24
# podman container inspect volume-httpd | grep -e volume -e htdocs
# echo "hello world" > /var/lib/containers/storage/volumes/test-
volume/_data/index.html
# podman container inspect volume-httpd | grep -e volume -e htdocs
# curl localhost:8082
```

볼륨 관리

앞에서 다루었지만, 런타임 엔진 기반으로 생성된 볼륨은 다음과 같이 관리가 가능하다.

```
# podman volume ls
# podman volume create
```

기본적으로 컨테이너는 **로컬**형태로 생성 및 구성이 된다. 로컬 디렉터리는 아래와 같다.

```
/var/lib/containers/storage/volumes
> testvolume
> htdocs
```

볼륨을 명령어로 생성하면 위의 위치에 생성이 되기 때문에, 직접 접근하여 관리를 하여도 되지만, 관리 및 안전성 문제로 가급적이면 명령어로 볼륨 정보를 관리한다.

볼륨 관리

포드만 볼륨 명령어는 다음과 같다. 아래 명령어는 **볼륨 생성** 및 데이터 **백업/복구**하는 방법이다.

```
# podman volume create backup-volume
# podman volume ls
# podman volume export backup-volume -o backup-volume.tar
# podman volume import backup-volume backup-volume.tar
```

볼륨이 어느 위치에 생성 및 구성이 되어 있는지 확인이 필요하면, 다음과 같이 조회가 가능하다.

```
# podman volume mount backup-volume
# podman volume unmount backup-volume
```

볼륨 관리

사용하지 않는 볼륨 디스크를 제거 시, 아래 명령어로 한번에 제거가 가능하다.

```
# podman volume prune
```

특별한 블록장치 연결

컨테이너에 블록 장치 연결은 직접적으로는 불가능하다. 하지만, 다음과 같은 옵션을 통해서 블록 장치를 임의로 컨테이너 내부로 연결이 가능하다.

```
# podman container run --rm --privileged --name bind-container -it --mount
type=bind,source=/dev/sdb,target=/data quay.io/centos/centos:stream9
> ls -r /sys/block/
> zram0  sr0  sdb  sda  dm-0
```

위의 옵션은 컨테이너를 생성하면서 특별한 권한(privileged)를 통해서 생성하며, "--mount"옵션을 통해서 특정 블록 장치를 컨테이너 내부에 연결이 가능하다.

```
# docker container run --rm --name device-container -it --device=/dev/sdb:/data
quay.io/centos/centos:stream9
> ls -ld /data
> brw-rw----.   1 root disk 8, 16 Apr 22 08:50 data
```

위의 명령어는 컨테이너를 생성하지만 "--device"옵션을 통해서 블록 장치를 컨테이너 내부 디렉터리로 마운트 한다. 다만, 특별권한 모드로 동작하는 게 아니기 때문에, 동작 방식은 조금 다르다.

스토리지 드라이버

포드만에서 사용하는 스토리지 설정은 아래 위치에 있다. 혹은 명령어로 확인이 가능하다.

```
/etc/containers/
podman info --format '{{ .Store.ConfigFile }}'
```

만약, 위의 위치에 "storage.conf"파일이 없으면, 다음과 같이 복사한다. 기본값으로 사용하는 경우, 굳이 파일을 복사할 필요가 없다.

```
# cp /usr/share/containers/storage.conf /etc/containers/storage.conf
```

설정 파일은 보통 다음처럼 구성이 있다. 아래 명령어로 확인이 가능하다.

```
# grep -Ev '^#|^$' storage.conf
[storage]
driver = "overlay"
runroot = "/run/containers/storage"
graphroot = "/var/lib/containers/storage"
[storage.options]
additionalimagestores = [ ]
pull_options = {enable_partial_images = "false", use_hard_links = "false", ostree_repos=""}
[storage.options.overlay]
mountopt = "nodev,metacopy=on"
[storage.options.thinpool]
```

스토리지 드라이버

컨테이너 런타임에서 사용하는 기본값은 다음 명령어로 확인이 가능하다. 굳이, 설정파일을 볼 필요가 없다.

```
# podman info
```

포드만은 루트리스 기능을 제공하는데, 루트리스 컨테이너는 보통 $XDG_CONFIG_HOME를 통해서 결정이 된다. 기본값은 사용자의 홈 디렉터리 밑에서 "$HOME/.config/containers/storage.conf"파일을 사용한다. 설정 파일은 아래와 같은 순서로 찾는다.

1. /etc/containers/storage.conf
2. $HOME/.config/containers/storage.conf
3. /usr/share/containers/storage.conf

기본 컨테이너 디렉터리는

루트리스 스토리지

루트리스 스토리지 설정을 위해서 다음과 같이 설정 파일을 변경한다.

```
# vi /etc/containers/storage.conf
> rootless_storage_path = "/var/rootless/$UID/storage"
# mkdir -p /var/rootless/1001/
```

변경 후, 별도의 재시작이 필요 없이 포드만 명령어로 설정 상태 확인이 가능하다.

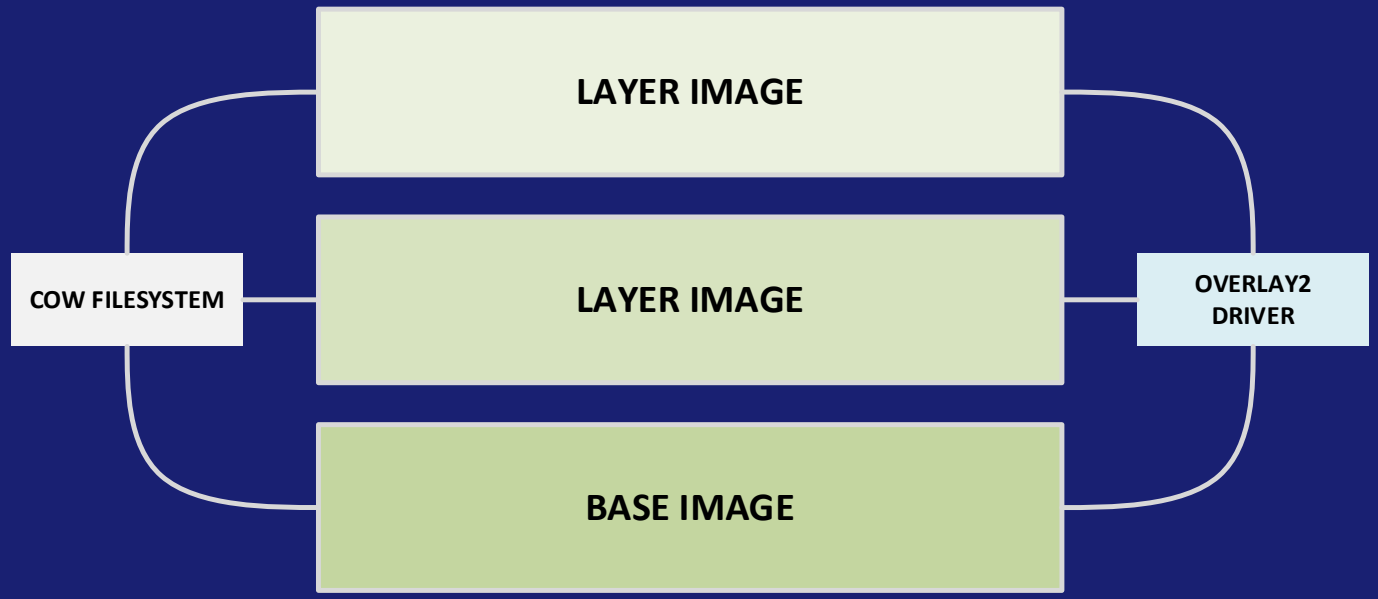
```
# podman info
rootless: true
  graphRoot: /var/rootless/1001/storage
  volumePath: /var/rootless/1001/storage/volumes
```

"storage.conf"를 위와 같이 설정하지 않으면, 기본적으로 "$HOME/.config/containers/"으로 생성한다. 만약, SELinux를 사용하는 경우, SELinux Context를 다시 재구성 해야 한다.

```
# semanage fcontext -a -e $HOME/.local/share/containers/storage/
# semanage fcontext -a -e /var/rootless/1001/
```

스토리지 장치

스토리지 장치는 다음과 같이 구성이 된다. 기본적으로 오버레이 모듈 및 COW통해서 구성하게 된다.



스토리지 장치

포드만에서 제공하는 스토리지 드라이버는 다음과 같다.

스토리지 드라이버	설명
`overlay2`	현재 포드만에서 기본적으로 사용하는 드라이버. 커널 수준에서 구현이 되며, "bind"와 비슷하게 구성이 되지만, 커널 가상 디바이스로 생성이 된다.
`vfs`	"virtual file system"약자. 이전에 많이 사용하였지만, 현재는 사용하지 않는다. 컨테이너 생성시 각 이미지를 전부 복사하여, 컨테이너를 생성한다. 이 방법은 성능이 매우 낮다.
`devmapper`	도커에서 사용하는 디바이스 맵퍼. 이전에는 overlay라는 모듈로 사용이 되었으며, 각각 레이어 별로 재구성이 되기 때문에 사이즈가 다른 형식보다 커질 수 밖에 없다.
`aufs`	이 드라이버는 업 스트림 커널에 통합이 된 적이 없다. 몇몇 배포판에서 사용이 가능하며, 초기 도커에서 지원하였다.
`btrfs`	네이티브 파일 시스템 기반으로 컨테이너 스냅샷 및 레이어를 관리 및 구성한다. 파일 시스템에서 제공하는 스냅샷 사용이 가능하기 때문에, 운영 시 많은 도움이 된다.
`zfs`	오라클 zfs파일 시스템. 이 파일 시스템 대신 btrfs 사용을 권장한다. 대다수 리눅스 배포판은 이 파일 시스템을 지원하지 않는다.
`xfs`	레드햇 고성능 파일 시스템. 레드햇은 xfs과 overlay2를 같이 사용한다. 고급 기능은 아직 지원하지 않으나, stratis를 통해서 고급 기능 구현이 가능하다.

FUSE STORAGE

리눅스에서 사용하는 기존 마운트 프로그램 혹은 overlay모듈은 기본적으로 커널에서 동작한다. 이러한 이유로 포드만은 더 이상 순수 커널 마운트를 사용하지 않고, 사용자 영역을 통해서 컨테이너 디스크 및 레이어를 구성한다.

이 때 사용하는 프로그램의 이름은 "fuse-overlayfs"라는 모듈이다. 루트리스 컨테이너가 구성이 되어서 동작하면, fuse를 통해서 디스크를 구성한다. 이를 통해서 HPC(High Performance Computing) 구현이 가능하다. 이 기능은 기본적으로 비활성화 되어 있기 때문에, 활성화 하기 위해서 다음과 같이 옵션을 수정해야 한다.

```
mount_program = "/usr/bin/fuse-overlayfs"
mountopt = "nodev,metacopy=on"
fs = "xfs"
```

수정 후, 다시 컨테이너를 재생성 및 재 시작하면, fuse기반으로 컨테이너가 동작한다.

```
# ps -ef | grep fuse-overlay
> /usr/bin/fuse-overlayfs -o lowerdir
```

스토리지 드라이버 구성

컨테이너 생성 시, 런타임에서 다음과 같은 옵션을 통해서 컨테이너 디스크 구성이 가능하다. 이 설정은 앞서 이야기한 "/etc/containers/storages.conf를 통해서 수정 및 설정이 가능하다.

스토리지 드라이버	설명
`inode`	컨테이너 이미지에서 사용이 가능한 inode 개수.
`skip_mount_home`	"private bind"를 통해서 스토리지 홈 디렉터리에 연결을 하지 않는다.
`size`	컨테이너 이미지의 최대 크기.
`force_mask`	컨테이너 생성 시, 강제로 마스크 퍼미션을 설정한다. ▪ private: 파일 시스템 퍼미션을 0700으로 설정한다. ▪ shared: 파일 시스템 퍼미션을 0755으로 설정한다.
`mountopt`	컨테이너 디스크 구성 시, 사용하는 마운트 옵션을 명시한다.
`mount_program`	컨테이너 디스크 구성 시, 사용하는 마운트 프로그램. 기본은 일반 mount명령어를 통해서 커널 기반으로 구성하거나 혹은 fuse기반으로 사용이 가능하다.
`ignore_chown_errors`	UID오류가 발생하여도 무시한다. 보통 컨테이너는 /etc/subuid기반으로 생성한다.

컨테이너 이미지

이미지 관리

이미지 빌드

컨테이너 이미지

이미지 관리

2023-05-30

이미지 설정

포드만 이미지는 "/etc/containers/registries.conf"를 통해서 관리가 된다. 정확히는 이미지를 내려받기 정보이다.

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

혹은 사용자에 따라서 다르게 저장소 정보를 사용하는 경우 다음 위치에 파일을 생성한다.

```
$HOME/.config/containers/registires.conf
```

"unqualified-search-registries" 옵션을 사용하는 경우, 불특정 다수의 저장소를 사용하여 접근한다. 기본적으로 docker.io, quay.io 그리고 fedoraproject.org를 사용해서 접근한다.

이미지 구성 방법 및 저장 위치는 위의 볼륨 목차에서 확인하였기 때문에, 자세한 내용은 넘어간다.

이미지 설정

이미지 관리를 위해서 앞으로 사용 내부 레지스트리 서버를 간단하게 docker-registry를 통해서 구현한다.

```
# setenforce 0
# podman run -d --rm --name docker-registry -p5000:5000 docker.io/opensuse/registry
# podman container ls
```

컨테이너 이미지를 올리기 위해서 registries.conf에 다음과 같이 수정한다. 수정하지 않으면 올바르게 이미지 접근이 되지 않는다.

```
[[registry]]
insecure = true
location = "localhost:5000"
```

올바르게 구성이 되었으면, 다음 명령어로 잘 동작하는지 확인한다.

```
# podman tag quay.io/fedora/httpd-24 localhost:5000/test/httpd:24
# podman push localhost:5000/test/httpd:24
# skopeo list-tags docker://localhost:5000/test/httpd
```

이미지 빌드 도구

자, 간단하게 준비가 되었으며, 이미지 빌드 도구에 대해서 이야기 한다.

현재, 표준 이미지 빌드 도구는 두 가지가 있다.

- podman build
- buildah

'podman build'은 이전 'docker build'와 동일하다. 'buildah'는 기존의 'docker build'기능을 별도의 이미지 빌드 도구로 만들었다. 이미지 빌드 방법은 뒤에서 학습하도록 한다.

앞서 구현한 컨테이너는 이미지 저장 및 배포 용도로 사용하는 레지스트리 서버이다. 여러 이미지 저장 서버가 있지만, 우리는 간단하게 "docker-registry"를 사용한다.

이미지 구조 확인

이미지의 구조를 확인하기 위해서 다음과 같은 명령어로 컨테이너 이미지 구조 확인이 가능하다.

```
# podman image tree c488a650ca45
Image ID: c488a650ca45
Tags:     [quay.io/fedora/httpd-24:latest localhost:5000/test/httpd:24]
Size:     419.2MB
Image Layers
├── ID: c12fdfa4ea5b Size: 186.9MB
├── ID: 84863322a4be Size: 34.59MB
└── ID: dde75f323505 Size: 197.7MB Top Layer of: [quay.io/fedora/httpd-24:latest
localhost:5000/test/httpd:24]
```

모든 이미지는 기본적으로 두 가지로 나누어진다. LowDir, UpperDir. 위의 tree명령어로 출력된 정보는 이미지에서 사용하는 기본 이미지(base image) 및 이미지 빌드 시 구성된 상위 이미지 이다.

이미지 변경 확인

포드만은 도커 런타임처럼, 기존 기본 이미지와 현재 기본 이미지 기반으로 생성 및 구성된 컨테이너에서 변경된 내용을 다음과 같은 명령어로 확인이 가능하다.

```
# podman diff volume-httpd
C /run/httpd
A /run/httpd/cgisock.1
A /run/httpd/httpd.pid
C /opt
C /opt/app-root
C /opt/app-root/etc
A /opt/app-root/etc/passwd
```

생성 및 변경이 된 내용이 있으면, "C", "A", "D"로 표시 된다.

이미지 목록

이미지 목록에 출력되는 정보는 모두 /var/lib/containers/storage/overlay-images에 저장이 되어있는 메타 정보 기반으로 출력이 된다. 다만, 이미지 크기는 실제 크기와는 조금 차이가 발생할 수 있다. 그 이유는, 모든 이미지에는 각 레이어 별로 메타정보가 저장되기 때문이다.

다만, 특정 이미지 목록만 확인하고 싶은 경우 아래처럼 사용이 가능하다. "httpd-24"이미지 기반으로 구성된 이미지 목록이 화면에 출력된다.

```
# podman images quay.io/fedora/httpd-24
REPOSITORY                  TAG         IMAGE ID      CREATED       SIZE
quay.io/fedora/httpd-24     latest      c488a650ca45  2 weeks ago   419 MB
localhost:5000/test/httpd   24          c488a650ca45  2 weeks ago   419 MB
```

이미지 정보

이미지 정보는 inspect명령어를 통해서 레이어 정보 및 이미지 메타 정보 확인이 가능하다.

```
$ podman inspect quay.io/fedora/httpd-24
 "NamesHistory": [
              "localhost:5000/test/httpd:24",
              "quay.io/fedora/httpd-24:latest"
          ]
```

이미지 파일 저장 및 복구

이미지는 파일 형태가 아닌 **디렉터리+파일+오브젝트** 형태로 구성이 되어 있다.

```
/var/lib/containers/storage/overlay/8853b21ed9ab4ab7fd6c118f5b1c11e974caa7e133a99981573434d3b6018cf0/diff
/var/lib/containers/storage/overlay-containers/3b479e41b4550c6badb77bf951f63bf0417d410cccd4850c85f8f5e809373b27
```

이 디렉터리를 파일 형태로 저장하기 위해서 아래와 같은 명령어로 구성해야 한다.

```
# podman save quay.io/fedora/httpd-24 -o fedora-httpd24.tar
```

반대로 파일을 컨테이너 런타임 저장소에 밀어 넣기 위해서 다음과 같이 실행한다.

```
# podman image load -i fedora-httpd24.tar
```

이미지 파일 복사

이미지를 다른 서버에 전달하려는 경우, 아래와 같이 명령어로 수행이 가능하다. 이 명령어는 포드만 버전에 따라서 지원이 되지 않을 수 있습니다.

```
# podman image scp quay.io/fedora/httpd-24 10.10.10.2::/home/root/httpd-24
# podman image scp quay.io/fedora/httpd-24 backup::
```

이미지 서명

생성한 이미지에 대해서 서명 및 인증을 위해서 아래와 같은 명령어로 수행이 가능하다. 다만, 서버에 접근 및 수정이 가능한 계정 권한을 가지고 있어야 한다.

```
# podman image sign --sign-by test docker://docker.io/library/rockylinux:9
```

이미지 레이어

앞에서 몇 번 사용 하였던 명령어이다. 컨테이너 이미지 레이어 정보를 화면에 출력한다.

```
# podman image tree quay.io/fedora/httpd-24:latest
Image ID: 7fbe9f13738f
Tags:     [quay.io/fedora/httpd-24:latest]
Size:     419.3MB
Image Layers
├── ID: c12fdfa4ea5b Size: 186.9MB
├── ID: a6adfa220b3f Size: 34.61MB
└── ID: cf9b48be54e9 Size: 197.8MB Top Layer of: [quay.io/fedora/httpd-
24:latest]
```

이미지 신뢰

모든 저장소는 policy.json를 통해서 저장소 정책을 관리한다. 아래 명령어로 저장소 신뢰성 확인이 가능하다.

```
# podman image trust show docker.io
TRANSPORT      NAME                          TYPE        ID
STORE
all            default                       accept
repository     registry.access.redhat.com    signed      security@redhat.com, security@redhat.com
https://access.redhat.com/webassets/docker/content/sigstore
repository     registry.redhat.io            signed      security@redhat.com, security@redhat.com
https://registry.redhat.io/containers/sigstore
docker-daemon                                accept
```

이미지 신뢰

기본적으로 설정 파일은 다음과 같이 되어 있다. 레드햇 계열 배포판은 docker, redhat.com기본적으로 허용하고 있다.

```
"transports": {
       "docker": {
           "registry.access.redhat.com": [
               {
                   "type": "signedBy",
                   "keyType": "GPGKeys",
                   "keyPaths": ["/etc/pki/rpm-gpg/RPM-GPG-KEY-redhat-release", "/etc/pki/rpm-gpg/RPM-GPG-KEY-redhat-beta"]
               }
           ],
           "registry.redhat.io": [
               {
                   "type": "signedBy",
                   "keyType": "GPGKeys",
                   "keyPaths": ["/etc/pki/rpm-gpg/RPM-GPG-KEY-redhat-release", "/etc/pki/rpm-gpg/RPM-GPG-KEY-redhat-beta"]
               }
           ]
       }
```

컨테이너 이미지

이미지 빌드

이미지 빌드

이미지 빌드는 두 가지 방법이 있다.

1. podman build
2. buildah bud

기존에 사용하던 방식 'docker build'과 동일하게 'podman build'가 사용이 가능하다. 또는, 이 기능을 분리 및 독립 시켜서 이미지 빌드 전용으로 만든 'buildah bud'를 사용해서 이미지 생성이 가능하다.

이미지 빌드를 하기 위해서 'image instruction'를 사용해야 하는데, 이때 사용하는 파일명은 기본적으로 **"Dockerfile"**이다. 하지만, 특정 제품 이름으로 파일을 구성하기 때문에 OCI에서는 이를 **"Containerfile"**로 변경하였다.

또한, 이미지 빌드 시, 파일 이름은 반드시 "Dockerfile", "Containerfile"로 설정해야 될 필요는 없다.

이미지 명령어

이미지 빌드 및 생성하기 위해서 다음과 아래와 같이 파일을 생성한다.

```
# vi Containerfile
ARG IMAGE_VERSION=39
FROM quay.io/fedora/fedora:${IMAGE_VERSION}
LABEL version="1.0"
EXPOSE 80
ENV SOURCE_DIR=/usr/local/src/kernel
RUN dnf install httpd -y
WORKDIR /var/www/html
VOLUME /var/www/html
COPY hdocs/index.html .
CMD ["/usr/sbin/httpd", "-DFOREGROUND"]
```

이미지 명령어 설명 1

간단하게 설명하면 다음과 같다. 아래는 제일 많이 사용하는 명령어이다.

명령어	설명
FROM	사용할 기본 이미지 입니다. 일반적으로 베이스 이미지라고 부르는 이미지를 사용한다.
ARGS	이미지 빌드 시, 인자 값이 필요한 경우 인자 값 설정을 한다.
COPY	로컬 파일 시스템에 있는 파일을 컨테이너 오버레이로 복사한다.
ADD	원격에서 파일을 내려받기 후, 컨테이너 오버레이로 복사한다.
LABLE	이미지에 레이블을 생성한다. 보통 메타 정보 생성 용도로 사용한다.
ENV	컨테이너 실행 시, 쉘에서 사용할 변수 값을 선언한다.
RUN	컨테이너 이미지 빌드 시, 명령어를 실행 합니다.
WORKDIR	컨테이너 프로세스가 동작하는 위치를 변경 합니다.
CMD	컨테이너 실행 시, 사용하는 프로그램을 명시한다. ENTRYPOINT와 같이 사용하는 경우, 명령어에서 사용하는 인자 값 처리를 한다.
ENTRYPOINT	컨테이너 실행 시, 사용하는 프로그램을 명시한다. ENTRYPOINT를 사용하는 경우 옵션 값은 보통 CMD에 선언한다.

이미지 명령어 설명 2

아래는 자주 사용하는 명령어는 아니지만, 알아 두면 유용한 명령어.

명령어	설명
`VOLUME`	컨테이너가 실행 시, 사용하는 볼륨 디스크 입니다. 없는 경우 자동으로 생성한다.
`RUN --mount`	이미지 빌드 시, 마운트가 필요한 경우 이 옵션을 통해서 연결 및 구성한다.
`RUN --network`	이미지 빌드 시, 기본 네트워크가 아닌, 다른 네트워크를 통해서 명령어를 실행한다.
`RUN --secret`	이미지 빌드 시, 시크릿 파일을 생성 및 구성한다. 이 명령어는 위의 "--mount"와 관계가 있다.
`ONBUILD`	이미지 빌드 시, 베이스 이미지에서 사용한 명령어를 다시 한번 실행한다. 이전에 사용한 명령어 기록이 있으면, 이전 이미지 레이어를 다시 활용한다.
`USER`	컨테이너 사용자를 변경한다. 컨테이너는 별도로 명시하지 않으면 root권한으로 동작한다.

이미지 빌드 시작

이미지 빌드는 다음과 실행한다.

```
# podman build .
# podman build -t fedora
# podman build -t localhost/fedora:httpd .
STEP 1/7: FROM quay.io/fedora/fedora
STEP 2/7: LABEL version="1.0"
--> Using cache
06eb474d1aa9f04d3df79c74e7271e354e5955f2f81ab124804856421ec18682
--> 06eb474d1aa9
STEP 3/7: EXPOSE 80
```

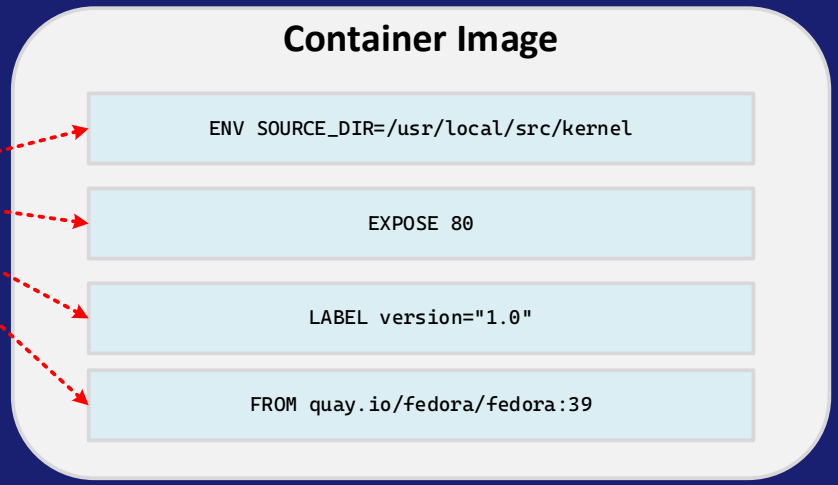
이미지 빌드 확인

이미지가 올바르게 빌드가 되었는지 다음 명령어로 확인한다.

```
# podman images
REPOSITORY                  TAG         IMAGE ID      CREATED         SIZE
localhost/fedora            httpd       87af73e5d75c  2 minutes ago   341 MB
```

이미지 빌드는 아래처럼 파일을 읽어와서 순차적으로 이미지 빌드를 시작한다. FROM 기본 이미지이다.

```
# vi Containerfile
ARG IMAGE_VERSION=39
FROM quay.io/fedora/fedora:${IMAGE_VERSION}
LABEL version="1.0"
EXPOSE 80
ENV SOURCE_DIR=/usr/local/src/kernel
```



이미지 실행

위의 Containerfile를 올바르게 실행이 되면 다음과 같이 확인이 가능하다.

```
# vi Containerfile
# podman build  -t localhost/fedora:httpd .
# podman images
# podman run -d --name fedora-httpd --rm --port 8080:80 localhost/fedora:httpd
# podman container ls
# podman volume ls
# podman inspect container
```

buildah

이미지 빌드는 앞서 사용한 포드만으로 가능하다. 다만, 이미지만 빌드 하기 위해서 포드만 설치는 너무 큰 공간 및 자원을 차지한다. 또한, CI/CD와 연동을 고려 하였을 때 독립적인 이미지 빌드 도구가 필요하다. 이러한, 이유로 나온 도구가 buildah(빌더)이다.

보통 발음 "**빌더아/빌더**"라고 발음하며, 공식 발음은 빌더 이다. 현재 OCI에서는 이미지 빌드 방법 및 도구로 빌더 사용을 권장한다.

빌더 설치

빌더 설치를 위해서 다음과 같은 단계를 진행한다.

```
# dnf install buildah
```

설치가 완료가 되면, 이전에 사용한 Containerfile를 통해서 다시 이미지 빌드를 시도한다. 반드시 띄어쓰기 해야 합니다.

```
# buildah bud .
# buildah build .
# buildah build -t localhost/fedora:httpd-2 .
# buildah bud -f Containerfile
# buildah images
```

빌더 활용

빌더 기반으로 이미지를 생성하면, 다음과 같이 이미지 테스트 및 확인이 가능하다.

```
# buildah containers --all
CONTAINER ID  BUILDER  IMAGE ID     IMAGE NAME                       CONTAINER NAME
44e63ebd8829           cfae2f4bcb67 localhost/fedora:httpd           great_poitras
7f65832e4e50           79b85e28ef15 localhost/fedora:httpd-2.        sad_tharp
```

위의 컨테이너 목록은 이미지 빌드 시 사용한 컨테이너이다. 컨테이너가 사용하는 디렉터리 및 레이어 정보는 다음 명령어로 확인이 가능하다.

```
# buildah mount 44e63ebd8829
/var/lib/containers/storage/overlay/4961a6c0358edaf66012c6071443b23b1c10b6d
5457e401ec81908a661a0d1ef/merged
```

빌더 메니페스트

쿠버네티스에서 사용하는 이미지에 메니페스트를 추가하기 위해서, 아래와 같이 실행한다.

```
# REPO=localhost/fedora/httpd
# buildah manifest create $REPO:24rev1
# buildah manifest add $REPO docker://$REPO:24rev1
# buildah manifest push --all $REPO:24rev1
```

혹은 추가적인 정보를 아래처럼 추가가 가능하다. 다만, 아래 명령어는 이 랩에서는 동작하지 않는다.

```
# buildah manifest annotate --arch x64 --annotation kubernetes=vanila --os
fedora--os-version 24 --variant v24 localhost/fedora/httpd:24
```

이미지 스크래치

스크래치 빌드는 더빌더에서 이미지 빌드와, 포드만에서 이미지 빌드에서 제일 큰 차이점은, 스크래치 빌드(scratch build)를 지원한다. 스크래치 빌드는 기반 이미지(base image)를 사용하지 않고, 처음부터 사용자가 직접 디렉터리 구성 및 생성한다.

```
# buildah from scratch
# scratchmnt=$(buildah mount working-container)
# echo $scratchmnt
# dnf install -y --releasever=9 --installroot=$scratchmnt rocky-release
# dnf install -y --setopt=reposdir=/etc/yum.repos.d \
      --installroot=$scratchmnt \
      --setopt=cachedir=/var/cache/dnf httpd httpd-core
# mkdir -p $scratchmnt/var/www/html
# echo "This is the first scratch container" > $scratchmnt/var/www/html/index.html
# buildah config --cmd "/usr/sbin/httpd -DFOREGROUND" working-container
# buildah config --port 80/tcp working-container
# buildah commit working-container localhost/scratch-httpd:latest
# buildah images
```

연습문제

이미지 조회

skopeo

SKOPEO

이미지 조회 및 관리를 위해서 사용하는 도구이다. 보통 이미지 태그 조회나 혹은 이미지 복사 및 백업 시 사용한다.

```
# dnf install skopeo -y
```

설치 된 명령어로 특정 이미지를 저장소에서 검색한다. docker.io는 접근에 제한이 발생하기 때문에, quay.io를 통해서 이미지 태그 목록을 조회한다.

```
# skopeo list-tags docker://quay.io/fedora/httpd-24
```

이미지 작업을 하기 위해서는 URI를 명시하여, 어떠한 프로토콜로 내려 받을지 결정해야 한다.

형식	설명
docker	도커 프로토콜 기반으로 검색한다.
docker-archive	저장된 도커 이미지를 다룰 시, 사용한다.

SKOPEO 복사

이미지를 파일로 내려받기 위해서 다음과 같이 실행한다.

```
# skopeo list-tags docker://quay.io/fedora/httpd-24
> 2.4
# skopeo copy docker://quay.io/fedora/httpd-24:2.4 docker-archive://httpd-24-24.tar
Getting image source signatures
Copying blob 95fe7b64e419 done
Copying blob b53509ff0308 done
Copying blob 532727926a66 done
Copying config 7fbe9f1373 done
Writing manifest to image destination
```

SKOEPO 이미지 검사

혹은 이미지 내려받기 하지 않고 검색하려는 경우 다음 명령어 조회가 가능하다. 메타 정보를 수집하는데 조금 시간이 조금 발생한다.

```
# skopeo inspect docker://quay.io/fedora/httpd-24:20240214
> {
>     "Name": "quay.io/fedora/httpd-24",
>     "Digest":
"sha256:e0a7fdacb8b010d2903d11ad51cef95343cb50b9aaa660479b29e18516573a1b",
>     "RepoTags": [
```

이미지 미러

이미지를 내부 서버로 미러가 필요한 경우, 'skopeo'를 사용하여 구현이 가능하다. 다만 "--src"에서 docker라고 선언하였기 때문에 따로 "docker://" URI선언이 필요하지 않는다.

```
# mkdir -p /tmp/fedora-httpd
# skopeo sync --src docker --dest dir --scoped quay.io/fedora/httpd-24
/tmp/fedora-http
```

쿠버네티스 이미지를 오프라인으로 복사를 하려면, 쿠버네티스 레지스트리 저장소는 sync를 허용하지 않기 때문에 목록을 작성 후 복사를 해야 한다.

```
# mkdir -p /tmp/kubernetes-image
# skopeo copy --insecure-policy --src-tls-verify=false --dest-tls-
verify=false --src --dest
```

연습문제

systemd 서브블록 컨테이너

systemd 컨테이너

서브블럭(systemd)

놀랍게도!?, 포드만은 systemd에서 서비스 형태로 컨테이너를 실행 할 수 있도록 한다. 물론, 사용자 세션이 종료가 되면 같이 종료가 되기 때문에 추가적으로 수정해야 될 부분들이 있다.

포드만은 systemd의 서비스 기반으로 동작한다. 현재 사용중인 포드 및 컨테이너가 있으면 다음처럼 화면에 출력이 된다. 이 중 "-infra"라고 붙어있는 접미어는 포드 컨테이너를 표현한다.

```
# podman generate systemd
31781a5b32e6-infra  (httpd-pod)  httpd-container     (httpd-pod)  sad_tharp
great_poitras                    httpd-pod
```

systemd서비스 생성

systemd밑으로 서비스 생성하기 위해서는 루트가 아닌 일반 사용자로 구성을 해야 한다.

```
# adduser container-user
# echo container-user | passwd --stdin container-user
# ssh container-user@localhost
$ podman run -d --rm --name systemd-container-service -p 8080:8080
quay.io/fedora/httpd-24
$
$ mkdir -p ~/.config/systemd/user
$ podman generate systemd --name systemd-container-service --files --new
$ mv systemd-container-service.service ~/.config/systemd/user
$ systemctl daemon-reload --user
$ systemctl start --user  container-systemd-container-service.service
$ systemctl status --user systemd-container-service
```

컨테이너 세션

하지만, 사용자가 세션을 종료하면 컨테이너 및 프로그램 실행이 중지가 되기 때문에, 백그라운드에서 동작하도록 설정해야 한다. 하지만, "linger"설정을 활성화 하면, 백 그라운드에서도 프로그램이 문제 없이 동작한다.

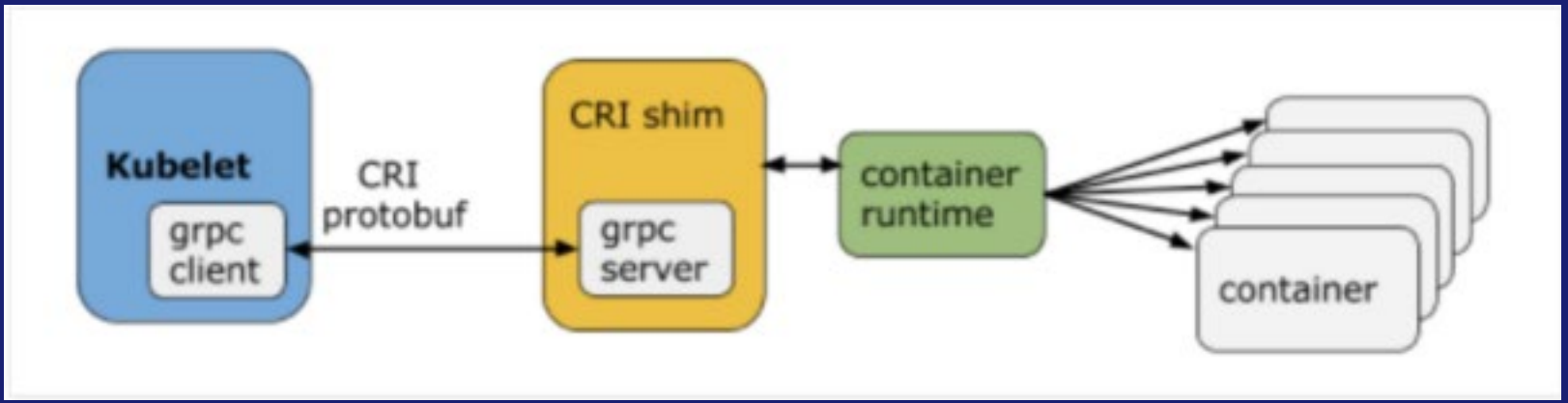
```
$ loginctl enable-linger container-user
$ loginctl show-user container-user | grep Linger
> Linger=yes
$ systemctl enable --user container-systemd-container-service.service
$ systemctl is-enabled --user container-systemd-container-service.service
> enabled
# curl localhost:8080
```

연습문제

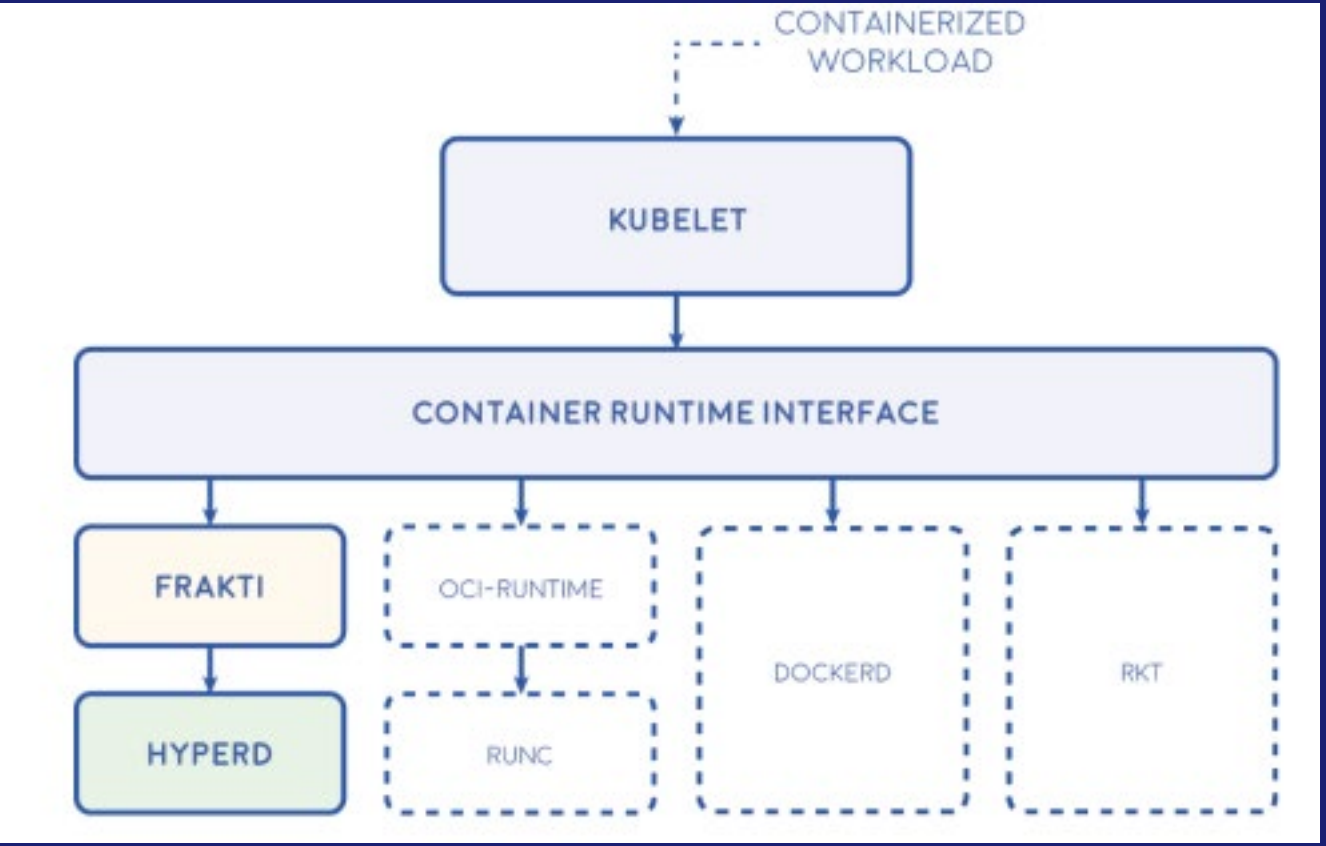
KUBELET RUNTIME

컨테이너 기초

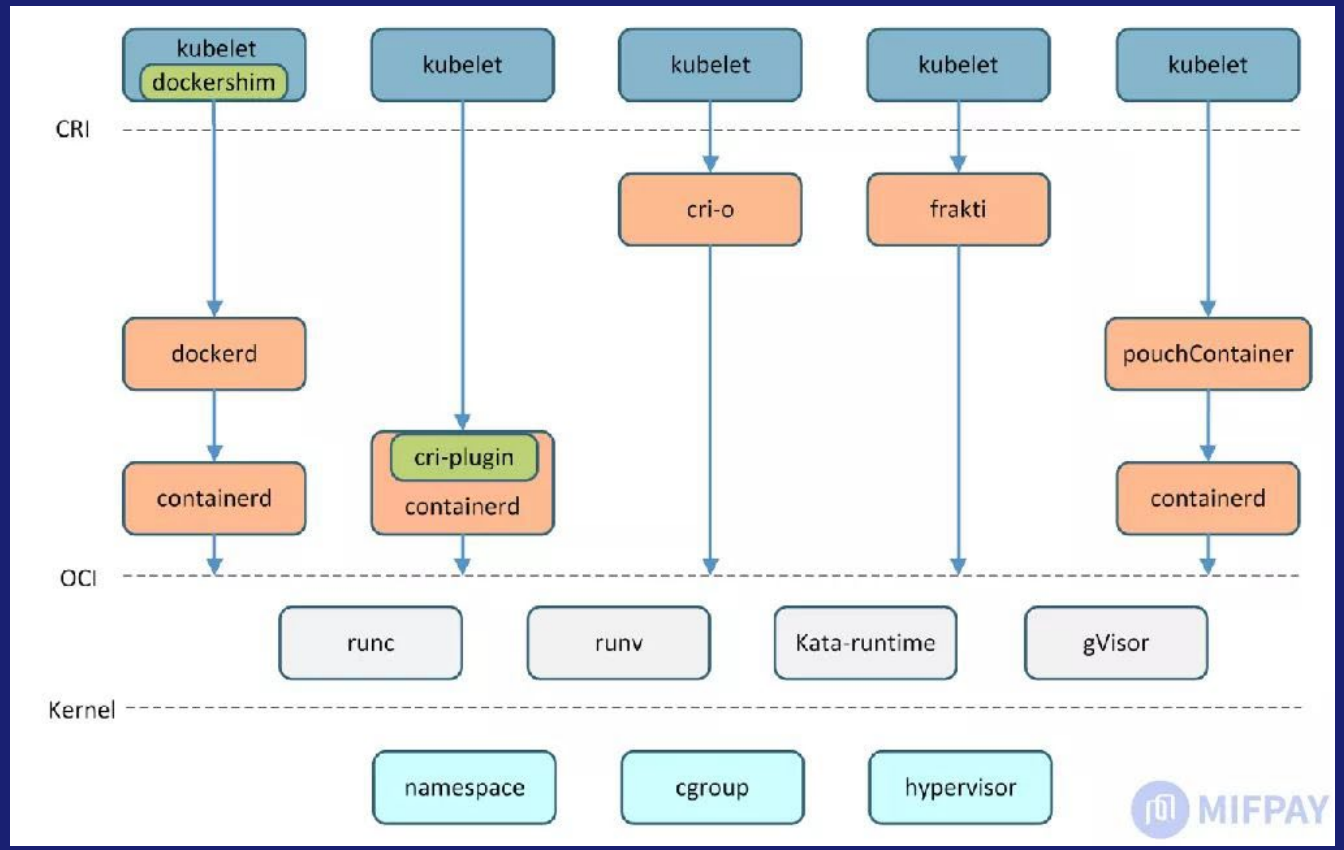
KUBELT + CRI STANDARD



KUBELET + CRI STANDARD



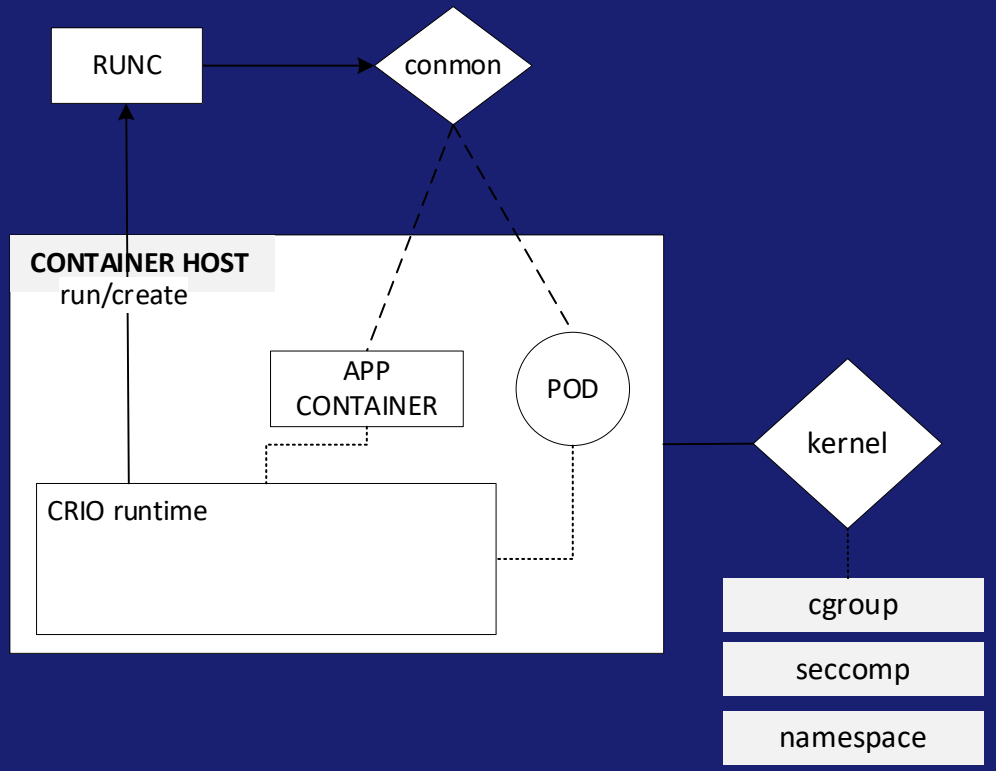
KUBELET TOP/DOWN LAYERS



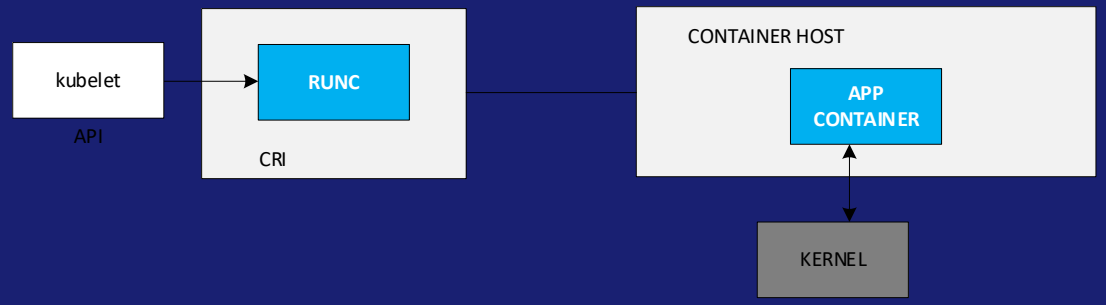
컨테이너 런타임

오픈소스 표준 컨테이너 엔진

RUNTIME


RUNC
conmon
CONTAINER HOST
run/create
APP
CONTAINER
POD
CRIO runtime
kernel
cgroup
seccomp
namespace

RUNTIME



PODMAN

podman

PODMAN

포드만은 docker의 대안으로 나온 컨테이너 런타임.

컨테이너 런타임은 데몬리스(daemonless)의 컨테이너 관리자이다.

자체적으로 API서비스를 가지고 있으며, 필요가 없는 경우 컨테이너 데몬을 실행할 필요가 없다.

PODMAN

포드만은 도커와 호환성을 제공한다.

io.podman.service, podman.service 데몬은 API를 제공.

- docker-compose
- docker-build
- volume
- network

PODMAN

- 포드만은 쿠버네티스 런타임으로 사용은 불가능함.
- 독립적인 구성만 가능하며, API기반으로 'DOCKER-EE'와 같은 기능을 제공함.

PODMAN

What is Podman? Podman is 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. Simply put: alias docker=podman

PODMAN

현재 사용하는 모든 컨테이너 시스템은 다음과 같은 표준을 따르고 있다. Podman(이하 앞으로 포드만)은 아래 사양을 따르고 있다.

OCI, Open Container Initiative

CNI, Container Network Interface

CSI, Container Storage Interface

CRI, Container Runtime Interface

DOCKER

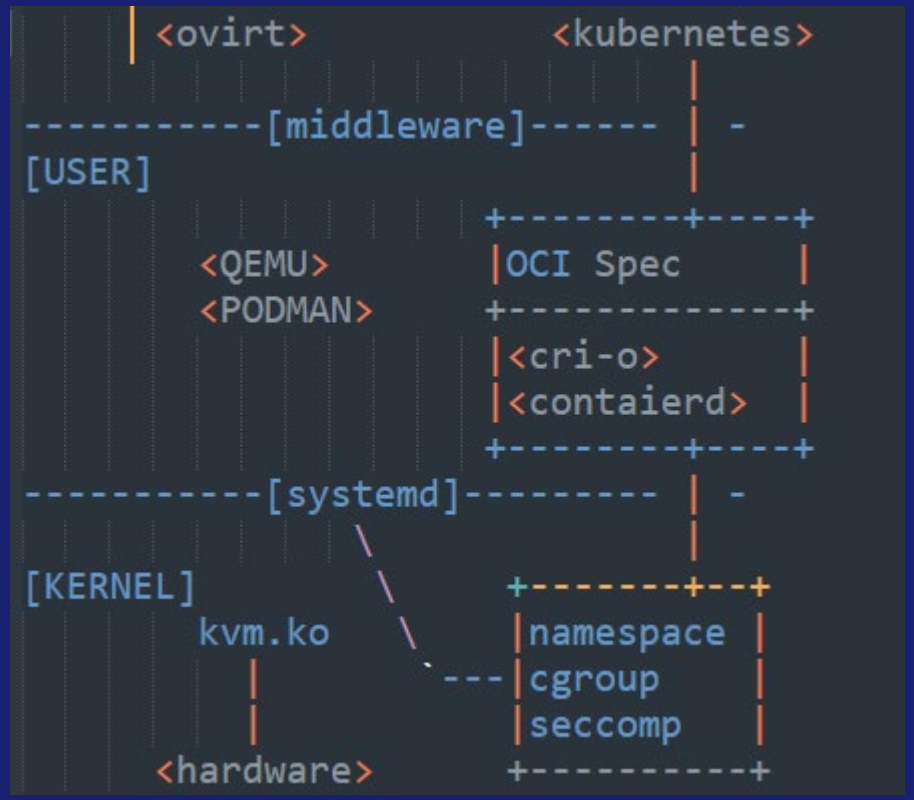
도커는 DOCKER CE, EE 두 가지 버전으로 지원.

현재는 Mirantis라는 회사로 인수 및 개발중지.

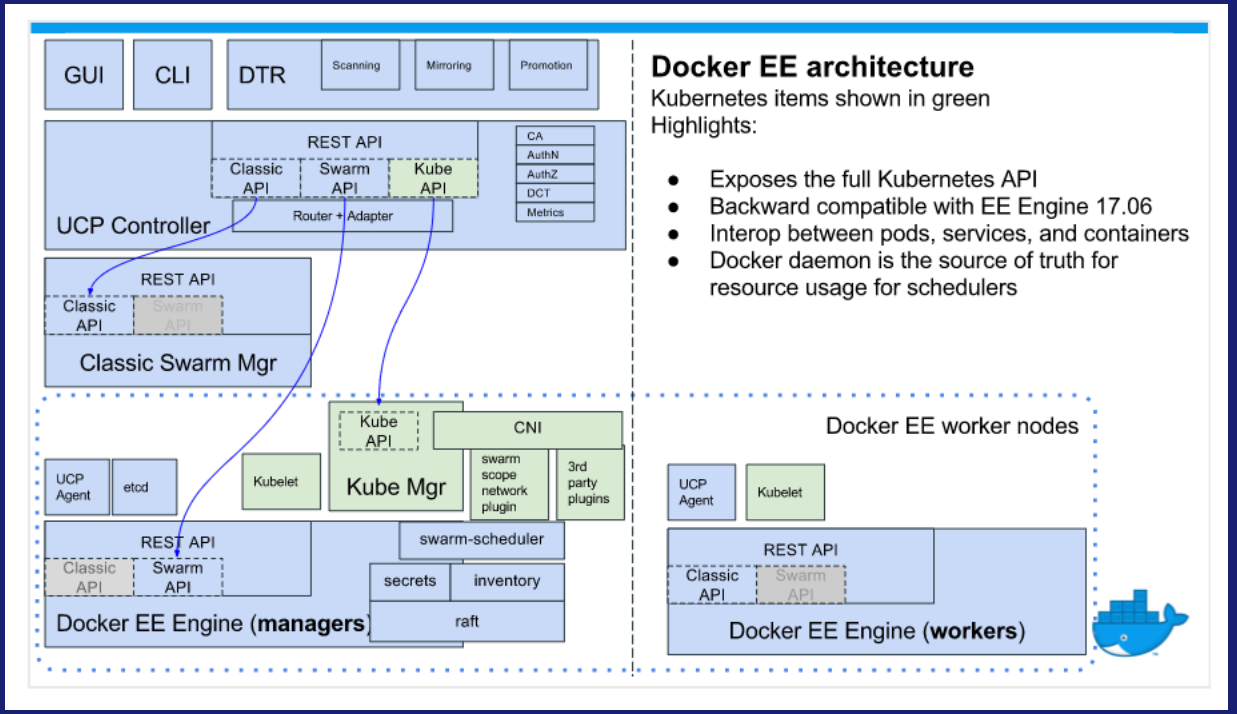
DOCKER-CE는 Containerd로 분리가 되었음.

Containerd는 현재 CRI/OCI기본 컨테이너 런타임 및 이미지 표준.

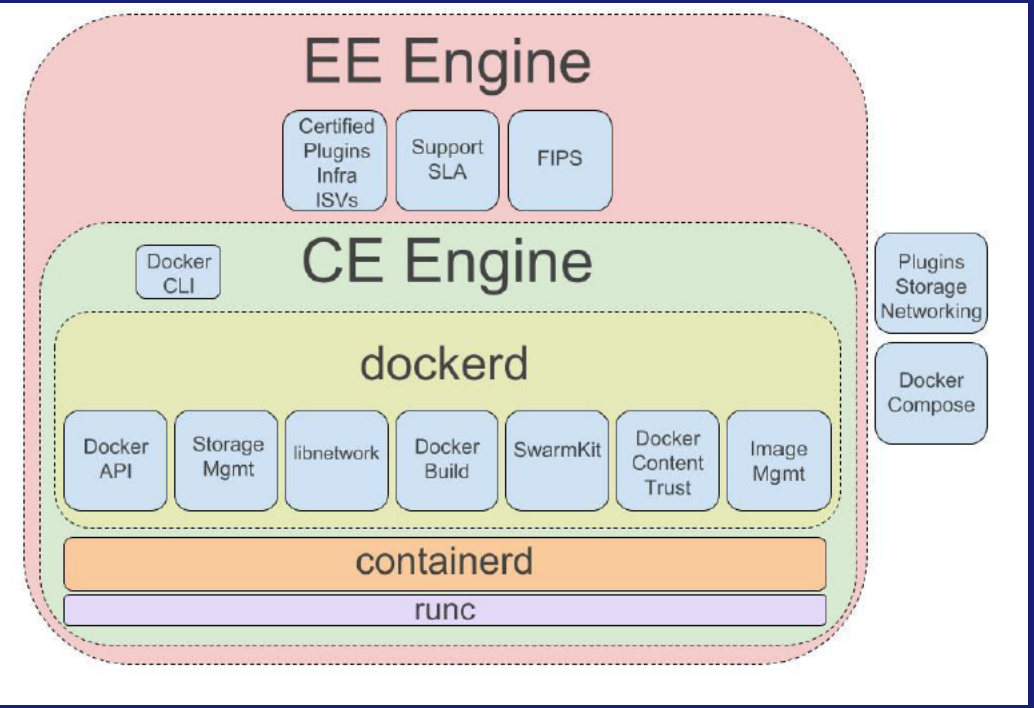
DOCKER


| <ovirt>
<kubernetes>
----------[middleware]------
| -
[USER]
<QEMU>
<PODMAN>
|OCI Spec
|<cri-o>
|<containerd>
----------[systemd]---------
| -
[KERNEL]
kvm.ko
|namespace
|cgroup
|seccomp
<hardware>

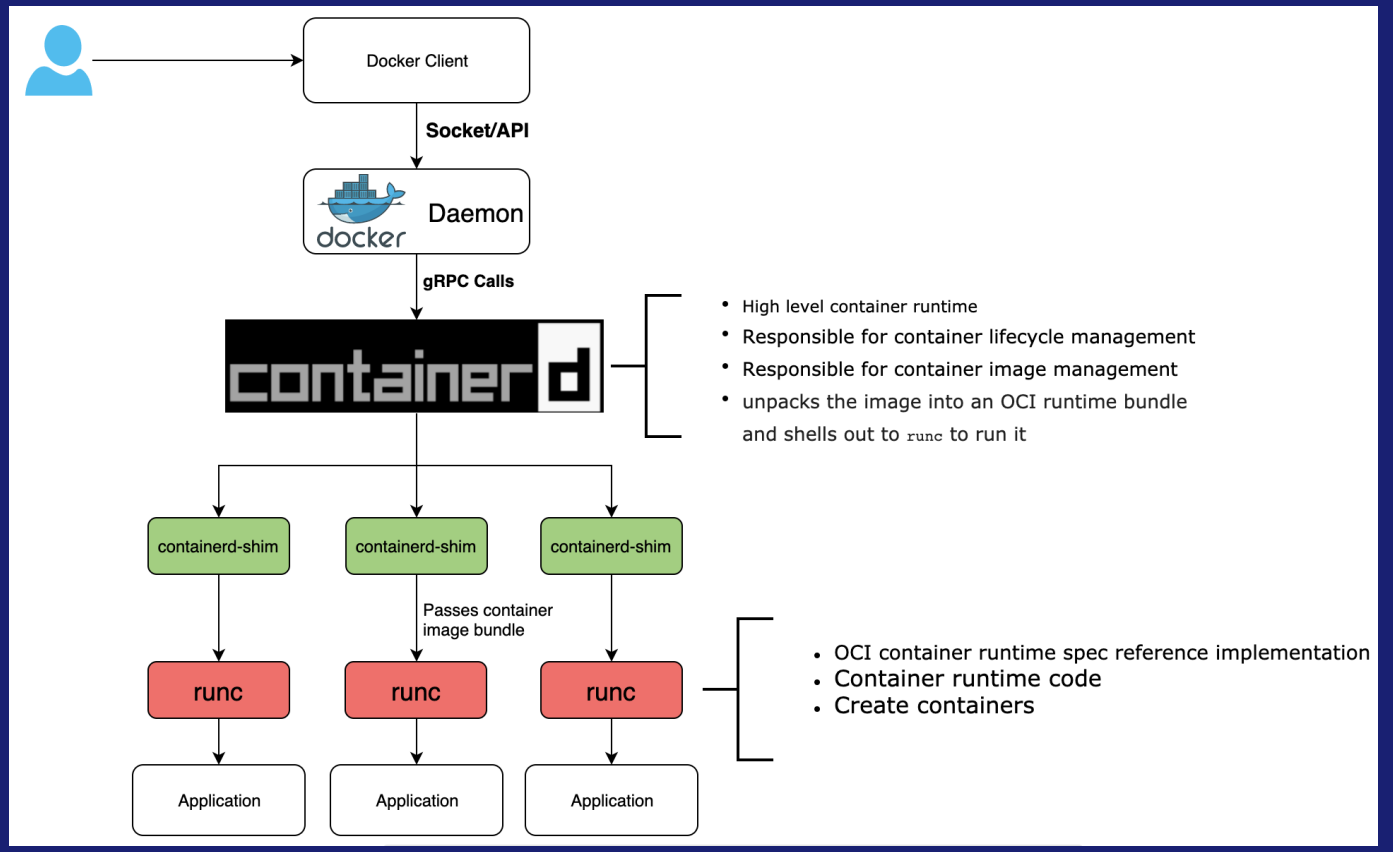
DOCKER CE/EE


GUI
CLI
DTR
Scanning
Mirroring
Promotion
REST API
Classic API
Swarm API
Kube API
CA
AuthN
AuthZ
DCT
Metrics
UCP Controller
Router + Adapter
REST API
Classic API
Swarm API
Classic Swarm Mgr
Kube API
CNI
UCP Agent
etcd
Kubelet
Kube Mgr
swarm scope network plugin
3rd party plugins
REST API
Classic API
Swarm API
swarm-scheduler
secrets
inventory
raft
Docker EE Engine (managers)
Docker EE architecture
Kubernetes items shown in green
Highlights:
Exposes the full Kubernetes API
Backward compatible with EE Engine 17.06
Interop between pods, services, and containers
Docker daemon is the source of truth for resource usage for schedulers
Docker EE worker nodes
UCP Agent
Kubelet
REST API
Classic API
Swarm API
Docker EE Engine (workers)

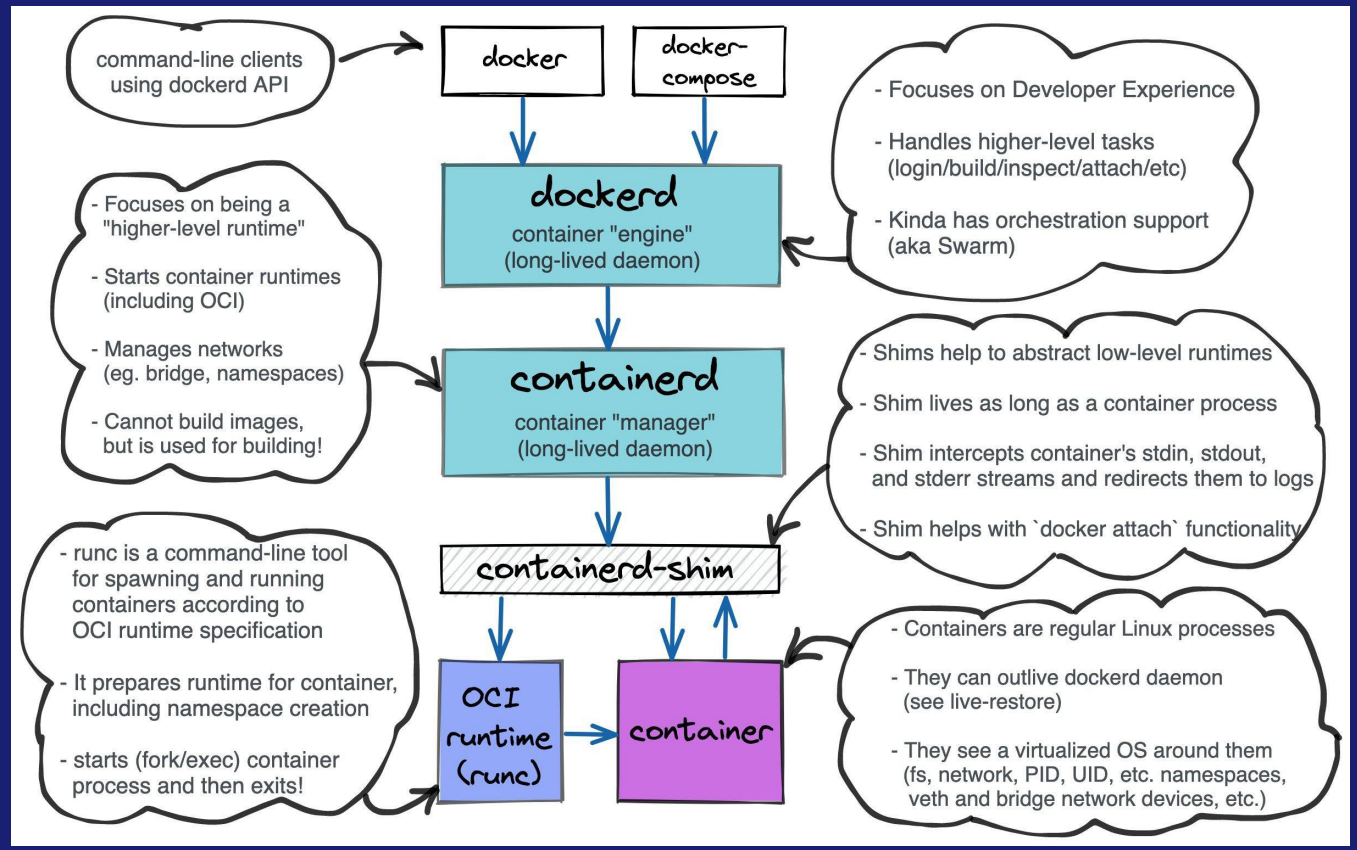
DOCKER CE/EE


EE Engine
Certified Plugins Infra ISVs
Support SLA
FIPS
CE Engine
Docker CLI
dockerd
Docker API
Storage Mgmt
libnetwork
Docker Build
SwarmKit
Docker Content Trust
Image Mgmt
containerd
runc
Plugins Storage Networking
Docker Compose

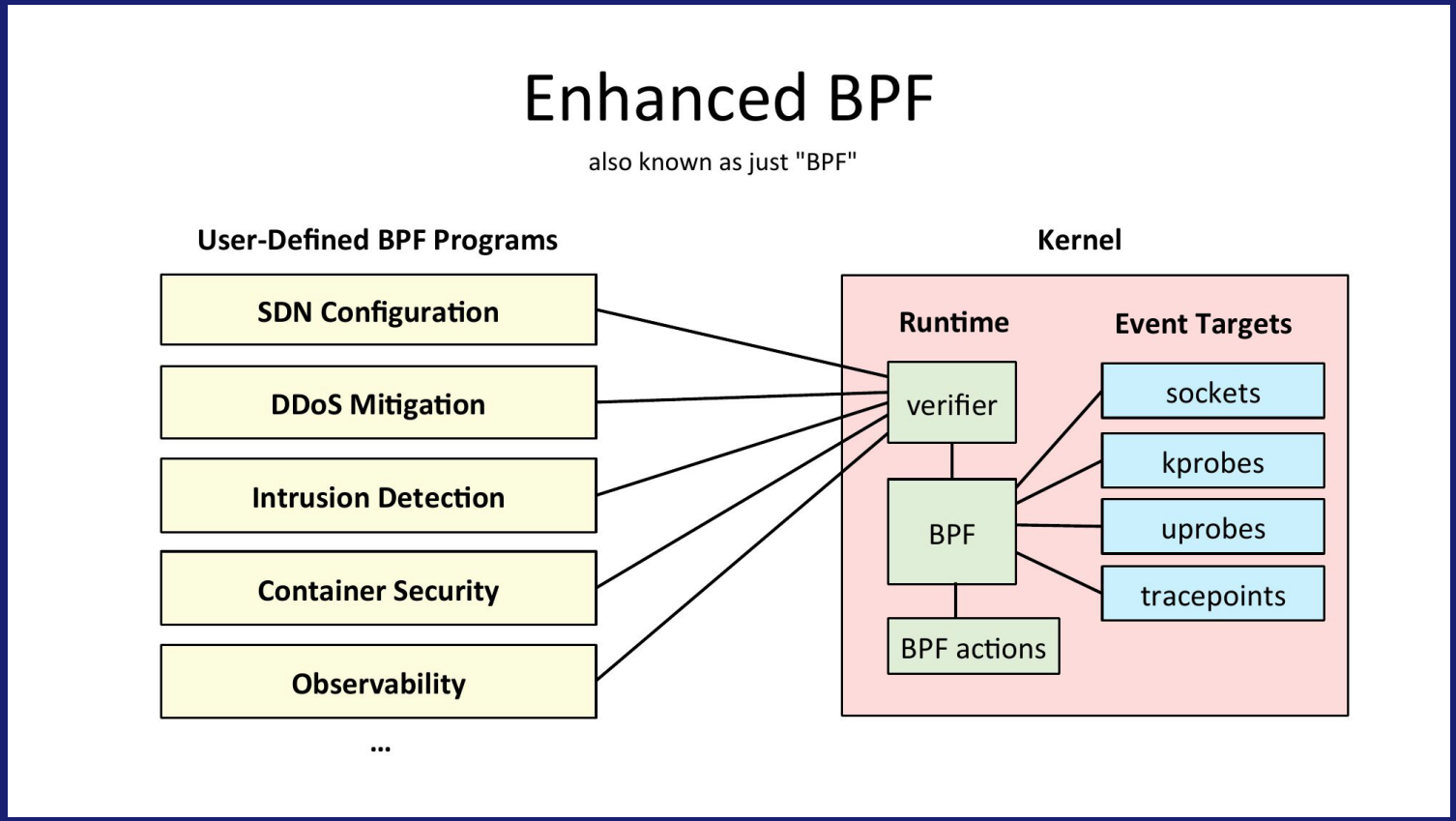
CONTAINERD(FROM DOCKER)



CONTAINERD(FROM DOCKER)



NETWORK BPF


Enhanced BPF
also known as just "BPF"
User-Defined BPF Programs
SDN Configuration
DDoS Mitigation
Intrusion Detection
Container Security
Observability
...
Kernel
Runtime
Event Targets
verifier
BPF
BPF actions
sockets
kprobes
uprobes
tracepoints

TRACING TIMELINE

A Linux Tracing Timeline

- 1990's: Static tracers, prototype dynamic tracers
- 2000: LTT + DProbes (dynamic tracing; not integrated)
- 2004: kprobes (2.6.9)
- 2005: DTrace (not Linux), SystemTap (out-of-tree)
- 2008: ftrace (2.6.27)
- 2009: perf (2.6.31)
- 2009: tracepoints (2.6.32)
- 2010-2016: ftrace & perf_events enhancements
- 2014-2016: BPF patches

also: LTTng, ktap, sysdig, ...

POD EMULATE

systemd

POD EMULATE #1

구성 조건

CREATE NAMESPACE

ip netns add redapp

ip netns add blueapp

ip netns

NAMESPACE WITH BRIDGE

```
# ip link set dev bridge-if1 up
# ip link
# ip link add veth-red1 type veth peer name veth-red1-br1
# ip link set veth-red1-br1 master bridge-if1
# ip link set veth-red1 netns redapp
# ip link set veth-blue1 netns blueapp
```

VIRTUAL DEVICE WITH NAMESPACE

```
# ip -n redapp addr add 192.168.1.11/24 dev veth-red1
# ip -n blueapp addr add 192.168.1.12/24 dev veth-blue1
# ip -n blueapp link set veth-blue1 up
# ip link set dev veth-blue1-br1 up
# ip link set dev veth-red1-br1 up
```

SET GATEWAY

```
# ip netns exec blueapp ip route add default via 192.168.1.10 dev veth-
blue1

# ip netns exec redapp ip route add default via 192.168.1.10 dev veth-red1
```

NAT NODE1

```
# iptables -t nat -A POSTROUTING -s 192.168.1.0/24 -d 10.130.0.10/32 -j
MASQUERADE
```

NODE1 TEST

```
# dnf install python3 -y
# ip netns exec redapp python3 -m http.server 8000
# ip netns exec redapp curl http://127.0.0.1:8000
# ip netns exec blueapp ping 192.168.1.11
# ip netns exec blueapp curl http://192.168.1.11:8000
```

NODE2 TEST

```
# ip netns exec blueapp ping 192.168.1.11
# ip netns exec blueapp curl http://192.168.1.11:8000
# ip netns exec redapp ping 192.168.1.11
# ip netns exec redapp curl http://192.168.1.11:8000
```

NODE1

```
# iptables -t nat -A PREROUTING -j DNAT -p tcp --dport 30001 -d 10.128.0.2
--to-destination 192.168.1.11:8000
```

쿠버네티스 호환

쿠버네티스 호환

2023-05-30