



Red Hat Enterprise Linux 8

컨테이너 실행 및 관리

Red Hat Enterprise Linux 8에서 Linux 컨테이너 구축, 실행 및 관리

Red Hat Enterprise Linux 8 컨테이너 실행 및 관리

Red Hat Enterprise Linux 8에서 Linux 컨테이너 구축, 실행 및 관리

Enter your first name here. Enter your surname here.

Enter your organisation's name here. Enter your organisational division here.

Enter your email address here.

법적 공지

Copyright © 2022 | You need to change the HOLDER entity in the en-US/Building_running_and_managing_containers.ent file |.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution-Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux[®] is the registered trademark of Linus Torvalds in the United States and other countries.

Java[®] is a registered trademark of Oracle and/or its affiliates.

XFS[®] is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL[®] is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js[®] is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack[®] Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

초록

이 문서에서는 podman, buildah, skopeo, runc 및 crun과 같은 명령줄 도구를 사용하여 Red Hat Enterprise Linux 8 시스템에서 Linux 컨테이너를 사용하는 방법을 설명합니다.

차례

보다 포괄적 수용을 위한 오픈 소스 용어 교체	6
RED HAT 문서에 관한 피드백 제공	7
1장. 컨테이너 시작	8
1.1. PODMAN, BUILDAH, SKOPEO의 특성	8
1.2. PODMAN 명령 개요	9
1.3. DOCKER 없이 컨테이너 실행	11
1.4. 컨테이너용 RHEL 아키텍처 선택	11
1.5. 컨테이너 툴 가져오기	11
1.6. ROOTLESS 컨테이너 설정	12
1.7. ROOTLESS 컨테이너로 업그레이드	13
1.8. ROOTLESS 컨테이너에 대한 특별 고려 사항	14
1.9. 추가 리소스	15
2장. 컨테이너 이미지 유형	16
2.1. RHEL 컨테이너 이미지의 일반 특성	16
2.2. UBI 이미지의 특성	16
2.3. UBI 표준 이미지 이해	17
2.4. UBI INIT 이미지 이해	17
2.5. UBI 최소 이미지 이해	18
2.6. UBI 마이크로 이미지 이해	19
2.7. UBI INIT 이미지 사용	19
2.8. UBI 마이크로 이미지 사용	21
3장. 컨테이너 이미지 작업	23
3.1. 컨테이너 레지스트리	23
3.2. 컨테이너 레지스트리 구성	24
3.3. 컨테이너 이미지 검색	26
3.4. 레지스트리에서 이미지 가져오기	27
3.5. 짧은 이름 별칭 구성	28
3.6. 짧은 이름 별칭을 사용하여 컨테이너 이미지 가져오기	29
3.7. 이미지 나열	30
3.8. 로컬 이미지 검사	31
3.9. 원격 이미지 검사	32
3.10. 컨테이너 이미지 복사	32
3.11. 로컬 디렉터리에 이미지 계층 복사	33
3.12. 이미지 태그 지정	34
3.13. 이미지 저장 및 로드	36
3.14. UBI 이미지 재배포	37
3.15. 이미지 서명의 기본 확인	38
3.16. 이미지 제거	39
4장. 컨테이너 작업	42
4.1. PODMAN RUN 명령	42
4.2. 호스트에서 컨테이너에서 명령 실행	43
4.3. 컨테이너 내에서 명령 실행	43
4.4. 컨테이너 나열	45
4.5. 컨테이너 시작	46
4.6. 호스트에서 컨테이너 검사	47
4.7. LOCALHOST의 디렉터리를 컨테이너에 마운트	49
4.8. 컨테이너 파일 시스템 마운트	49

4.9. 정적 IP를 사용하여 데몬으로 서비스 실행	51
4.10. 실행 중인 컨테이너 내에서 명령 실행	51
4.11. 두 컨테이너 간 파일 공유	53
4.12. 컨테이너 내보내기 및 가져오기	56
4.13. 컨테이너 중지	59
4.14. 컨테이너 제거	60
4.15. RUNC 컨테이너 런타임	61
4.16. CRUN 컨테이너 런타임	62
4.17. RUNC 및 CRUN을 사용하여 컨테이너 실행	62
4.18. 임시로 컨테이너 런타임 변경	64
4.19. 컨테이너 런타임 영구 변경	66
4.20. 컨테이너에 대한 SELINUX 정책 생성	67
5장. 노드 작업	68
5.1. POD 생성	68
5.2. POD 정보 표시	70
5.3. POD 중지	72
5.4. POD 제거	73
6장. 컨테이너 네트워크 관리	74
6.1. 컨테이너 네트워크 나열	74
6.2. 네트워크 검사	74
6.3. 네트워크 생성	76
6.4. 네트워크에 컨테이너 연결	77
6.5. 네트워크에서 컨테이너 연결 해제	77
6.6. 네트워크 제거	78
6.7. 사용되지 않는 모든 네트워크 제거	79
7장. 컨테이너 간 통신	81
7.1. 네트워크 모드 및 계층	81
7.2. 컨테이너의 네트워크 설정 검사	81
7.3. 컨테이너와 애플리케이션 간 통신	82
7.4. 컨테이너와 호스트 간의 통신	83
7.5. 포트 매핑을 사용하여 컨테이너 간 통신	84
7.6. DNS를 사용하여 컨테이너 간 통신	86
7.7. POD의 두 컨테이너 간 통신	87
7.8. POD에서 통신	88
7.9. 컨테이너 네트워크에 POD 연결	89
8장. 컨테이너 네트워크 모드 설정	91
8.1. 고정 IP로 컨테이너 실행	91
8.2. SYSTEMD 없이 DHCP 플러그인 실행	91
8.3. SYSTEMD를 사용하여 DHCP 플러그인 실행	93
8.4. MACVLAN 플러그인	94
8.5. 네트워크 스택을 CNI에서 NETAVARK로 전환	95
8.6. 네트워크 스택을 NETAVARK에서 CNI로 전환	96
9장. PODMAN을 사용하여 OPENSIFT로 컨테이너 이식	98
9.1. PODMAN을 사용하여 KUBERNETES YAML 파일 생성	98
9.2. OPENSIFT 환경에서 KUBERNETES YAML 파일 생성	100
9.3. PODMAN을 사용하여 컨테이너 및 포트 시작	101
9.4. OPENSIFT 환경에서 컨테이너 및 포트 시작	102
9.5. PODMAN을 사용하여 수동으로 컨테이너 및 POD 실행	102
9.6. PODMAN을 사용하여 YAML 파일 생성	105

9.7. PODMAN을 사용하여 컨테이너 및 POD 자동 실행	107
9.8. PODMAN을 사용하여 POD 자동 중지 및 제거	109
10장. PODMAN을 사용하여 컨테이너를 SYSTEMD로 포팅	111
10.1. SYSTEMD 서비스 활성화	111
10.2. PODMAN을 사용하여 SYSTEMD 장치 파일 생성	112
10.3. PODMAN을 사용하여 SYSTEMD 장치 파일 자동 생성	114
10.4. SYSTEMD를 사용하여 컨테이너 자동 시작	117
10.5. SYSTEMD를 사용하여 POD 자동 시작	118
10.6. PODMAN을 사용하여 컨테이너 자동 업데이트	122
10.7. SYSTEMD를 사용하여 컨테이너 자동 업데이트	124
11장. UBI 컨테이너에 소프트웨어 추가	127
11.1. 서브스크립션 호스트의 UBI 컨테이너에 소프트웨어 추가	127
11.2. 표준 UBI 컨테이너에 소프트웨어 추가	127
11.3. 최소 UBI 컨테이너에 소프트웨어 추가	129
11.4. 서브스크립션된 호스트의 UBI 컨테이너에 소프트웨어 추가	130
11.5. UBI 기반 이미지 빌드	131
11.6. APPLICATION STREAM 런타임 이미지 사용	133
11.7. UBI 컨테이너 이미지 소스 코드 가져오기	133
12장. 컨테이너에서 SKOPEO, BUILDAH, PODMAN 실행	136
12.1. 컨테이너에서 SKOPEO 실행	137
12.2. 자격 증명을 사용하여 컨테이너에서 SKOPEO 실행	138
12.3. AUTHFILES를 사용하여 컨테이너에서 SKOPEO 실행	139
12.4. 호스트에서 또는 호스트로 컨테이너 이미지 복사	140
12.5. 컨테이너에서 BUILDAH 실행	141
12.6. 권한 있는 PODMAN 컨테이너 및 권한이 없는 PODMAN 컨테이너	143
12.7. 확장된 권한으로 PODMAN 실행	143
12.8. 더 적은 권한으로 PODMAN 실행	145
12.9. PODMAN 컨테이너 내부에 컨테이너 빌드	146
13장. BUILDAH를 사용하여 컨테이너 이미지 빌드	149
13.1. BUILDAH 툴	149
13.2. BUILDAH 설치	150
13.3. BUILDAH로 이미지 가져오기	151
13.4. 컨테이너 내에서 명령 실행	152
13.5. BUILDAH를 사용하여 CONTAINERFILE에서 이미지 빌드	152
13.6. BUILDAH를 사용하여 컨테이너 및 이미지 검사	154
13.7. BUILDAH 마운트를 사용하여 컨테이너 수정	155
13.8. BUILDAH 복사 및 BUILDAH 구성을 사용하여 컨테이너 수정	157
13.9. BUILDAH를 사용하여 처음부터 이미지 생성	158
13.10. 프라이빗 레지스트리로 컨테이너 내보내기	160
13.11. DOCKER HUB로 컨테이너 푸시	162
13.12. BUILDAH로 이미지 제거	163
13.13. BUILDAH를 사용하여 컨테이너 제거	164
14장. 컨테이너 모니터링	166
14.1. 컨테이너에서 상태 점검 수행	166
14.2. PODMAN 시스템 정보 표시	168
14.3. PODMAN 이벤트 유형	172
14.4. PODMAN 이벤트 모니터링	175
15장. 컨테이너 체크포인트 생성 및 복원	177
15.1. 로컬에서 컨테이너 체크포인트 생성 및 복원	177

15.2. 컨테이너 복원을 사용하여 시작 시간 단축	179
15.3. 시스템 간 컨테이너 마이그레이션	181
16장. HPC 환경에서 PODMAN 사용	183
16.1. MPI로 PODMAN 사용	183
16.2. MPIRUN 옵션	184
17장. 특수 컨테이너 이미지 실행	186
17.1. 호스트에 대한 권한 열기	186
17.2. RUNLABELS가 있는 컨테이너 이미지	186
17.3. RUNLABELS로 RSYSLOG 실행	187
18장. CONTAINER-TOOLS API 사용	190
18.1. ROOT 모드에서 SYSTEMD를 사용하여 PODMAN API 활성화	190
18.2. ROOTLESS 모드에서 SYSTEMD를 사용하여 PODMAN API 활성화	191
18.3. PODMAN API 수동 실행	192

보다 포괄적 수용을 위한 오픈 소스 용어 교체

Red Hat은 코드, 문서, 웹 속성에서 문제가 있는 용어를 교체하기 위해 최선을 다하고 있습니다. 먼저 마스터(master), 슬레이브(slave), 블랙리스트(blacklist), 화이트리스트(whitelist) 등 네 가지 용어를 교체하고 있습니다. 이러한 변경 작업은 작업 범위가 크므로 향후 여러 릴리스에 걸쳐 점차 구현할 예정입니다. 자세한 내용은 [CTO Chris Wright의 메시지](#)를 참조하십시오.

RED HAT 문서에 관한 피드백 제공

문서 개선을 위한 의견을 보내 주십시오. 문서를 개선하는 방법을 알려주십시오.

- 특정 문구에 대한 간단한 의견 작성 방법은 다음과 같습니다.
 1. 문서가 *Multi-page HTML* 형식으로 표시되는지 확인합니다. 또한 문서 오른쪽 상단에 **피드백** 버튼이 있는지 확인합니다.
 2. 마우스 커서를 사용하여 주석 처리하려는 텍스트 부분을 강조 표시합니다.
 3. 강조 표시된 텍스트 아래에 표시되는 **피드백 추가** 팝업을 클릭합니다.
 4. 표시된 지침을 따릅니다.
- Bugzilla를 통해 피드백을 제출하려면 새 티켓을 생성합니다.
 1. [Bugzilla](#) 웹 사이트로 이동하십시오.
 2. Component로 **Documentation**을 선택하십시오.
 3. **Description** 필드에 문서 개선을 위한 제안 사항을 기입하십시오. 관련된 문서의 해당 부분 링크를 알려주십시오.
 4. **Submit Bug**를 클릭하십시오.

1장. 컨테이너 시작

Linux 컨테이너는 경량 애플리케이션 격리와 이미지 기반 배포 방법의 유연성을 결합한 핵심 오픈 소스 애플리케이션 패키징 및 제공 기술로 부각되었습니다. RHEL은 다음과 같은 핵심 기술을 사용하여 Linux 컨테이너를 구현합니다.

- 리소스 관리를 위한 컨트롤 그룹(cgroup)
- 프로세스 격리를 위한 네임스페이스
- 보안을 위한 SELinux
- 보안 멀티 테넌시

이러한 기술은 보안 위협의 가능성을 줄이고 엔터프라이즈급 컨테이너를 생성하고 실행하는 환경을 제공합니다.

Red Hat OpenShift는 포드라는 단위로 컨테이너를 빌드, 관리 및 실행하기 위한 강력한 명령줄 및 웹 UI 툴을 제공합니다. Red Hat을 사용하면 OpenShift 외부에서 개별 컨테이너 및 컨테이너 이미지를 빌드하고 관리할 수 있습니다. 이 가이드에서는 RHEL 시스템에서 직접 실행되는 작업을 수행하기 위해 제공되는 툴을 설명합니다.

다른 컨테이너 툴 구현과 달리 여기에 설명된 툴은 모놀리식 Docker 컨테이너 엔진 및 **docker** 명령을 중심으로 진행하지 않습니다. 대신 Red Hat은 컨테이너 엔진 없이 작동할 수 있는 명령줄 도구 집합을 제공합니다. 여기에는 다음이 포함됩니다.

- **Podman** - 포드 및 컨테이너 이미지를 직접 관리하는 경우 (**run,stop,start,ps,attach,exec** 등)
- **Buildah** - 컨테이너 이미지 빌드, 내보내기 및 서명
- **skopeo** - 이미지 복사, 검사, 삭제 및 서명
- **runc** - podman 및 buildah에 컨테이너 실행 및 빌드 기능을 제공하기 위한
- **crun** - rootless 컨테이너에 대해 보다 높은 유연성, 제어 및 보안을 제공하고 구성할 수 있는 선택적 런타임

이러한 툴은 OCI(Open Container Initiative)와 호환되므로 Docker 및 기타 OCI 호환 컨테이너 엔진에서 생성 및 관리하는 것과 동일한 Linux 컨테이너를 관리하는 데 사용할 수 있습니다. 그러나 이러한 기능은 단일 노드 사용 사례에서 Red Hat Enterprise Linux에서 직접 실행하는 데 특히 적합합니다.

멀티 노드 컨테이너 플랫폼은 [OpenShift](#) 및 [CRI-O 컨테이너 엔진](#) 사용을 참조하십시오.

1.1. PODMAN, BUILDHAH, SKOPEO의 특성

Docker 명령 기능을 대체하기 위해 Podman, Skopeo 및 Buildah 툴이 개발되었습니다. 이 시나리오의 각 툴은 더 가벼우며 기능 하위 집합에 중점을 둡니다.

Podman, Skopeo 및 Buildah 툴의 주요 장점은 다음과 같습니다.

- **rootless 모드에서 실행** - rootless 컨테이너는 추가 권한 없이 실행되므로 훨씬 더 안전합니다.
- **필요한 데몬 없음** - 이러한 툴에는 컨테이너를 실행하지 않는 경우 Podman이 실행되지 않기 때문에 유휴 상태에서 리소스 요구 사항이 훨씬 낮아집니다. 반면 Docker에는 항상 데몬이 실행 중입니다.

- 기본 systemd 통합 - Podman을 사용하면 systemd 유닛 파일을 생성하고 시스템 서비스로 컨테이너를 실행할 수 있습니다.

Podman, Skopeo 및 Buildah의 특성은 다음과 같습니다.

- Podman, Buildah 및 CRI-O 컨테이너 엔진은 기본적으로 Docker 스토리지 위치 **/var/lib/ docker**를 사용하는 대신 동일한 백엔드 저장소 디렉터리인 **/var/lib/ containers**를 사용합니다.
- Podman, Buildah 및 CRI-O는 동일한 스토리지 디렉터리를 공유하지만 서로의 컨테이너와 상호 작용할 수 없습니다. 이러한 도구는 이미지를 공유할 수 있습니다.
- Podman과 프로그래밍 방식으로 상호 작용하려면 Podman v2.0 RESTful API를 사용하여 rootful 및 rootless 환경 모두에서 작동합니다. 자세한 내용은 [container-tools API 사용 장](#)을 참조하십시오.

추가 리소스

- [Buildah, Podman, Skopeo에 대해 "Hello"를 입력합니다.](#)
- [Docker 사용자를 Podman 및 Buildah](#)
- [Buildah - OCI 컨테이너 이미지를 빌드하는 툴](#)
- [Podman - 컨테이너 실행 및 관리를 위한 툴](#)
- [Skopeo - 컨테이너 이미지를 복사하고 검사하는 툴](#)

1.2. PODMAN 명령 개요

표 1.1은 **podman** 명령과 함께 사용할 수 있는 명령 목록을 보여줍니다. **podman -h**를 사용하여 모든 Podman 명령 목록을 확인합니다.

표 1.1. podman에서 지원하는 명령

Podman 명령	설명	Podman 명령	설명
attach	실행 중인 컨테이너에 연결	커밋	변경된 컨테이너에서 새 이미지 만들기
build	Containerfile 지침을 사용하여 이미지 빌드	create	컨테이너를 만들지만 시작하지 않습니다.
diff	컨테이너 파일 시스템에서 변경 사항 검사	exec	실행 중인 컨테이너에서 프로세스 실행
export	컨테이너의 파일 시스템 콘텐츠를 tar 아카이브로 내보냅니다.	도움말, h	명령 목록 또는 한 명령에 대한 도움말 표시
내역	지정된 이미지의 기록 표시	이미지	로컬 스토리지의 이미지 나열

import	tarball을 가져와 파일 시스템 이미지 생성	info	시스템 정보 표시
inspect	컨테이너 또는 이미지의 구성 표시	강제 종료	실행 중인 하나 이상의 컨테이너에 특정 신호 전송
load	아카이브에서 이미지 로드	login	컨테이너 레지스트리에 로그인
logout	컨테이너 레지스트리에서 로그아웃	logs	컨테이너의 로그를 가져옵니다
Mount	작동 중인 컨테이너의 루트 파일 시스템 마운트	pause	하나 이상의 컨테이너에서 모든 프로세스를 일시 중지
ps	컨테이너 나열	port	포트 매핑 또는 컨테이너에 대한 특정 매핑 나열
pull	레지스트리에서 이미지 가져오기	push	지정된 대상으로 이미지 푸시
재시작	하나 이상의 컨테이너를 다시 시작	rm	호스트에서 하나 이상의 컨테이너를 제거합니다. 실행 중인 경우 -f 를 추가합니다.
rmi	로컬 스토리지에서 하나 이상의 이미지 제거	run	새 컨테이너에서 명령 실행
save	아카이브에 이미지 저장	search	이미지 레지스트리 검색
start	하나 이상의 컨테이너 시작	통계	하나 이상의 컨테이너의 CPU, 메모리, 네트워크 I/O, 블록 I/O 및 PID 표시
중지	하나 이상의 컨테이너 중지	tag	로컬 이미지에 추가 이름 추가
top	컨테이너의 실행 중인 프로세스 표시	umount, 마운트 해제	작동 중인 컨테이너의 루트 파일 시스템 마운트 해제
일시 중지 해제	하나 이상의 컨테이너에서 프로세스 일시 중지 해제	버전	podman 버전 정보 표시
wait	하나 이상의 컨테이너에서 차단		

추가 리소스

- [podman Basics Cheatsheet](#)
- [지금 사용해 볼 수 있는 5개의 podman 기능](#)

1.3. DOCKER 없이 컨테이너 실행

Red Hat은 RHEL 8에서 Docker 컨테이너 엔진 및 docker 명령을 제거했습니다.

RHEL에서 Docker를 계속 사용하려면 여러 업스트림 프로젝트에서 Docker를 가져올 수 있지만 RHEL 8에서는 지원되지 않습니다.

- **docker** 명령을 실행할 때마다 **podman-docker** 패키지를 설치할 수 있습니다. 실제로는 **podman** 명령을 실행합니다.
- Podman은 Docker 소켓 API도 지원하므로 **podman-docker** 패키지에서는 `/var/run/docker.sock` 및 `/var/run/podman/podman.sock` 간 링크도 설정합니다. 결과적으로 Docker 데몬 없이 **docker-py** 및 **docker-compose** 툴을 사용하여 Docker API 명령을 계속 실행할 수 있습니다. Podman은 요청을 서비스합니다.
- **podman** 명령은 **docker** 명령과 같이 **Containerfile** 또는 **Dockerfile**에서 컨테이너 이미지를 빌드할 수 있습니다. **Containerfile** 및 **Dockerfile** 내에서 사용할 수 있는 사용 가능한 명령은 동일합니다.
- **podman**에서 지원하지 않는 **docker** 명령에 대한 옵션은 네트워크, 노드, 플러그인(**podman**은 플러그인을 지원하지 않음), 이름 변경(**podman**을 사용하여 컨테이너의 이름을 변경하기 위해 `rm` 사용), 시크릿, 서비스, 스택, 집단(**podman Swarm**을 지원하지 않음). 컨테이너 및 이미지 옵션은 **podman**에서 직접 사용되는 하위 명령을 실행하는 데 사용됩니다.

추가 리소스

- [Docker 사용자용 Podman 및 Buildah](#)

1.4. 컨테이너용 RHEL 아키텍처 선택

Red Hat은 다음 컴퓨터 아키텍처에 대한 컨테이너 이미지 및 컨테이너 관련 소프트웨어를 제공합니다.

- AMD64 및 Intel 64(기본 및 계층화된 이미지, 32비트 아키텍처는 지원하지 않음)
- PowerPC 8 및 9 64비트 (기본 이미지 및 대부분의 계층화된 이미지)
- 64비트 IBM Z(기본 이미지 및 대부분의 계층화된 이미지)
- ARM 64비트 (기본 이미지만 해당)

처음에는 모든 아키텍처에서 모든 Red Hat 이미지가 지원되지는 않지만, 나열된 모든 아키텍처에서 거의 모든 Red Hat 이미지를 사용할 수 있습니다.

추가 리소스

- [UBI\(Universal Base Image\): 이미지, 리포지토리 및 패키지](#)

1.5. 컨테이너 툴 가져오기

다음 절차에서는 Podman, Buildah, Skopeo, CRIU, Udrca 및 모든 필수 라이브러리를 포함하는 **container-tools** 모듈을 설치하는 방법을 보여줍니다.

절차

1. RHEL; 설치.
2. RHEL 등록: 사용자 이름과 암호를 입력합니다. 사용자 이름과 암호는 Red Hat 고객 포털의 로그인 자격 증명과 동일합니다.

```
# subscription-manager register
Registering to: subscription.rhsm.redhat.com:443/subscription
Username: *****
Password: *****
```

3. RHEL;에 가입.
 - RHEL에 자동 등록하려면 다음을 수행합니다.

```
# subscription-manager attach --auto
```

- RHEL by Pool ID에 가입하려면 다음을 수행합니다.

```
# subscription-manager attach --pool PoolID
```

4. **container-tools** 모듈을 설치합니다.

```
# yum module install -y container-tools
```

5. 선택 사항: **podman-docker** 패키지를 설치합니다.

```
# yum install -y podman-docker
```

podman-docker 패키지에서는 Docker 명령줄 인터페이스와 **docker-api** 를 일치하는 Podman 명령으로 교체합니다.

1.6. ROOTLESS 컨테이너 설정

Podman, Skopeo 또는 Buildah와 같은 컨테이너 툴을 슈퍼유저 권한(root 사용자)이 있는 사용자로 실행하는 것이 컨테이너에서 시스템에서 사용 가능한 모든 기능에 대한 전체 액세스 권한을 갖는 가장 좋은 방법입니다. 그러나 일반적으로 RHEL 8.1부터 "Rootless Containers"라는 기능을 사용하면 일반 사용자로 컨테이너를 사용할 수 있습니다.

Docker와 같은 컨테이너 엔진을 사용하면 Docker 명령을 일반(루트가 아닌) 사용자로 실행할 수 있지만 해당 요청을 수행하는 Docker 데몬은 root로 실행됩니다. 결과적으로 일반 사용자는 시스템을 손상시킬 수 있는 컨테이너를 통해 요청할 수 있습니다. rootless 컨테이너 사용자를 설정하여 시스템 관리자는 일반 사용자에게 잠재적으로 컨테이너 활동을 손상시키는 것을 방지하는 한편, 해당 사용자는 자신의 계정으로 대부분의 컨테이너 기능을 안전하게 실행할 수 있습니다.

이 절차에서는 루트가 아닌 사용자(rootless)로 컨테이너를 사용하기 위해 Podman, Skopeo 및 Buildah 툴을 사용하도록 시스템을 설정하는 방법을 설명합니다. 일반 사용자 계정에 컨테이너를 실행해야 하는 모든 운영 체제 기능에 대한 전체 액세스 권한이 없으므로 발생하는 몇 가지 제한 사항에 대해서도 설명합니다.

사전 요구 사항

- 루트가 아닌 사용자 계정이 컨테이너 툴을 사용할 수 있도록 RHEL 시스템을 설정하려면 root 사용자가 되어야 합니다.

절차

1. RHEL; 설치.
2. **podman** 패키지를 설치합니다.

```
# yum install podman -y
```

3. 새 사용자 계정을 생성합니다.

```
# useradd -c "Joe Jones" joe
# passwd joe
```

- 사용자는 rootless Podman을 사용하도록 자동으로 구성됩니다.
- **useradd** 명령은 **/etc/subuid** 및 **/etc/subgid** 파일에서 액세스 가능한 사용자 및 그룹 ID의 범위를 자동으로 설정합니다.
- **/etc/subuid** 또는 **/etc/subgid** 를 수동으로 변경하는 경우 **podman system migrate** 명령을 실행하여 새 변경 사항을 적용해야 합니다.

4. 사용자에게 연결합니다.

```
$ ssh joe@server.example.com
```



참고

이러한 명령에서는 올바른 환경 변수를 설정하지 않으므로 **su** 또는 **su -** 명령을 사용하지 마십시오.

5. **registry.access.redhat.com/ubi8/ubi** 컨테이너 이미지를 가져옵니다.

```
$ podman pull registry.access.redhat.com/ubi8/ubi
```

6. **myubi** 라는 컨테이너를 실행하고 OS 버전을 표시합니다.

```
$ podman run --rm --name=myubi registry.access.redhat.com/ubi8/ubi cat \
/etc/os-release
NAME="Red Hat Enterprise Linux"
VERSION="8 (Plow)"
```

추가 리소스

- [Podman이 있는 rootless 컨테이너: 기본 사항](#)
- [podman-system-migrate](#) 매뉴얼 페이지

1.7. ROOTLESS 컨테이너로 업그레이드

이 섹션에서는 RHEL 7에서 rootless 컨테이너로 업그레이드하는 방법을 보여줍니다. 사용자 및 그룹 ID를 수동으로 구성해야 합니다.

다음은 RHEL 7에서 rootless 컨테이너로 업그레이드할 때 고려해야 할 사항은 다음과 같습니다.

- 여러 rootless 컨테이너 사용자를 설정하는 경우 각 사용자에게 대해 고유한 범위를 사용합니다.
- 기존 컨테이너 이미지와의 최대 호환성을 위해 65536 UID 및 GID를 사용하지만 번호를 줄일 수 있습니다.
- 1000 미만의 UID 또는 GID를 사용하지 않거나 기존 사용자 계정(기본적으로 1000부터 시작)에서 UID 또는 GID를 재사용하지 마십시오.

사전 요구 사항

- 사용자 계정이 생성되었습니다.

절차

- **usermod** 명령을 실행하여 UID 및 GID를 사용자에게 할당합니다.

```
# usermod --add-subuids 200000-201000 --add-subgids 200000-201000 username
```

- **usermod --add-subuid** 명령은 액세스 가능한 사용자 ID 범위를 사용자 계정에 수동으로 추가합니다.
- **usermod --add-subgids** 명령은 액세스 가능한 사용자 GID 및 그룹 ID 범위를 사용자 계정에 수동으로 추가합니다.

검증 단계

- UID 및 GID가 올바르게 설정되었는지 확인합니다.

```
# grep username /etc/subuid /etc/subgid
#/etc/subuid:username:200000:1001
#/etc/subgid:username:200000:1001
```

1.8. ROOTLESS 컨테이너에 대한 특별 고려 사항

루트가 아닌 사용자로 컨테이너를 실행할 때 고려해야 할 몇 가지 사항이 있습니다.

- 호스트 컨테이너 스토리지의 경로는 root 사용자(**/var/lib/containers/storage**) 및 루트가 아닌 사용자(**\$HOME/.local/share/containers/storage**)에 따라 다릅니다.
- rootless 컨테이너를 실행하는 사용자에게는 호스트 시스템에서 다양한 사용자 및 그룹 ID로 실행할 수 있는 특별 권한이 부여됩니다. 그러나 호스트에서 운영 체제에 대한 루트 권한이 없습니다.
- **/etc/subuid** 또는 **/etc/subgid** 를 수동으로 변경하는 경우 **podman system migrate** 명령을 실행하여 새 변경 사항을 적용해야 합니다.
- rootless 컨테이너 환경을 구성해야 하는 경우 홈 디렉터리(**\$HOME/.config/containers**)에 구성 파일을 생성합니다. 구성 파일에는 **storage.conf** (스토리지 구성용) 및 **containers.conf** (다양한 컨테이너 설정의 경우)가 포함됩니다. **registries.conf** 파일을 생성하여 Podman을 사용하여 이미지를 가져오거나 검색하거나 실행할 때 사용할 수 있는 컨테이너 레지스트리를 식별할 수도 있습니다.

- 루트 권한 없이 변경할 수 없는 일부 시스템 기능이 있습니다. 예를 들어 컨테이너 내부에 **SYS_TIME** 기능을 설정하고 **ntpd**(네트워크 시간 서비스)를 실행하여 시스템 시계를 변경할 수 없습니다. **rootless** 컨테이너 환경을 우회하고 **root** 사용자의 환경을 사용하여 해당 컨테이너를 **root**로 실행해야 합니다. 예를 들면 다음과 같습니다.

```
$ sudo podman run -d --cap-add SYS_TIME ntpd
```

이 예에서는 **ntpd**가 컨테이너 내부뿐만 아니라 전체 시스템의 시간을 조정할 수 있습니다.

- rootless** 컨테이너는 1024 미만의 포트에 액세스할 수 없습니다. 예를 들어, **rootless** 컨테이너 네임스페이스 내에서는 컨테이너에서 **httpd** 서비스에서 포트 80을 노출하는 서비스를 시작하지만 네임스페이스 외부에서는 액세스할 수 없습니다.

```
$ podman run -d httpd
```

그러나 해당 포트를 호스트 시스템에 노출하려면 컨테이너에 **root** 사용자의 컨테이너 환경을 사용하는 **root** 권한이 필요합니다.

```
$ sudo podman run -d -p 80:80 httpd
```

- workstation** 관리자는 사용자가 1024 미만의 포트에 서비스를 노출할 수 있지만 보안상의 영향을 이해해야 합니다. 예를 들어 일반 사용자는 공식 포트 80에서 웹 서버를 실행하고 외부 사용자가 관리자가 구성했다고 판단할 수 있습니다. 이 작업은 워크스테이션에서 테스트할 수 있지만, 네트워크로 액세스할 수 있는 개발 서버에 적합하지 않을 수 있으며, 프로덕션 서버에서는 이 작업을 수행하면 안 됩니다. 사용자가 포트 80에 대한 포트에 바인딩할 수 있도록 하려면 다음 명령을 실행합니다.

```
# echo 80 > /proc/sys/net/ipv4/ip_unprivileged_port_start
```

추가 리소스

- [Rootless Podman의 단점](#)

1.9. 추가 리소스

- [실용적인 컨테이너 용어 소개](#)

2장. 컨테이너 이미지 유형

컨테이너 이미지는 단일 컨테이너를 실행하기 위한 모든 요구 사항과 요구 사항 및 기능을 설명하는 메타 데이터를 포함하는 바이너리입니다.

컨테이너 이미지는 다음 두 가지 유형이 있습니다.

- Red Hat Enterprise Linux 기본 이미지(RHEL 기본 이미지)
- Red Hat Universal Base Images(UBI 이미지)

두 유형의 컨테이너 이미지는 Red Hat Enterprise Linux의 일부에서 빌드됩니다. 이러한 컨테이너를 사용하면 뛰어난 신뢰성, 보안, 성능 및 라이프사이클의 이점을 누릴 수 있습니다.

두 가지 유형의 컨테이너 이미지의 주요 차이점은 UBI 이미지를 다른 사용자와 컨테이너 이미지를 공유할 수 있다는 것입니다. UBI를 사용하여 컨테이너화된 애플리케이션을 빌드하고 선택한 레지스트리 서버에 푸시하고 다른 사용자와 공유하며 Red Hat이 아닌 플랫폼에 배포할 수도 있습니다. UBI 이미지는 컨테이너에서 개발된 클라우드 네이티브 및 웹 애플리케이션 사용 사례의 기반이 되도록 설계되었습니다.

2.1. RHEL 컨테이너 이미지의 일반 특성

다음 기능은 RHEL 기본 이미지와 UBI 이미지에 모두 적용됩니다.

일반적으로 RHEL 컨테이너 이미지는 다음과 같습니다.

- **지원됨:** Red Hat에서 컨테이너화된 애플리케이션과 함께 사용할 수 있도록 지원합니다. 여기에는 Red Hat Enterprise Linux에 있는 것과 동일한 보안, 테스트 및 인증된 소프트웨어 패키지가 포함되어 있습니다.
- **카탈로그:** [Red Hat Container Catalog](#)에 나열된 설명, 기술 세부 정보 및 각 이미지의 상태 색인입니다.
- **업데이트됨:** 최신 소프트웨어를 얻으려면 잘 정의된 업데이트 일정이 포함된 [Red Hat Container Image Updates](#) 문서를 참조하십시오.
- **추적 됨:** Red Hat 제품 Errata가 추적하여 각 업데이트에 추가된 변경 사항을 이해하는 데 도움이 됩니다.
- **재사용 가능:** 컨테이너 이미지를 한 번 다운로드하여 프로덕션 환경에 캐시해야 합니다. 각 컨테이너 이미지는 기본으로 포함된 모든 컨테이너에서 재사용할 수 있습니다.

2.2. UBI 이미지의 특성

UBI 이미지를 사용하면 컨테이너 이미지를 다른 사용자와 공유할 수 있습니다. 마이크로, 최소, 표준 및 init의 네 가지 UBI 이미지가 제공됩니다. 사전 빌드 언어 런타임 이미지 및 YUM 리포지토리를 사용하여 애플리케이션을 빌드할 수 있습니다.

UBI 이미지에 다음과 같은 특징이 적용됩니다.

- **RHEL 콘텐츠의 하위 집합에서 빌드:** Red Hat Universal Base 이미지는 일반적인 Red Hat Enterprise Linux 콘텐츠의 하위 집합에서 빌드됩니다.
- **배포 가능:** UBI 이미지를 사용하면 Red Hat 고객, 파트너, ISV 등이 표준화할 수 있습니다. UBI 이미지를 사용하면 자유롭게 공유하고 배포할 수 있는 공식 Red Hat 소프트웨어를 기반으로 컨테이너 이미지를 빌드할 수 있습니다.

- 마이크로, 최소, 표준, init의 네 가지 기본 이미지 세트를 제공합니다.
- 사전 빌드된 언어 런타임 컨테이너 이미지 세트를 제공합니다. 애플리케이션 스트림 을 기반으로 하는 런타임 이미지는 python, perl, php, dotnet, nodejs, ruby 등 지원되는 표준 런타임을 활용할 수 있는 애플리케이션의 토대를 제공합니다.
- 관련 YUM 리포지토리 세트를 제공합니다. YUM 리포지토리에는 애플리케이션 종속성을 추가하고 UBI 컨테이너 이미지를 다시 빌드할 수 있는 RPM 패키지 및 업데이트가 포함되어 있습니다.
 - **ubi-8-baseos** 리포지토리는 컨테이너에 포함할 수 있는 RHEL 패키지의 하위 집합을 보유합니다.
 - **ubi-8-appstream** 리포지토리에는 UBI 이미지에 추가할 수 있는 애플리케이션 스트림 패키지가 포함되어 있어 특정 런타임이 필요한 애플리케이션에서 사용하는 환경을 표준화할 수 있습니다.
 - **UBI RPM 추가:** 사전 구성된 UBI 리포지토리의 UBI 이미지에 RPM 패키지를 추가할 수 있습니다. 연결이 끊긴 환경에 있는 경우 해당 기능을 사용하려면 UBI Content Delivery Network(<https://cdn-ubi.redhat.com>) 목록을 허용해야 합니다. 자세한 내용은 <https://cdn-ubi.redhat.com>에 연결 솔루션을 참조하십시오.
- 라이선스: [Red Hat Universal Base Image End User Licensing Agreement](#) 를 준수하면 UBI 이미지를 자유롭게 사용하고 재배포할 수 있습니다.

추가 리소스

- [Red Hat Universal Base Image 소개](#)
- [UBI\(Universal Base Image\): 이미지, 리포지토리 및 패키지](#)
- [Red Hat Universal Base Image에 대해 알아야 할 모든 것](#)
- [FAQ - 범용 기본 이미지](#)

2.3. UBI 표준 이미지 이해

표준 이미지(이름이 **ubi**)는 RHEL에서 실행되는 모든 애플리케이션에 대해 설계되었습니다. UBI 표준 이미지의 주요 기능은 다음과 같습니다.

- **init 시스템:** systemd 서비스를 관리하는 데 필요한 systemd 초기화 시스템의 모든 기능을 표준 기본 이미지에서 사용할 수 있습니다. 이러한 init 시스템을 사용하면 웹 서버(httpd) 또는 **FTP 서버(vsftpd)**와 같이 서비스를 자동으로 시작하도록 미리 구성된 RPM 패키지를 설치할 수 있습니다.
- **yum:** 소프트웨어 추가 및 업데이트를 위한 무료 yum 리포지토리에 액세스할 수 있습니다. 표준 **yum 명령 집합(yum, yum -config-manager, yum downloader 등)**을 사용할 수 있습니다.
- **유틸리티:** 유틸리티에는 **tar, dmidecode, gzip, getfacl** 및 추가 **acl 명령, dmsetup** 및 여기에 언급되지 않은 다른 유틸리티 간에 추가 장치 매핑 명령이 포함됩니다.

2.4. UBI INIT 이미지 이해

ubi-init 라는 UBI init 이미지에는 systemd 초기화 시스템이 포함되어 있어 웹 서버 또는 파일 서버와 같은 systemd 서비스를 실행하려는 이미지를 빌드하는 데 유용합니다. init 이미지 콘텐츠는 표준 이미지로 가져오는 것보다 작지만, 최소 이미지에 있는 것보다 더 중요합니다.

참고

ubi8-init 이미지는 **ubi8** 이미지 위에 빌드되므로 해당 콘텐츠는 대부분 동일합니다. 그러나 몇 가지 중요한 차이점이 있습니다.

- **ubi8-init:**
 - 기본적으로 systemd Init 서비스를 시작하도록 CMD가 **/sbin/init** 로 설정됩니다.
 - **ps** 및 프로세스 관련 명령 포함 (**procps-ng** package)
 - **ubi8-init** 의 systemd에서 종료할 일반 신호를 무시하므로 **SIGRTMIN+3** 을 **StopSignal** 로 설정 하지만 **SIGRTMIN+3**을 수신하면 종료됩니다.
- **ubi8:**
 - CMD가 **/bin/bash**로 설정됨
 - **ps** 및 프로세스 관련 명령을 포함하지 않습니다 (**procps-ng** 패키지)
 - 종료되는 일반 신호를 무시하지 않습니다 (**SIGTERM** 및 **SIGKILL**)

2.5. UBI 최소 이미지 이해

ubi-minimal 이라는 **UBI** 최소 이미지는 사전 설치된 최소화된 콘텐츠 세트와 패키지 관리자 (**microdnf**)를 제공합니다. 따라서 이미지에 포함된 종속성을 최소화하면서 **Containerfile** 을 사용할 수 있습니다.

UBI 최소 이미지의 주요 기능은 다음과 같습니다.

- 작은 크기: 최소 이미지는 압축 시 디스크에서 약 **92M**과 **32M**입니다. 이렇게 하면 표준 이미지의 크기가 절반 미만입니다.
- 소프트웨어 설치 (**microdnf**): 소프트웨어 리포지토리 및 **RPM** 소프트웨어 패키지로 작업하기 위해 완전히 개발된 **yum** 기능을 포함하는 대신, 최소 이미지에는 **microdnf** 유틸리티가 포함됩니다. **microdnf** 는 리포지토리를 활성화 및 비활성화하고 패키지를 제거 및 업데이트하며 패키지를 설치한 후 캐시를 정리할 수 있는 스케일 다운 버전의 **dnf** 입니다.
-

RHEL 패키징 기반: 최소한의 이미지는 일반 **RHEL** 소프트웨어 **RPM** 패키지를 포함하며, 몇 가지 기능이 제거되었습니다. 최소한의 이미지에는 초기화 및 서비스 관리 시스템(예: **systemd** 또는 **System V init**, **Python** 런타임 환경 및 일부 셸 유틸리티)이 포함되지 않습니다. 이미지 빌드를 위해 **RHEL** 리포지토리를 사용하는 동시에 가능한 최소 오버헤드 양을 전송할 수 있습니다.

•

microdnf 모듈이 지원됩니다: **microdnf** 명령에 사용되는 모듈은 사용 가능한 경우 동일한 소프트웨어의 여러 버전을 설치할 수 있습니다. **microdnf module enable**, **microdnf module disable**, **microdnf module reset** 을 사용하여 모듈 스트림을 각각 활성화, 비활성화 및 재설정할 수 있습니다.

○

예를 들어 **UBI** 최소 컨테이너 내에서 **nodejs:14** 모듈 스트림을 활성화하려면 다음을 입력합니다.

```
# microdnf module enable nodejs:14
Downloading metadata...
...
Enabling module streams:
nodejs:14

Running transaction test...
```

Red Hat은 최신 버전의 **UBI**만 지원하며 도트 릴리스에서는 지원하지 않습니다. 특정 점 릴리스에 대해 자세히 알아보려면 **EUS (Extended Update Support)**를 참조하십시오.

2.6. UBI 마이크로 이미지 이해

ubi-micro 는 패키지 관리자 및 일반적으로 컨테이너 이미지에 포함된 모든 종속 항목을 제외하고 얻은 최소 **UBI** 이미지입니다. 따라서 **ubi-micro** 이미지를 기반으로 컨테이너 이미지의 공격 면적을 최소화하고 다른 애플리케이션에 **UBI Standard**, **Minimal** 또는 **Init**을 사용하는 경우에도 최소 애플리케이션에 적합합니다. **Linux** 배포 패키징이 없는 컨테이너 이미지를 **Distroless** 컨테이너 이미지라고 합니다.

2.7. UBI INIT 이미지 사용

다음 절차에서는 컨테이너가 호스트 시스템에서 실행될 때 **systemd** 서비스(/sbin/init)에서 자동으로 시작하도록 웹 서버(**httpd**)를 설치하고 구성하는 **Containerfile** 을 사용하여 컨테이너를 빌드하는 방법을 보여줍니다. **podman build** 명령은 하나 이상의 **Containerfiles** 및 지정된 빌드 컨텍스트 디렉터리의 지침을 사용하여 이미지를 빌드합니다. 컨텍스트 디렉터리는 아카이브, **Git** 리포지토리 또는 **Containerfile** 의 **URL**로 지정할 수 있습니다. 컨텍스트 디렉터리를 지정하지 않으면 현재 작업 디렉터리가 빌드 컨텍스트로 간주되며 컨테이너 파일이 포함되어야 합니다. **--file** 옵션을 사용하여 **Containerfile** 을 지정할 수도 있습니다.

절차

1.

새 디렉터리에 다음 콘텐츠를 사용하여 **Containerfile** 을 생성합니다.

```
FROM registry.access.redhat.com/ubi8/ubi-init
RUN yum -y install httpd; yum clean all; systemctl enable httpd;
RUN echo "Successful Web Server Test" > /var/www/html/index.html
RUN mkdir /etc/systemd/system/httpd.service.d; echo -e '[Service]\nRestart=always' >
/etc/systemd/system/httpd.service.d/httpd.conf
EXPOSE 80
CMD [ "/sbin/init" ]
```

Containerfile 은 **httpd** 패키지를 설치하고, 부팅 시 **httpd** 서비스가 시작되고, 테스트 파일 (**index.html**)을 만들고, 컨테이너를 시작할 때 웹 서버를 호스트(포트 80)에 노출하며 **systemd** **init** 서비스(**/sbin/init**)를 시작합니다.

2.

컨테이너를 빌드합니다.

```
# podman build --format=docker -t mysysd .
```

3.

선택 사항: **systemd**를 사용하여 컨테이너를 실행하고 **SELinux**가 활성화되어 있는 경우 **container_manage_cgroup** 부울 변수를 설정해야 합니다.

```
# setsebool -P container_manage_cgroup 1
```

4.

이름이 **mysysd_run** 인 컨테이너를 실행합니다.

```
# podman run -d --name=mysysd_run -p 80:80 mysysd
```

mysysd 이미지는 **mysysd_run** 컨테이너에서 호스트 시스템의 포트 80으로 노출된 컨테이너의 포트 80을 데몬 프로세스로 실행합니다.

참고

rootless 모드에서는 호스트 포트 번호 **>= 1024**를 선택해야 합니다. 예를 들면 다음과 같습니다.

```
$ podman run -d --name=mysysd -p 8081:80 mysysd
```

포트 번호 **< 1024**를 사용하려면 **net.ipv4.ip_unprivileged_port_start** 변수를 수정해야 합니다.

```
$ sudo sysctl net.ipv4.ip_unprivileged_port_start=80
```

5. 컨테이너가 실행 중인지 확인합니다.

```
# podman ps
a282b0c2ad3d localhost/mysysd:latest /sbin/init 15 seconds ago Up 14 seconds ago
0.0.0.0:80->80/tcp mysysd_run
```

6. 웹 서버를 테스트합니다.

```
# curl localhost/index.html
Successful Web Server Test
```

추가 리소스

- [Rootless Podman의 단점](#)

2.8. UBI 마이크로 이미지 사용

다음 절차에서는 **Buildah** 툴을 사용하여 **ubi-micro** 컨테이너 이미지를 빌드하는 방법을 설명합니다.

사전 요구 사항

- **container-tools** 모듈이 설치되어 있어야 합니다.

```
# yum module install -y container-tools
```

절차

1. **registry.access.redhat.com/ubi8/ubi-micro** 이미지를 가져와서 빌드합니다.

```
# microcontainer=$(buildah from registry.access.redhat.com/ubi8/ubi-micro)
```

2. 작동하는 컨테이너 루트 파일 시스템을 마운트합니다.

```
# micromount=$(buildah mount $microcontainer)
```

3.

micromount 디렉터리에 **httpd** 서비스를 설치합니다.

```
# yum install \
  --installroot $micromount \
  --releasever 8 \
  --setopt install_weak_deps=false \
  --nodocs -y \
  httpd
# yum clean all \
  --installroot $micromount
```

4.

작업 컨테이너에서 루트 파일 시스템을 마운트 해제합니다.

```
# buildah umount $microcontainer
```

5.

작업 중인 컨테이너에서 **ubi-micro-httpd** 이미지를 만듭니다.

```
# buildah commit $microcontainer ubi-micro-httpd
```

검증 단계

1.

ubi-micro-httpd 이미지에 대한 세부 정보를 표시합니다.

```
# podman images ubi-micro-httpd
localhost/ubi-micro-httpd latest 7c557e7fbe9f 22 minutes ago 151 MB
```

3장. 컨테이너 이미지 작업

Podman 툴은 컨테이너 이미지에서 작동하도록 설계되었습니다. 이 도구를 사용하여 이미지, 검사, 태그, 저장, 로드, 재배포 및 이미지 서명을 정의할 수 있습니다.

3.1. 컨테이너 레지스트리

컨테이너 레지스트리는 컨테이너 이미지 및 컨테이너 기반 애플리케이션 아티팩트를 저장하기 위한 리포지토리 또는 리포지토리 컬렉션입니다. **Red Hat**이 제공하는 레지스트리는 다음과 같습니다.

- **registry.redhat.io**(인증 필요)
- **registry.access.redhat.com** (인증이 필요하지 않음)
- **registry.connect.redhat.com** (Red Hat Partner Connect 프로그램 이미지 포함)

Red Hat의 자체 컨테이너 레지스트리와 같은 원격 레지스트리에서 컨테이너 이미지를 가져와 로컬 시스템에 추가하려면 **podman pull** 명령을 사용합니다.

```
# podman pull <registry>[:<port>]/[<namespace>]/<name>:<tag>
```

여기서 **<registry>[:<port>]/[<namespace>]/<name>:<tag>** 는 컨테이너 이미지의 이름입니다.

예를 들어 **registry.redhat.io/ubi8/ubi** 컨테이너 이미지는 다음을 통해 식별됩니다.

- 레지스트리 서버(**registry.redhat.io**)
- 네임 스페이스(**ubi8**)
- 이미지 이름(**ubi**)

동일한 이미지가 여러 개 있는 경우 이미지 이름을 명시적으로 지정하는 태그를 추가합니다. 기본적으

로 **Podman**은 **:latest** 태그(예: **ubi8/ubi:latest**)를 사용합니다.

일부 레지스트리는 **<namespace>**를 사용하여 다른 사용자 또는 조직에서 소유한 동일한 **<name>**의 이미지를 구분합니다. 예를 들면 다음과 같습니다.

네임스페이스	예 (<namespace>/<name>)
조직	redhat/kubernetes, google/kubernetes
로그인 (사용자 이름)	alice/application, bob/application
role	devel/database, test/database, prod/database

registry.redhat.io로 전환하는 방법에 대한 자세한 내용은 [Red Hat Container Registry Authentication](#)을 참조하십시오. **registry.redhat.io**에서 컨테이너를 가져오려면 **RHEL** 서브스크립션 자격 증명을 사용하여 인증해야 합니다.

3.2. 컨테이너 레지스트리 구성

registries.conf 구성 파일에서 컨테이너 레지스트리 목록을 찾을 수 있습니다. 루트 사용자로 **/etc/containers/registries.conf** 파일을 편집하여 기본 시스템 전체 검색 설정을 변경합니다.

사용자로 **\$HOME/.config/containers/registries.conf** 파일을 생성하여 시스템 전체 설정을 재정의합니다.

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

기본적으로 **podman pull** 및 **podman search** 명령은 지정된 순서로 **unqualified-search-registries** 목록에 나열된 레지스트리에서 컨테이너 이미지를 검색합니다.

로컬 컨테이너 레지스트리 구성

TLS 확인 없이 로컬 컨테이너 레지스트리를 구성할 수 있습니다. **TLS** 확인을 비활성화하는 방법에는 두 가지 옵션이 있습니다. 먼저 **Podman**에서 **--tls-verify=false** 옵션을 사용할 수 있습니다. 둘째, **registries.conf** 파일에 **insecure=true**를 설정할 수 있습니다.

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

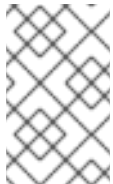
레지스트리, 네임스페이스 또는 이미지 차단

로컬 시스템이 액세스할 수 없는 레지스트리를 정의할 수 있습니다. **blocked=true** 로 설정하여 특정 레지스트리를 차단할 수 있습니다.

```
[[registry]]
location = "registry.example.org"
blocked = true
```

접두사를 **prefix="registry.example.org/namespace"** 로 설정하여 네임스페이스를 차단할 수도 있습니다. 예를 들어, 지정된 접두사가 일치하므로 **podman pull registry**를 사용하여 이미지를 가져옵니다. **example.org/example/image:latest** 명령이 차단됩니다.

```
[[registry]]
location = "registry.example.org"
prefix="registry.example.org/namespace"
blocked = true
```



참고

prefix 는 선택 사항이며 기본값은 **location** 값과 동일합니다.

prefix="registry.example.org/namespace/image" 를 설정하여 특정 이미지를 차단할 수 있습니다.

```
[[registry]]
location = "registry.example.org"
prefix="registry.example.org/namespace/image"
blocked = true
```

레지스트리 미러링

원본 레지스트리에 액세스할 수 없는 경우 레지스트리 미러를 설정할 수 있습니다. 예를 들어 중요도가 높은 환경에서 작업하므로 인터넷에 연결할 수 없습니다. 지정된 순서로 연결하는 미러를 여러 개 지정할 수 있습니다. 예를 들어 **podman pull registry.example.com/myimage:latest** 명령을 실행하면 **mirror-1.com** 이 먼저 시도되고 **mirror-2.com** 이 먼저 시도됩니다.

```
[[registry]]
location="registry.example.com"
[[registry.mirror]]
location="mirror-1.com"
[[registry.mirror]]
location="mirror-2.com"
```

추가 리소스

- [Linux 컨테이너 레지스트리 관리 방법](#)

3.3. 컨테이너 이미지 검색

podman search 명령을 사용하여 선택한 컨테이너 레지스트리에서 이미지의 이미지를 검색할 수 있습니다. [Red Hat Container Catalog](#) 에서 이미지를 검색할 수도 있습니다. **Red Hat Container Registry** 에는 이미지 설명, 콘텐츠, 상태 색인 및 기타 정보가 포함되어 있습니다.



참고

podman search 명령은 이미지의 존재 또는 존재를 확인하는 신뢰할 수 있는 방법이 아닙니다. **v1** 및 **v2 Docker** 배포 **API**의 **Podman** 검색 동작은 각 레지스트리의 구현에 따라 다릅니다. 일부 레지스트리는 검색을 전혀 지원하지 않을 수 있습니다. 검색 용어가 없는 검색은 **v2 API**를 구현하는 레지스트리에서만 작동합니다. **docker search** 명령도 마찬가지입니다.

이 섹션에서는 **quay.io** 레지스트리에서 **postgresql-10** 이미지를 검색하는 방법을 설명합니다.

사전 요구 사항

- 레지스트리가 구성되어 있습니다.

절차

1. 레지스트리에 인증합니다.

```
# podman login quay.io
```

2. 이미지를 검색합니다.

- 특정 레지스트리에서 특정 이미지를 검색하려면 다음을 입력합니다.

```
podman search quay.io/postgresql-10
INDEX      NAME
AUTOMATED
```

DESCRIPTION

STARS OFFICIAL

```
redhat.io registry.redhat.io/rhel8/postgresql-10 This container image ... 0
redhat.io registry.redhat.io/rhsc1/postgresql-10-rhel7 PostgreSQL is an ... 0
```

- 또는 특정 레지스트리에서 제공하는 모든 이미지를 표시하려면 다음을 입력합니다.

```
# podman search quay.io/
```

- 모든 레지스트리에서 이미지 이름을 검색하려면 다음을 입력합니다.

```
# podman search postgresql-10
```

전체 설명을 표시하려면 명령에 **--no-trunc** 옵션을 전달합니다.

추가 리소스

- **podman-search** 도움말 페이지

3.4. 레지스트리에서 이미지 가져오기

podman pull 명령을 사용하여 이미지를 로컬 시스템으로 가져옵니다.

절차

1. **registry.redhat.io** 레지스트리에 로그인합니다.

```
$ podman login registry.redhat.io
Username: username
Password: *****
Login Succeeded!
```

2. **registry.redhat.io/ubi8/ubi** 컨테이너 이미지를 가져옵니다.

```
$ podman pull registry.redhat.io/ubi8/ubi
```

검증 단계

- 로컬 시스템으로 가져온 모든 이미지를 나열합니다.

```
$ podman images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
registry.redhat.io/ubi8/ubi	latest	3269c37eae33	7 weeks ago	208 MB

추가 리소스

- [podman-pull 도움말 페이지](#)

3.5. 짧은 이름 별칭 구성

Red Hat은 항상 정규화된 이름으로 이미지를 가져올 것을 권장합니다. 그러나 이미지를 짧은 이름으로 가져오는 것이 일반적입니다. 예를 들어 `registry.access.redhat.com/ubi8:latest` 대신 `ubi8` 을 사용할 수 있습니다.

`registries.conf` 파일을 사용하면 짧은 이름에 대한 별칭을 지정할 수 있으므로 관리자는 이미지를 가져온 위치를 완전히 제어할 수 있습니다. 별칭은 `"name" = "value"` 형식의 `[aliases]` 테이블에 지정됩니다. `/etc/containers/registries.conf.d` 디렉토리에 별칭 목록을 볼 수 있습니다. Red Hat은 이 디렉토리에 별칭 집합을 제공합니다. 예를 들어 `podman pull ubi8` 은 올바른 이미지인 `registry.access.redhat.com/ubi8:latest`로 직접 확인됩니다.

예를 들면 다음과 같습니다.

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

```
[aliases]
"fedora"="registry.fedoraproject.org/fedora"
```

짧은 이름 모드는 다음과 같습니다.

- **강제:** 이미지를 가져오는 동안 일치하는 별칭을 찾을 수 없는 경우 Podman은 사용자에게 정규화되지 않은 검색 레지스트리 중 하나를 선택하라는 메시지를 표시합니다. 선택한 이미지를 성공적으로 가져오면 Podman은 `$HOME/.cache/containers/short-name-aliases.conf` 파일 (rootless user) 또는 `/var/cache/containers/short-name-aliases.conf` (root user)에 새로운 짧은 이름 별칭을 자동으로 기록합니다. 사용자에게 메시지가 표시될 수 없는 경우(예: `stdin` 또는 `stdout`은 TTY가 아님) Podman이 실패합니다. 둘 다 동일한 별칭을 지정하는 경우 `short-name-aliases.conf` 파일이 `registries.conf` 파일보다 우선합니다.
- **Permissive:** 강제 모드와 유사하지만, 사용자에게 메시지를 표시할 수 없는 경우 Podman은 실패하지 않습니다. 대신 Podman은 지정된 순서로 모든 정규화되지 않은 레지스트리를 검색합니다. 별칭은 기록되지 않습니다.

- **disabled:** 정규화되지 않은 모든 레지스트리는 지정된 순서로 시도되며 별칭은 기록되지 않습니다.



참고

레지스트리, 네임스페이스, 이미지 이름, 태그 등의 정규화된 이미지 이름을 사용하는 것이 좋습니다. 이름을 사용할 때는 항상 스푸핑 위험이 있습니다. 알 수 없거나 익명 사용자가 임의의 이름으로 계정을 생성할 수 없는 레지스트리인 신뢰할 수 있는 레지스트리를 추가합니다. 예를 들어 사용자는 **example.registry.com** 레지스트리에서 예제 컨테이너 이미지를 가져오려고 합니다. **example.registry.com** 이 검색 목록에 처음 없는 경우 공격자는 검색 목록의 앞부분에서 레지스트리에 다른 예제 이미지를 배치할 수 있습니다. 사용자가 의도한 콘텐츠가 아니라 공격자 이미지를 실수로 가져와서 실행했습니다.

추가 리소스

- [Podman의 컨테이너 이미지 단축 이름](#)

3.6. 짧은 이름 별칭을 사용하여 컨테이너 이미지 가져오기

보안 짧은 이름을 사용하여 이미지를 로컬 시스템으로 가져올 수 있습니다. 다음 절차에서는 **fedora** 또는 **nginx** 컨테이너 이미지를 가져오는 방법을 설명합니다.

절차

- 컨테이너 이미지를 가져옵니다.
 - **fedora** 이미지를 가져옵니다.

```
$ podman pull fedora
Resolved "fedora" as an alias (/etc/containers/registries.conf.d/000-
shortnames.conf)
Trying to pull registry.fedoraproject.org/fedora:latest...
...
Storing signatures
...
```

별칭을 찾고 **registry.fedoraproject.org/fedora** 이미지를 안전하게 가져옵니다. **unqualified-search-registries** 목록은 **fedora** 이미지 이름을 확인하는 데 사용되지 않습니다.

○

nginx 이미지를 가져옵니다.

```
$ podman pull nginx
? Please select an image:
registry.access.redhat.com/nginx:latest
registry.redhat.io/nginx:latest
  ▶ docker.io/library/nginx:latest
✓ docker.io/library/nginx:latest
Trying to pull docker.io/library/nginx:latest...
...
Storing signatures
...
```

일치하는 별칭이 없으면 **unqualified-search-registries** 목록 중 하나를 선택하라는 메시지가 표시됩니다. 선택한 이미지를 성공적으로 가져오면 새로운 짧은 이름 별칭이 로컬에 기록되고, 그렇지 않으면 오류가 발생합니다.

검증

●

로컬 시스템으로 가져온 모든 이미지를 나열합니다.

```
$ podman images
REPOSITORY                                TAG    IMAGE ID    CREATED    SIZE
registry.fedoraproject.org/fedora        latest 28317703decd 12 days ago 184 MB
docker.io/library/nginx                  latest 08b152afcfae 13 days ago 137 MB
```

추가 리소스

●

[Podman의 컨테이너 이미지 단축 이름](#)

3.7. 이미지 나열

podman images 명령을 사용하여 로컬 스토리지의 이미지를 나열합니다.

사전 요구 사항

●

가져온 이미지는 로컬 시스템에서 사용할 수 있습니다.

절차

- 로컬 스토리지의 모든 이미지를 나열합니다.

```
$ podman images
REPOSITORY                                TAG      IMAGE ID      CREATED      SIZE
registry.access.redhat.com/ubi8/ubi      latest   3269c37eae33  6 weeks ago  208 MB
```

추가 리소스

- podman-images** 도움말 페이지

3.8. 로컬 이미지 검사

로컬 시스템으로 이미지를 가져와서 실행한 후 **podman inspect** 명령을 사용하여 이미지를 조사할 수 있습니다. 예를 들어, 이미지를 사용하여 이미지의 기능을 이해하고 해당 이미지 내의 소프트웨어를 확인합니다. **podman inspect** 명령은 이름 또는 ID로 식별된 컨테이너 및 이미지에 대한 정보를 표시합니다.

사전 요구 사항

- 가져온 이미지는 로컬 시스템에서 사용할 수 있습니다.

절차

- registry.redhat.io/ubi8/ubi** 이미지를 검사합니다.

```
$ podman inspect registry.redhat.io/ubi8/ubi
...
"Cmd": [
  "/bin/bash"
],
"Labels": {
  "architecture": "x86_64",
  "build-date": "2020-12-10T01:59:40.343735",
  "com.redhat.build-host": "cpt-1002.osbs.prod.upshift.rdu2.redhat.com",
  "com.redhat.component": "ubi8-container",
  "com.redhat.license_terms": "https://www.redhat.com/...,
  "description": "The Universal Base Image is ..."
}
...
```

"Cmd" 키는 컨테이너 내에서 실행할 기본 명령을 지정합니다. **podman run** 명령에 인수로 명령을 지정하여 이 명령을 재정의할 수 있습니다. **podman run**으로 시작할 때 다른 인수를 지정

하지 않으면 이 **ubi8/ubi** 컨테이너는 **bash** 셸을 실행합니다. "**Entrypoint**" 키가 설정된 경우 값은 "**Cmd**" 값 대신 사용되며 "**Cmd**" 값은 **Entrypoint** 명령에 대한 인수로 사용됩니다.

추가 리소스

- [podman-inspect 도움말 페이지](#)

3.9. 원격 이미지 검사

이미지를 시스템으로 가져오기 전에 **skopeo inspect** 명령을 사용하여 원격 컨테이너 레지스트리의 이미지에 대한 정보를 표시합니다.

절차

- [registry.redhat.io/ubi8/ubi-init](#) 이미지를 검사합니다.

```
# skopeo inspect docker://registry.redhat.io/ubi8/ubi-init
{
  "Name": "registry.redhat.io/ubi8/ubi8-init",
  "Digest": "sha256:c6d1e50ab...",
  "RepoTags": [
    ...
    "latest"
  ],
  "Created": "2020-12-10T07:16:37.250312Z",
  "DockerVersion": "1.13.1",
  "Labels": {
    "architecture": "x86_64",
    "build-date": "2020-12-10T07:16:11.378348",
    "com.redhat.build-host": "cpt-1007.osbs.prod.upshift.rdu2.redhat.com",
    "com.redhat.component": "ubi8-init-container",
    "com.redhat.license_terms": "https://www.redhat.com/en/about/red-hat-end-user-
license-agreements#UBI",
    "description": "The Universal Base Image Init is designed to run an init system as
PID 1 for running multi-services inside a container
    ...
  }
```

추가 리소스

- [Skopeo-inspect 도움말 페이지](#)

3.10. 컨테이너 이미지 복사

skopeo copy 명령을 사용하여 하나의 레지스트리에서 다른 레지스트리로 컨테이너 이미지를 복사할

수 있습니다. 예를 들어 외부 레지스트리의 이미지를 사용하여 내부 리포지토리를 채우거나 다른 두 위치에 이미지 레지스트리를 동기화할 수 있습니다.

절차

- **docker://quay.io**에서 **docker://registry.example.com**에 **skopeo** 컨테이너 이미지를 복사합니다.

```
$ skopeo copy docker://quay.io/skopeo/stable:latest
docker://registry.example.com/skopeo:latest
```

추가 리소스

- **Skopeo-copy** 도움말 페이지

3.11. 로컬 디렉터리에 이미지 계층 복사

skopeo copy 명령을 사용하여 컨테이너 이미지의 계층을 로컬 디렉터리에 복사할 수 있습니다.

절차

1. **/var/lib/images/nginx** 디렉터를 생성합니다.

```
$ mkdir -p /var/lib/images/nginx
```

2. **docker://docker.io/nginx:latest** 이미지 의 계층을 새로 생성된 디렉터리에 복사합니다.

```
$ skopeo copy docker://docker.io/nginx:latest dir:/var/lib/images/nginx
```

검증

- **/var/lib/images/nginx** 디렉터리의 내용을 표시합니다.

```
$ ls /var/lib/images/nginx
08b11a3d692c1a2e15ae840f2c15c18308dcb079aa5320e15d46b62015c0f6f3
...
4fcb23e29ba19bf305d0d4b35412625fea51e82292ec7312f9be724cb6e31ffd
manifest.json
version
```

추가 리소스

- [Skopeo-copy 도움말 페이지](#)

3.12. 이미지 태그 지정

podman tag 명령을 사용하여 로컬 이미지에 추가 이름을 추가합니다. 이 추가 이름은 **registryhost/username/NAME:tag** 의 여러 부분으로 구성될 수 있습니다.

사전 요구 사항

- 가져온 이미지는 로컬 시스템에서 사용할 수 있습니다.

절차

1. 모든 이미지를 나열합니다.

```
$ podman images
REPOSITORY          TAG         IMAGE ID      CREATED       SIZE
registry.redhat.io/ubi8/ubi      latest  3269c37eae33  7 weeks ago  208 MB
```

2. 다음 중 하나를 사용하여 **myubi** 이름을 **registry.redhat.io/ubi8/ubi** 이미지에 할당합니다.

- 이미지 이름:

```
$ podman tag registry.redhat.io/ubi8/ubi myubi
```

- 이미지 ID:

```
$ podman tag 3269c37eae33 myubi
```

두 명령 모두 동일한 결과를 제공합니다.

3. 모든 이미지를 나열합니다.

```
$ podman images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
registry.redhat.io/ubi8/ubi	latest	3269c37eae33	2 months ago	208 MB
localhost/myubi	latest	3269c37eae33	2 months ago	208 MB

기본 태그는 두 이미지 모두 최신입니다. 모든 이미지 이름이 단일 이미지 ID **3269c37eae33**에 할당되어 있음을 확인할 수 있습니다.

4.

다음 중 하나를 사용하여 **registry.redhat.io/ubi8/ubi** 이미지에 **8** 태그를 추가합니다.

-

이미지 이름:

```
$ podman tag registry.redhat.io/ubi8/ubi myubi:8
```

-

이미지 ID:

```
$ podman tag 3269c37eae33 myubi:8
```

두 명령 모두 동일한 결과를 제공합니다.

5.

모든 이미지를 나열합니다.

```
$ podman images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
registry.redhat.io/ubi8/ubi	latest	3269c37eae33	2 months ago	208 MB
localhost/myubi	latest	3269c37eae33	2 months ago	208 MB
localhost/myubi	8	3269c37eae33	2 months ago	208 MB

기본 태그는 두 이미지 모두 최신입니다. 모든 이미지 이름이 단일 이미지 ID **3269c37eae33**에 할당되어 있음을 확인할 수 있습니다.

registry.redhat.io/ubi8/ubi 이미지에 태그를 지정한 후에는 다음 세 가지 옵션을 사용하여 컨테이너를 실행할 수 있습니다.

-

ID 별(3269c37eae33)

-

이름(localhost/myubi:latest)

- 이름(**localhost/myubi:8**)

추가 리소스

- **podman-tag man** 페이지

3.13. 이미지 저장 및 로드

podman save 명령을 사용하여 이미지를 컨테이너 아카이브에 저장합니다. 나중에 다른 컨테이너 환경으로 복원하거나 다른 컨테이너 환경으로 보낼 수 있습니다. **format** 옵션을 사용하여 아카이브 형식을 지정할 수 있습니다. 지원되는 형식은 다음과 같습니다.

- **docker-archive**
- **OCI-archive**
- **OCI-dir** (oci 매니페스트 유형이 있는 디렉토리)
- **docker-dir** (v2s2 매니페스트 유형이 있는 디렉토리)

기본 형식은 **docker-dir** 형식입니다.

podman load 명령을 사용하여 컨테이너 이미지 아카이브의 이미지를 컨테이너 스토리지로 로드합니다.

사전 요구 사항

- 가져온 이미지는 로컬 시스템에서 사용할 수 있습니다.

절차

1. **registry.redhat.io/rhel8/rsyslog** 이미지를 **tarball**로 저장합니다.

- 기본 **docker-dir** 형식은 다음과 같습니다.

```
$ podman save -o myrsyslog.tar registry.redhat.io/rhel8/rsyslog:latest
```

- **oci-archive** 형식에서 **--format** 옵션을 사용합니다.

```
$ podman save -o myrsyslog-oci.tar --format=oci-archive
registry.redhat.io/rhel8/rsyslog
```

myrsyslog.tar 및 **myrsyslog-oci.tar** 아카이브는 현재 디렉토리에 저장됩니다. 다음 단계는 **myrsyslog.tar tarball**을 사용하여 수행합니다.

2. **myrsyslog.tar**의 파일 유형을 확인합니다.

```
$ file myrsyslog.tar
myrsyslog.tar: POSIX tar archive
```

3. **myrsyslog .tar**에서 **registry.redhat.io/rhel8/rsyslog:latest** 이미지를 로드하려면 다음을 수행합니다.

```
$ podman load -i myrsyslog.tar
...
Loaded image(s): registry.redhat.io/rhel8/rsyslog:latest
```

추가 리소스

- **podman-save man** 페이지

3.14. UBI 이미지 재배포

podman push 명령을 사용하여 **UBI** 이미지를 자체 또는 타사 레지스트리로 푸시하고 다른 사용자와 공유합니다. **UBI yum** 리포지토리에서 원하는 대로 해당 이미지를 업그레이드하거나 추가할 수 있습니다.

사전 요구 사항

- 가져온 이미지는 로컬 시스템에서 사용할 수 있습니다.

절차

1. 선택 사항: **ubi** 이미지에 추가 이름을 추가합니다.

```
# podman tag registry.redhat.io/ubi8/ubi registry.example.com:5000/ubi8/ubi
```

2. 로컬 스토리지에서 **registry.example.com:5000/ubi8/ubi** 이미지를 레지스트리로 푸시합니다.

```
# podman push registry.example.com:5000/ubi8/ubi
```

중요

이러한 이미지 사용 방법에 대한 제한 사항은 거의 없지만 참조하는 방법에 대한 몇 가지 제한 사항이 있습니다. 예를 들어 **Red Hat Container Certification** 또는 **Red Hat OpenShift Operator** 인증을 통해 인증하지 않는 한 **Red Hat 인증 또는 Red Hat 지원** 이미지를 호출할 수 없습니다.

3.15. 이미지 서명의 기본 확인

Red Hat Container Registries `/etc/containers/registries.d/registry.access.redhat.com.yaml` 및 `/etc/containers/registries.d/registry.redhat.io.yaml` 파일의 정책 **YAML** 파일은 **container - tools:latest** 모듈에 포함된 **containers- common** 패키지에 포함되어 있습니다. **podman image trust** 명령을 사용하여 **RHEL**에서 컨테이너 이미지 서명을 확인합니다.

절차

1. **registry.access.redhat.com**에 대한 기존 신뢰 범위를 업데이트합니다.

```
# podman image trust set -f /etc/pki/rpm-gpg/RPM-GPG-KEY-redhat-release registry.access.redhat.com
```

2. 선택 사항: 신뢰 정책 구성을 확인하려면 `/etc/containers/policy.json` 파일을 표시합니다.

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

```
    ]
  },
  ...

```

3.

registry.redhat.io의 기존 신뢰 범위를 업데이트합니다.

```
# podman image trust set -f /etc/pki/rpm-gpg/RPM-GPG-KEY-redhat-release
registry.redhat.io
```

4.

선택 사항: 신뢰 정책 구성을 확인하려면 `/etc/containers/policy.json` 파일을 표시합니다.

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

```

추가 리소스

- [podman-image-trust](#) 도움말 페이지

3.16. 이미지 제거

`podman rmi` 명령을 사용하여 로컬에 저장된 컨테이너 이미지를 제거합니다. ID 또는 이름으로 이미지를 제거할 수 있습니다.

절차

1. 로컬 시스템의 모든 이미지를 나열합니다.

```
$ podman images
REPOSITORY          TAG   IMAGE ID   CREATED   SIZE
registry.redhat.io/rhel8/rsyslog  latest  4b32d14201de  7 weeks ago  228 MB
registry.redhat.io/ubi8/ubi      latest  3269c37eae33  7 weeks ago  208 MB
localhost/myubi              X.Y    3269c37eae33  7 weeks ago  208 MB
```

2.

모든 컨테이너를 나열합니다.

```
$ podman ps -a
CONTAINER ID   IMAGE                                COMMAND                  CREATED        STATUS
PORTS         NAMES
7ccd6001166e  registry.redhat.io/rhel8/rsyslog:latest  /bin/rsyslog.sh  6 seconds ago  Up 5 seconds ago  msyslog
```

registry.redhat.io/rhel8/rsyslog 이미지를 제거하려면 **podman stop** 명령을 사용하여 이 이미지에서 실행 중인 모든 컨테이너를 중지해야 합니다. 컨테이너를 **ID** 또는 이름으로 중지할 수 있습니다.

3.

mysyslog 컨테이너를 중지합니다.

```
$ podman stop msyslog
7ccd6001166e9720c47fbeb077e0afd0bb635e74a1b0ede3fd34d09eaf5a52e9
```

4.

registry.redhat.io/rhel8/rsyslog 이미지를 제거합니다.

```
$ podman rmi registry.redhat.io/rhel8/rsyslog
```

- 여러 이미지를 제거하려면 다음을 수행합니다.

```
$ podman rmi registry.redhat.io/rhel8/rsyslog registry.redhat.io/ubi8/ubi
```

- 시스템에서 모든 이미지를 제거하려면 다음을 수행합니다.

```
$ podman rmi -a
```

- 여러 이름(태그)이 연결된 이미지를 제거하려면 **-f** 옵션을 추가하여 제거합니다.

```
$ podman rmi -f 1de7d7b3f531
1de7d7b3f531...
```

추가 리소스

- [podman-rmi 매뉴얼 페이지](#)

4장. 컨테이너 작업

컨테이너는 압축 해제된 컨테이너 이미지에 있는 파일에서 생성된 실행 중이거나 중지된 프로세스를 나타냅니다. **Podman** 툴을 사용하여 컨테이너 작업을 수행할 수 있습니다.

4.1. PODMAN RUN 명령

podman run 명령은 컨테이너 이미지를 기반으로 새 컨테이너에서 프로세스를 실행합니다. 컨테이너 이미지가 아직 로드되지 않은 경우 **podman run** 은 해당 이미지에서 컨테이너를 시작하기 전에 **podman pull image** 를 실행하는 것과 동일한 방식으로 리포지토리에서 이미지 및 모든 이미지 종속성을 가져옵니다. 컨테이너 프로세스에는 자체 파일 시스템, 자체 네트워킹 및 자체 격리된 프로세스 트리가 있습니다.

podman run 명령에는 다음 형식이 있습니다.

```
podman run [options] image [command [arg ...]]
```

기본 옵션은 다음과 같습니다.

- **--detach (-d):** 컨테이너를 백그라운드에서 실행하고 새 컨테이너 ID를 출력합니다.
- **--attach (-a):** 포그라운드 모드에서 컨테이너를 실행합니다.
- **--name (-n):** 컨테이너에 이름을 할당합니다. **name**이 있는 컨테이너에 이름이 할당되지 않은 경우 임의의 문자열 이름을 생성합니다. 이 작업은 백그라운드 및 포그라운드 컨테이너 둘 다에서 작동합니다.
- **--rm:** 종료 시 컨테이너를 자동으로 제거합니다. 컨테이너를 성공적으로 생성하거나 시작할 수 없는 경우 컨테이너가 제거되지 않습니다.
- **--tty (-t):** 컨테이너의 표준 입력에 의사 터미널을 할당하고 연결합니다.
- **--interactive (-i):** 대화형 프로세스의 경우 **-i** 및 **-t** 를 함께 사용하여 컨테이너 프로세스의 터미널을 할당합니다. **i -t** 는 종종 **-it** 로 작성됩니다.

4.2. 호스트에서 컨테이너에서 명령 실행

다음 절차에서는 **podman run** 명령을 사용하여 컨테이너의 운영 체제 유형을 표시하는 방법을 설명합니다.

사전 요구 사항

- Podman 툴이 설치되어 있습니다.

```
# yum module install -y container-tools
```

절차

1. **cat /etc/os-release** 명령을 사용하여 **registry.access.redhat.com/ubi8/ubi** 컨테이너 이미지를 기반으로 컨테이너의 운영 체제 유형을 표시합니다.

```
$ podman run --rm registry.access.redhat.com/ubi8/ubi cat /etc/os-release
NAME="Red Hat Enterprise Linux"
...
ID="rhel"
...
HOME_URL="https://www.redhat.com/"
BUG_REPORT_URL="https://bugzilla.redhat.com/"

REDHAT_BUGZILLA_PRODUCT="Red Hat Enterprise Linux 8"
...
```

2. 선택 사항: 모든 컨테이너를 나열합니다.

```
$ podman ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
```

rm 옵션 때문에 컨테이너가 표시되지 않아야 합니다. 컨테이너가 제거되었습니다.

추가 리소스

- **podman-run** 도움말 페이지

4.3. 컨테이너 내에서 명령 실행

다음 절차에서는 **podman run** 명령을 사용하여 컨테이너를 대화식으로 실행하는 방법을 설명합니다.

사전 요구 사항

- Podman 툴이 설치되어 있습니다.

```
# yum module install -y container-tools
```

절차

1. registry.redhat.io/ubi8/ubi 이미지를 기반으로 my ubi라는 컨테이너를 실행합니다.

```
$ podman run --name=myubi -it registry.access.redhat.com/ubi8/ubi /bin/bash
[root@6ccffd0f6421 /]#
```

- i 옵션은 대화형 세션을 만듭니다. t 옵션을 사용하지 않으면 셸은 열려 있지만 셸에 아무 것도 입력할 수 없습니다.
- t 옵션은 터미널 세션을 엽니다. i 옵션을 사용하지 않으면 셸이 열리고 종료됩니다.

2. 일련의 시스템 유틸리티가 포함된 **procps-ng** 패키지를 설치합니다(예: **ps**, **top**, **uptime** 등).

```
[root@6ccffd0f6421 /]# yum install procps-ng
```

3. **ps -ef** 명령을 사용하여 현재 프로세스를 나열합니다.

```
# ps -ef
UID      PID  PPID  C STIME TTY      TIME CMD
root      1    0  0 12:55 pts/0    00:00:00 /bin/bash
root     31    1  0 13:07 pts/0    00:00:00 ps -ef
```

4. **exit** 를 입력하여 컨테이너를 종료하고 호스트로 돌아갑니다.

```
# exit
```

5. 선택 사항: 모든 컨테이너를 나열합니다.

```
$ podman ps
CONTAINER ID IMAGE COMMAND CREATED STATUS
PORTS NAMES
1984555a2c27 registry.redhat.io/ubi8/ubi:latest /bin/bash 21 minutes ago Exited (0)
21 minutes ago myubi
```

컨테이너가 **Exited** 상태인지 확인할 수 있습니다.

추가 리소스

- [podman-run 도움말 페이지](#)

4.4. 컨테이너 나열

podman ps 명령을 사용하여 시스템에서 실행 중인 컨테이너를 나열합니다.

사전 요구 사항

- **Podman** 툴이 설치되어 있습니다.

```
# yum module install -y container-tools
```

절차

1. **registry.redhat.io/rhel8/rsyslog** 이미지를 기반으로 하여 컨테이너를 실행합니다.

```
$ podman run -d registry.redhat.io/rhel8/rsyslog
```

2. 모든 컨테이너를 나열합니다.

- 실행 중인 컨테이너를 모두 나열하려면 다음을 수행합니다.

```
$ podman ps
CONTAINER ID IMAGE COMMAND CREATED STATUS
PORTS NAMES
74b1da000a11 rhel8/rsyslog /bin/rsyslog.sh 2 minutes ago Up About a minute
musing_brown
```

- 실행 중이거나 중지된 모든 컨테이너를 나열하려면 다음을 수행합니다.

```
$ podman ps -a
CONTAINER ID IMAGE          COMMAND          CREATED   STATUS    PORTS
NAMES      IS INFRA
d65aecc325a4 ubi8/ubi      /bin/bash       3 secs ago Exited (0) 5 secs ago
peaceful_hopper false
74b1da000a11 rhel8/rsyslog rsyslog.sh      2 mins ago Up About a minute
musing_brown  false
```

실행 중이 아니지만 제거되지 않은(--rm 옵션) 컨테이너가 있는 경우 컨테이너가 존재하며 다시 시작할 수 있습니다.

추가 리소스

- [podman-ps man 페이지](#)

4.5. 컨테이너 시작

컨테이너를 실행하고 중지한 다음 제거하지 않으면 컨테이너가 다시 실행할 준비가 된 로컬 시스템에 저장됩니다. **podman start** 명령을 사용하여 컨테이너를 다시 실행할 수 있습니다. 컨테이너 **ID** 또는 이름으로 컨테이너를 지정할 수 있습니다.

사전 요구 사항

- Podman 툴이 설치되어 있습니다.

```
# yum module install -y container-tools
```

- 하나 이상의 컨테이너가 중지되었습니다.

절차

1. **myubi** 컨테이너를 시작합니다.

- 비 대화형 모드에서 다음을 수행합니다.

```
$ podman start myubi
```

또는 `podman start 555a2c27` 을 사용할 수 있습니다.

- 대화형 모드에서 `-a (--attach)` 및 `-t (--interactive)` 옵션을 사용하여 컨테이너 `bash` 셸을 사용합니다.

```
$ podman start -a -i myubi
```

또는 `podman start -a -i 555a2c27` 을 사용할 수 있습니다.

2. `exit` 를 입력하여 컨테이너를 종료하고 호스트로 돌아갑니다.

```
[root@6ccffd0f6421 /]# exit
```

추가 리소스

- `podman-start` 도움말 페이지

4.6. 호스트에서 컨테이너 검사

`podman inspect` 명령을 사용하여 **JSON** 형식으로 기존 컨테이너의 메타데이터를 검사합니다. 컨테이너 ID 또는 이름으로 컨테이너를 지정할 수 있습니다.

사전 요구 사항

- Podman 툴이 설치되어 있습니다.

```
# yum module install -y container-tools
```

절차

- ID `64ad95327c74`로 정의된 컨테이너를 검사합니다.
 - 모든 메타데이터를 가져오려면 다음을 수행합니다.

```
$ podman inspect 64ad95327c74
[
```

```
{
  "Id":
  "64ad95327c740ad9de468d551c50b6d906344027a0e645927256cd061049f681",
  "Created": "2021-03-02T11:23:54.591685515+01:00",
  "Path": "/bin/rsyslog.sh",
  "Args": [
    "/bin/rsyslog.sh"
  ],
  "State": {
    "OciVersion": "1.0.2-dev",
    "Status": "running",
    ...
  }
}
```

○

JSON 파일에서 특정 항목을 가져오려면(예: **StartedAt** 타임스탬프).

```
$ podman inspect --format='{{.State.StartedAt}}' 64ad95327c74
2021-03-02 11:23:54.945071961 +0100 CET
```

정보는 계층 구조로 저장됩니다. 컨테이너 **StartedAt** 타임스탬프(시작됨) 타임스탬프가 **State** 상태에 있는 경우 **--format** 옵션과 컨테이너 **ID** 또는 이름을 사용합니다.

검사하려는 다른 항목의 예는 다음과 같습니다.

- **.path** 컨테이너와 함께 실행되는 명령을 확인합니다.
- 명령에 대한 인수 **.args**
- **.config.ExposedPorts** TCP 또는 UDP 포트가 컨테이너에서 노출됩니다.
- 컨테이너 의 프로세스 ID를 확인하는 **.state.Pid**
- **.HostConfig.PortBindings** 컨테이너에서 호스트로의 포트 매핑

추가 리소스

- **podman-inspect** 도움말 페이지

4.7. LOCALHOST의 디렉터리를 컨테이너에 마운트

다음 절차에서는 컨테이너 내부의 호스트 **/dev/log** 장치를 마운트하여 호스트 시스템에서 사용할 수 있는 컨테이너 내부에서 로그 메시지를 생성하는 방법을 보여줍니다.

사전 요구 사항

- **Podman** 툴이 설치되어 있습니다.

```
# yum module install -y container-tools
```

절차

1. **log_test** 라는 컨테이너를 실행하고 컨테이너 내부에 호스트 **/dev/log** 장치를 마운트합니다.

```
# podman run --name="log_test" -v /dev/log:/dev/log --rm \
registry.redhat.io/ubi8/ubi logger "Testing logging to the host"
```

2. **journalctl** 유틸리티를 사용하여 로그를 표시합니다.

```
# journalctl -b | grep Testing
Dec 09 16:55:00 localhost.localdomain root[14634]: Testing logging to the host
```

rm 옵션은 종료 시 컨테이너를 제거합니다.

추가 리소스

- **podman-run** 도움말 페이지

4.8. 컨테이너 파일 시스템 마운트

podman mount 명령을 사용하여 호스트에서 액세스할 수 있는 위치에 작업 중인 컨테이너 루트 파일 시스템을 마운트합니다.

사전 요구 사항

- **Podman** 툴이 설치되어 있습니다.

```
# yum module install -y container-tools
```

절차

1.

mysyslog 라는 컨테이너를 실행합니다.

```
# podman run -d --name=mysyslog registry.redhat.io/rhel8/rsyslog
```

2.

선택 사항: 모든 컨테이너를 나열합니다.

```
# podman ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
c56ef6a256f8	registry.redhat.io/rhel8/rsyslog:latest	/bin/rsyslog.sh	20 minutes ago	Up 20 minutes ago
		mysyslog		

3.

mysyslog 컨테이너를 마운트합니다.

```
# podman mount mysyslog
/var/lib/containers/storage/overlay/990b5c6ddcdeed4bde7b245885ce4544c553d108310
e2b797d7be46750894719/merged
```

4.

ls 명령을 사용하여 마운트 지점의 내용을 표시합니다.

```
# ls
/var/lib/containers/storage/overlay/990b5c6ddcdeed4bde7b245885ce4544c553d108310
e2b797d7be46750894719/merged
bin boot dev etc home lib lib64 lost+found media mnt opt proc root run sbin
srv sys tmp usr var
```

5.

OS 버전을 표시합니다.

```
# cat
/var/lib/containers/storage/overlay/990b5c6ddcdeed4bde7b245885ce4544c553d108310
e2b797d7be46750894719/merged/etc/os-release
NAME="Red Hat Enterprise Linux"
VERSION="8 (Ootpa)"
ID="rhel"
ID_LIKE="fedora"
...
```

추가 리소스

- [podman-mount man 페이지](#)

4.9. 정적 IP를 사용하여 데몬으로 서비스 실행

다음 예제에서는 **rsyslog** 서비스를 백그라운드에서 데몬 프로세스로 실행합니다. **ip** 옵션은 컨테이너 네트워크 인터페이스를 특정 **IP** 주소(예: **10.88.0.44**)로 설정합니다. 그런 다음 **podman inspect** 명령을 실행하여 **IP** 주소가 올바르게 설정되었는지 확인할 수 있습니다.

사전 요구 사항

- **Podman** 툴이 설치되어 있습니다.

```
# yum module install -y container-tools
```

절차

1. 컨테이너 네트워크 인터페이스를 **IP** 주소 **10.88.0.44**로 설정합니다.

```
# podman run -d --ip=10.88.0.44 registry.access.redhat.com/rhel8/rsyslog
efde5f0a8c723f70dd5cb5dc3d5039df3b962fae65575b08662e0d5b5f9fbe85
```

2. **IP** 주소가 올바르게 설정되었는지 확인합니다.

```
# podman inspect efde5f0a8c723 | grep 10.88.0.44
"IPAddress": "10.88.0.44",
```

추가 리소스

- [podman-inspect 도움말 페이지](#)
- [podman-run 도움말 페이지](#)

4.10. 실행 중인 컨테이너 내에서 명령 실행

podman exec 명령을 사용하여 실행 중인 컨테이너에서 명령을 실행하고 해당 컨테이너를 조사합니다. **podman run** 명령 대신 **podman exec** 명령을 사용하는 이유는 컨테이너 활동을 중단하지 않고 실행 중인 컨테이너를 조사할 수 있기 때문입니다.

사전 요구 사항

- Podman 툴이 설치되어 있습니다.

```
# yum module install -y container-tools
```

- 컨테이너가 실행 중입니다.

절차

1. 설치된 모든 패키지를 나열하려면 **myrsyslog** 컨테이너 내에서 **rpm -qa** 명령을 실행합니다.

```
$ podman exec -it myrsyslog rpm -qa
tzdata-2020d-1.el8.noarch
python3-pip-wheel-9.0.3-18.el8.noarch
redhat-release-8.3-1.0.el8.x86_64
filesystem-3.8-3.el8.x86_64
...
```

2. **myrsyslog** 컨테이너에서 **/bin/bash** 명령을 실행합니다.

```
$ podman exec -it myrsyslog /bin/bash
```

3. 일련의 시스템 유틸리티가 포함된 **procps-ng** 패키지를 설치합니다(예: **ps, top, uptime** 등).

```
# yum install procps-ng
```

4. 컨테이너를 검사합니다.

- 시스템의 모든 프로세스를 나열하려면 다음을 수행합니다.

```
# ps -ef
UID      PID  PPID  C  STIME TTY      TIME CMD
root      1    0  0  10:23 ?        00:00:01 /usr/sbin/rsyslogd -n
root      8    0  0  11:07 pts/0    00:00:00 /bin/bash
root     47    8  0  11:13 pts/0    00:00:00 ps -ef
```

- 파일 시스템 디스크 공간 사용을 표시하려면 다음을 수행합니다.

```
# df -h
Filesystem      Size  Used Avail Use% Mounted on
fuse-overlayfs  27G  7.1G  20G   27% /
tmpfs           64M   0  64M   0% /dev
tmpfs           269M  936K  268M   1% /etc/hosts
shm             63M   0  63M   0% /dev/shm
...
```

- 시스템 정보를 표시하려면 다음을 수행합니다.

```
# uname -r
4.18.0-240.10.1.el8_3.x86_64
```

- 사용 가능한 메모리 및 사용된 메모리의 양을 메가바이트 단위로 표시하려면 다음을 수행합니다.

```
# free --mega
total      used      free   shared  buff/cache   available
Mem:    2818      615     1183      12     1020     1957
Swap:   3124         0     3124
```

추가 리소스

- [podman-exec 매뉴얼 페이지](#)

4.11. 두 컨테이너 간 파일 공유

컨테이너를 삭제한 경우에도 볼륨을 사용하여 컨테이너에 데이터를 유지할 수 있습니다. 볼륨은 여러 컨테이너 간에 데이터를 공유하는 데 사용할 수 있습니다. 볼륨은 호스트 시스템에 저장된 폴더입니다. 컨테이너와 호스트 간에 볼륨을 공유할 수 있습니다.

주요 이점은 다음과 같습니다.

- 컨테이너 간에 볼륨을 공유할 수 있습니다.
- 볼륨은 백업 또는 마이그레이션이 더 쉽습니다.
- 볼륨이 컨테이너 크기를 늘리지 않습니다.

사전 요구 사항

- Podman 툴이 설치되어 있습니다.

```
# yum module install -y container-tools
```

절차

1. 볼륨을 생성합니다.

```
$ podman volume create hostvolume
```

2. 볼륨에 대한 정보를 표시합니다.

```
$ podman volume inspect hostvolume
[
  {
    "name": "hostvolume",
    "labels": {},
    "mountpoint":
"/home/username/.local/share/containers/storage/volumes/hostvolume/_data",
    "driver": "local",
    "options": {},
    "scope": "local"
  }
]
```

volumes 디렉터리에 볼륨을 생성합니다. `$ mntPoint=$(podman volume inspect hostvolume --format {{.Mountpoint}})` 를 쉽게 조작하기 위해 마운트 지점 경로를 변수에 저장할 수 있습니다.

`sudo podman volume create hostvolume` 을 실행하면 마운트 지점이 `/var/lib/containers/storage/volumes/hostvolume/_data` 로 변경됩니다.

3. **mntPoint** 변수에 저장된 경로를 사용하여 디렉터리 내에 텍스트 파일을 생성합니다.

```
$ echo "Hello from host" >> $mntPoint/host.txt
```

4. **the mntPoint** 변수에 의해 정의된 디렉터리에 있는 모든 파일을 나열합니다.

```
$ ls $mntPoint/
host.txt
```

5.

myubi1 이라는 컨테이너를 실행하고 호스트의 **hostvolume** 볼륨 이름으로 정의된 디렉토리를 컨테이너의 **/containervolume1** 디렉토리에 매핑합니다.

```
$ podman run -it --name myubi1 -v hostvolume:/containervolume1
registry.access.redhat.com/ubi8/ubi /bin/bash
```

the **mntPoint** 변수(**-v \$mntPoint:/containervolume1**)에서 정의한 볼륨 경로를 사용하는 경우 **podman volume prune** 명령을 실행할 때 데이터가 손실되어 사용되지 않는 볼륨을 제거할 수 있습니다. 항상 **-v hostvolume_name:/containervolume_name** 을 사용합니다.

6.

컨테이너의 공유 볼륨에 있는 파일을 나열합니다.

```
# ls /containervolume1
host.txt
```

호스트에서 생성한 **host.txt** 파일을 확인할 수 있습니다.

7.

/containervolume1 디렉토리에 텍스트 파일을 생성합니다.

```
# echo "Hello from container 1" >> /containervolume1/container1.txt
```

8.

CTRL+p 및 **CTRL+q** 를 사용하여 컨테이너에서 분리합니다.

9.

호스트의 공유 볼륨에 있는 파일을 나열하면 다음 두 개의 파일이 표시됩니다.

```
$ ls $mntPoint
container1.rxt host.txt
```

이때 컨테이너와 호스트 간에 파일을 공유합니다. 두 컨테이너 간에 파일을 공유하려면 **myubi2** 라는 다른 컨테이너를 실행합니다.

10.

myubi2 라는 컨테이너를 실행하고 호스트의 **hostvolume** 볼륨 이름으로 정의된 디렉토리를 컨테이너의 **/containervolume2** 디렉토리에 매핑합니다.

■

```
$ podman run -it --name myubi2 -v hostvolume:/containervolume2
registry.access.redhat.com/ubi8/ubi /bin/bash
```

11.

컨테이너의 공유 볼륨에 있는 파일을 나열합니다.

```
# ls /containervolume2
container1.txt host.txt
```

myubi1 컨테이너 내부에서 생성한 **host.txt** 파일과 **container1.txt** 파일을 확인할 수 있습니다.

12.

/containervolume2 디렉터리에 텍스트 파일을 생성합니다.

```
# echo "Hello from container 2" >> /containervolume2/container2.txt
```

13.

CTRL+p 및 **CTRL+q** 를 사용하여 컨테이너에서 분리합니다.

14.

호스트의 공유 볼륨에 있는 파일을 나열하고 세 개의 파일이 표시됩니다.

```
$ ls $mntPoint
container1.rxt container2.txt host.txt
```

추가 리소스

•

podman-volume 도움말 페이지

4.12. 컨테이너 내보내기 및 가져오기

podman export 명령을 사용하여 실행 중인 컨테이너의 파일 시스템을 로컬 시스템의 **tarball**로 내보낼 수 있습니다. 예를 들어, 자주 사용하는 대형 컨테이너가 있거나 스냅샷을 저장하여 나중에 되돌리려는 대형 컨테이너가 있는 경우 **podman export** 명령을 사용하여 실행 중인 컨테이너의 현재 스냅샷을 **tarball**로 내보낼 수 있습니다.

podman import 명령을 사용하여 **tarball**을 가져와 파일 시스템 이미지로 저장할 수 있습니다. 그런 다음 이 파일 시스템 이미지를 실행하거나 다른 이미지의 계층으로 사용할 수 있습니다.

사전 요구 사항

- Podman 툴이 설치되어 있습니다.

```
# yum module install -y container-tools
```

절차

1. registry.access.redhat.com/ubi8/ubi 이미지를 기반으로 my ubi 컨테이너를 실행합니다.

```
$ podman run -dt --name=myubi registry.access.redhat.com/8/ubi
```

2. 선택 사항: 모든 컨테이너를 나열합니다.

```
$ podman ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
a6a6d4896142	registry.access.redhat.com/8:latest	/bin/bash	7 seconds ago	Up 7 seconds ago
	myubi			

3. myubi 컨테이너에 연결합니다.

```
$ podman attach myubi
```

4. testfile 이라는 파일을 만듭니다.

```
[root@a6a6d4896142 /]# echo "hello" > testfile
```

5. CTRL+p 및 CTRL+ q 를 사용하여 컨테이너에서 분리합니다.

6. 로컬 시스템에서 myubi -container.tar로 myubi-container.tar 파일 시스템을 내보냅니다.

```
$ podman export -o myubi.tar a6a6d4896142
```

7. 선택 사항: 현재 디렉터리 내용을 나열합니다.

```
$ ls -l
-rw-r--r--. 1 user user 210885120 Apr 6 10:50 myubi-container.tar
...
```

8.

선택 사항: **myubi-container** 디렉터리를 만들고 **myubi-container.tar** 아카이브에서 모든 파일을 추출합니다. 트리와 유사한 형식으로 **myubi-directory**의 내용을 나열합니다.

```
$ mkdir myubi-container
$ tar -xf myubi-container.tar -C myubi-container
$ tree -L 1 myubi-container
├── bin -> usr/bin
├── boot
├── dev
├── etc
├── home
├── lib -> usr/lib
├── lib64 -> usr/lib64
├── lost+found
├── media
├── mnt
├── opt
├── proc
├── root
├── run
├── sbin -> usr/sbin
├── srv
├── sys
├── testfile
├── tmp
├── usr
└── var
```

20 directories, 1 file

myubi-container.tar에 컨테이너 파일 시스템이 포함되어 있는 것을 확인할 수 있습니다.

9.

myubi.tar을 가져와서 파일 시스템 이미지로 저장합니다.

```
$ podman import myubi.tar myubi-imported
Getting image source signatures
Copying blob 277cab30fe96 done
Copying config c296689a17 done
Writing manifest to image destination
Storing signatures
c296689a17da2f33bf9d16071911636d7ce4d63f329741db679c3f41537e7cbf
```

10.

모든 이미지를 나열합니다.

```
$ podman images
REPOSITORY          TAG    IMAGE ID    CREATED    SIZE
docker.io/library/myubi-imported  latest c296689a17da  51 seconds ago  211 MB
```

11.

testfile 파일의 내용을 표시합니다.

```
$ podman run -it --name=myubi-imported docker.io/library/myubi-imported cat testfile
hello
```

추가 리소스

- **podman-export man** 페이지
- **podman-import man** 페이지

4.13. 컨테이너 중지

podman stop 명령을 사용하여 실행 중인 컨테이너를 중지합니다. 컨테이너 ID 또는 이름으로 컨테이너를 지정할 수 있습니다.

사전 요구 사항

- Podman 툴이 설치되어 있습니다.

```
# yum module install -y container-tools
```

- 하나 이상의 컨테이너가 실행 중입니다.

절차

- **myubi** 컨테이너를 중지합니다.

- 컨테이너 이름 사용:

```
$ podman stop myubi
```

- 컨테이너 ID 사용:

```
$ podman stop 1984555a2c27
```

터미널 세션에 연결된 실행 중인 컨테이너를 중지하려면 컨테이너 내부에 **exit** 명령을 입력하면 됩니다.

podman stop 명령은 **SIGTERM** 신호를 전송하여 실행 중인 컨테이너를 종료합니다. 정의된 기간(기본적으로 10초) 후에 컨테이너가 중지되지 않으면 Podman에서 **SIGKILL** 신호를 보냅니다.

podman kill 명령을 사용하여 컨테이너(**SIGKILL**)를 종료하거나 컨테이너에 다른 신호를 보낼 수도 있습니다. 다음은 **SIGHUP** 신호를 컨테이너로 보내는 예입니다(애플리케이션에서 지원하는 경우 **SIGHUP**으로 인해 애플리케이션이 구성 파일을 다시 읽습니다).

```
# podman kill --signal="SIGHUP" 74b1da000a11
74b1da000a114015886c557deec8bed9dfb80c888097aa83f30ca4074ff55fb2
```

추가 리소스

- [podman-stop 매뉴얼 페이지](#)
- [podman-kill man 페이지](#)

4.14. 컨테이너 제거

podman rm 명령을 사용하여 컨테이너를 제거합니다. 컨테이너 ID 또는 이름을 사용하여 컨테이너를 지정할 수 있습니다.

사전 요구 사항

- Podman 툴이 설치되어 있습니다.

```
# yum module install -y container-tools
```

- 하나 이상의 컨테이너가 중지되었습니다.

절차

1. 실행 중이거나 중지된 모든 컨테이너를 나열합니다.

```
$ podman ps -a
CONTAINER ID IMAGE      COMMAND      CREATED   STATUS    PORTS
NAMES      IS INFRA
d65aecc325a4 ubi8/ubi    /bin/bash   3 secs ago Exited (0) 5 secs ago peaceful_hopper
false
74b1da000a11 rhel8/rsyslog rsyslog.sh 2 mins ago Up About a minute
musing_brown false
```

2.

컨테이너를 제거합니다.

•

spring_hopper 컨테이너를 제거하려면 다음을 수행합니다.

```
$ podman rm peaceful_hopper
```

authenticate_hopper 컨테이너가 **Exited** 상태이므로 중지되었으며 즉시 제거할 수 있습니다.

•

musing_brown 컨테이너를 제거하려면 먼저 컨테이너를 중지한 다음 제거합니다.

```
$ podman stop musing_brown
$ podman rm musing_brown
```

참고

○

여러 컨테이너를 제거하려면 다음을 수행합니다.

```
$ podman rm clever_yonath furious_shockley
```

○

로컬 시스템에서 모든 컨테이너를 제거하려면 다음을 수행합니다.

```
$ podman rm -a
```

추가 리소스

•

podman-rm man 페이지

4.15. RUNC 컨테이너 런타임

runc 컨테이너 런타임은 **OCI(Open Container Initiative)** 컨테이너 런타임 사양의 경량 이식 가능한 구현입니다. **runc** 런타임은 **Docker**와 많은 하위 수준 코드를 공유하지만 **Docker** 플랫폼의 구성 요소에 종속되지 않습니다. **runc**는 **Linux** 네임스페이스, 실시간 마이그레이션을 지원하며 이식 가능한 성능 프로필이 있습니다.

또한 **SELinux**, 제어 그룹(**cgroups**), **seccomp** 등과 같은 **Linux** 보안 기능을 완벽하게 지원합니다. **runc**를 사용하여 이미지를 빌드하고 실행하거나 **runc**를 사용하여 **OCI** 호환 이미지를 실행할 수 있습니다.

4.16. CRUN 컨테이너 런타임

crun은 **C**로 작성된 빠른 메모리 공간 **OCI** 컨테이너 런타임입니다. **crun** 바이너리는 최대 **50**배 더 작아서 **runc** 바이너리보다 **2**배 더 빠릅니다. **crun**을 사용하면 컨테이너를 실행할 때 최소한의 프로세스 수를 설정할 수도 있습니다. **crun** 런타임은 **OCI** 후크도 지원합니다.

crun의 추가 기능은 다음과 같습니다.

- **rootless** 컨테이너에 대해 그룹별로 파일 공유
- **OCI** 후크의 **stdout** 및 **stderr** 제어
- **cgroup v2**에서 이전 버전의 **systemd** 실행
- 다른 프로그램에서 사용하는 **C** 라이브러리
- 확장성
- 이식성

추가 리소스

- [빠르고 메모리가 낮은 풋프린트 컨테이너 런타임 소개](#)

4.17. RUNC 및 CRUN을 사용하여 컨테이너 실행

runc 또는 **crun**을 사용하면 컨테이너는 번들을 사용하여 구성됩니다. 컨테이너의 번들은 **config.json**이라는 사양 파일과 루트 파일 시스템이 포함된 디렉터리입니다. 루트 파일 시스템에는 컨테이너 내용이 포함됩니다.



참고

<runtime> 은 **crun** 또는 **runc**일 수 있습니다.

절차

1. **registry.access.redhat.com/ubi8/ubi** 컨테이너 이미지를 가져옵니다.

```
# podman pull registry.access.redhat.com/ubi8/ubi
```

2. **registry.access.redhat.com/ubi8/ubi** 이미지를 **rhel.tar** 아카이브로 내보냅니다.

```
# podman export $(podman create registry.access.redhat.com/ubi8/ubi) > rhel.tar
```

3. **bundle/rootfs** 디렉토리를 생성합니다.

```
# mkdir -p bundle/rootfs
```

4. **rhel.tar** 아카이브를 **bundle/rootfs** 디렉토리로 추출합니다.

```
# tar -C bundle/rootfs -xf rhel.tar
```

5. 번들에 대해 **config.json**이라는 새 사양 파일을 생성합니다.

```
# <runtime> spec -b bundle
```

- **b** 옵션은 번들 디렉토리를 지정합니다. 기본값은 현재 디렉터리입니다.

6. 선택 사항: 설정을 변경합니다.

```
# vi bundle/config.json
```

7.

번들에 대해 **myubi** 라는 컨테이너의 인스턴스를 생성합니다.

```
# <runtime> create -b bundle/ myubi
```

8.

myubi 컨테이너를 시작합니다.

```
# <runtime> start myubi
```



참고

컨테이너 인스턴스의 이름은 호스트에 고유해야 합니다. 컨테이너의 새 인스턴스를 시작하려면 **# <runtime> start <container_name>**

검증

•

<runtime> 으로 시작된 컨테이너를 나열합니다 :

```
# <runtime> list
ID          PID    STATUS  BUNDLE    CREATED                OWNER
myubi       0      stopped /root/bundle 2021-09-14T09:52:26.659714605Z root
```

추가 리소스

•

crun 도움말 페이지

•

runc 도움말 페이지

•

[빠르고 메모리가 낮은 포트블 컨테이너 런타임 소개](#)

4.18. 임시로 컨테이너 런타임 변경

podman run 명령을 **--runtime** 옵션과 함께 사용하여 컨테이너 런타임을 변경할 수 있습니다.



참고

<runtime> 은 **crun** 또는 **runc**일 수 있습니다.

사전 요구 사항

- **Podman** 툴이 설치되어 있습니다.

```
# yum module install -y container-tools
```

절차

- **registry.access.redhat.com/ubi8/ubi** 컨테이너 이미지를 가져옵니다.

```
$ podman pull registry.access.redhat.com/ubi8/ubi
```

1. **--runtime** 옵션을 사용하여 컨테이너 런타임을 변경합니다.

```
$ podman run --name=myubi -dt --runtime=<runtime> ubi8
bashe4654eb4df12ac031f1d0f2657dc4ae6ff8eb0085bf114623b66cc664072e69b
```

2. 선택 사항: 모든 이미지를 나열합니다.

```
$ podman ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS
PORTS NAMES
e4654eb4df12 registry.access.redhat.com/ubi8:latest bash 4 seconds ago Up 4
seconds ago myubi
```

검증

- **myubi** 컨테이너에서 **OCI** 런타임이 **<runtime>** 으로 설정되어 있는지 확인합니다.

```
$ podman inspect myubi --format "{{.OCIRuntime}}"
<runtime>
```

추가 리소스

- [빠르고 메모리가 낮은 플랫폼 컨테이너 런타임 소개](#)

4.19. 컨테이너 런타임 영구 변경

`/etc/containers/containers.conf` 구성 파일에서 컨테이너 런타임 및 옵션을 **root** 사용자로 설정하거나 `$HOME/.config/containers/containers.conf` 구성 파일에서 루트가 아닌 사용자로 설정할 수 있습니다.



참고

`<runtime>` 은 **crun** 또는 **runc runtime**일 수 있습니다.

사전 요구 사항

- Podman 틀이 설치되어 있습니다.

```
# yum module install -y container-tools
```

절차

- `/etc/containers/containers.conf` 파일에서 런타임을 변경합니다.

```
# vim /etc/containers/containers.conf
[engine]
runtime = "<runtime>"
```

- **myubi**라는 컨테이너를 실행합니다.

```
# podman run --name=myubi -dt ubi8 bash
Resolved "ubi8" as an alias (/etc/containers/registries.conf.d/001-rhel-
shortnames.conf)
Trying to pull registry.access.redhat.com/ubi8:latest...
...
Storing signatures
```

검증

- **myubi** 컨테이너에서 **OCI** 런타임이 `<runtime>` 으로 설정되어 있는지 확인합니다.

```
# podman inspect myubi --format "{{.OCIRuntime}}"
<runtime>
```

추가 리소스

- [빠르고 메모리가 낮은 플랫폼 컨테이너 런타임 소개](#)
- [containers.conf 도움말 페이지](#)

4.20. 컨테이너에 대한 SELINUX 정책 생성

컨테이너에 대한 SELinux 정책을 생성하려면 UDICA 툴을 사용합니다. 자세한 내용은 [Introduction to the udica SELinux policy generator](#) 에서 참조하십시오.

5장. 노드 작업

컨테이너는 **Podman**, **Skopeo** 및 **Buildah** 컨테이너 도구로 관리할 수 있는 최소 단위입니다. **Podman** 포드는 하나 이상의 컨테이너 그룹입니다. **Pod** 개념은 **Kubernetes**에서 도입했습니다. **Podman** 포드는 **Kubernetes** 정의와 유사합니다. 포드는 **OpenShift** 또는 **Kubernetes** 환경에서 생성, 배포 및 관리할 수 있는 최소 컴퓨팅 단위입니다. 모든 **Podman** 포드에는 인프라 컨테이너가 포함됩니다. 이 컨테이너에는 포드와 연결된 네임스페이스가 있으며 **Podman**은 다른 컨테이너를 포드에 연결할 수 있습니다. 포드 내에서 컨테이너를 시작하고 중지할 수 있으며 포드가 계속 실행됩니다. registry.access.redhat.com/ubi8/pause 이미지의 기본 인프라 컨테이너입니다.

5.1. POD 생성

다음 절차에서는 하나의 컨테이너가 있는 포드를 생성하는 방법을 설명합니다.

사전 요구 사항

- **Podman** 툴이 설치되어 있습니다.

```
# yum module install -y container-tools
```

절차

1. 빈 **Pod**를 생성합니다.

```
$ podman pod create --name mypod
223df6b390b4ea87a090a4b5207f7b9b003187a6960bd37631ae9bc12c433aff
The pod is in the initial state Created.
```

포드가 **Created**(생성) 초기 상태입니다.

2. 선택 사항: 모든 **Pod**를 나열합니다.

```
$ podman pod ps
POD ID      NAME      STATUS      CREATED              # OF CONTAINERS  INFRA ID
223df6b390b4 mypod     Created     Less than a second ago 1                  3afdc93de3e
```

포드에 하나의 컨테이너가 있습니다.

3.

선택 사항: 연결된 모든 Pod 및 컨테이너를 나열합니다.

```
$ podman ps -a --pod
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS
NAMES POD
3afdc93de3e registry.access.redhat.com/ubi8/pause Less than a second ago
Created 223df6b390b4-infra 223df6b390b4
```

podman ps 명령의 포트 ID가 podman pod ps 명령의 Pod ID와 일치하는 것을 확인할 수 있습니다. 기본 인프라 컨테이너는 registry.access.redhat.com/ubi8/pause 이미지를 기반으로 합니다.

4.

my pod 라는 기존 Pod에서 myubi 라는 컨테이너를 실행합니다.

```
$ podman run -dt --name myubi --pod mypod registry.access.redhat.com/ubi8/ubi
/bin/bash
5df5c48fea87860cf75822ceab8370548b04c78be9fc156570949013863ccf71
```

5.

선택 사항: 모든 Pod를 나열합니다.

```
$ podman pod ps
POD ID NAME STATUS CREATED # OF CONTAINERS INFRA ID
223df6b390b4 mypod Running Less than a second ago 2 3afdc93de3e
```

포트에 두 개의 컨테이너가 있는 것을 확인할 수 있습니다.

6.

선택 사항: 연결된 모든 Pod 및 컨테이너를 나열합니다.

```
$ podman ps -a --pod
CONTAINER ID IMAGE COMMAND CREATED
STATUS PORTS NAMES POD
5df5c48fea87 registry.access.redhat.com/ubi8/ubi:latest /bin/bash Less than a
second ago Up Less than a second ago myubi 223df6b390b4
3afdc93de3e registry.access.redhat.com/ubi8/pause Less than a
second ago Up Less than a second ago 223df6b390b4-infra 223df6b390b4
```

추가 리소스

- podman-pod-create 도움말 페이지

-

Podman: 로컬 컨테이너 런타임에서 Pod 및 컨테이너 관리

5.2. POD 정보 표시

다음 절차에서는 **Pod** 정보를 표시하는 방법에 대해 설명합니다.

사전 요구 사항

-

Podman 툴이 설치되어 있습니다.

```
# yum module install -y container-tools
```

-

포드가 생성되었습니다. 자세한 내용은 [Pod 생성](#) 섹션을 참조하십시오.

절차

-

Pod에서 실행 중인 활성 프로세스를 표시합니다.

-

Pod에서 실행 중인 컨테이너의 프로세스를 표시하려면 다음을 입력합니다.

```
$ podman pod top mypod
USER PID PPID %CPU ELAPSED TTY TIME COMMAND
0 1 0 0.000 24.077433518s ? 0s /pause
root 1 0 0.000 24.078146025s pts/0 0s /bin/bash
```

-

하나 이상의 **Pod**에서 컨테이너에 대한 리소스 사용 통계를 실시간 표시하려면 다음을 입력합니다.

```
$ podman pod stats -a --no-stream
ID NAME CPU % MEM USAGE / LIMIT MEM % NET IO BLOCK
IO PIDS
a9f807ffaacd frosty_hodgkin -- 3.092MB / 16.7GB 0.02% -- / -- -- / -- 2
3b33001239ee sleepy_stallman -- -- / -- -- -- / -- --
```

-

Pod 설명 정보를 표시하려면 다음을 입력합니다.

```
$ podman pod inspect mypod
{
```

```

    "Id":
    "db99446fa9c6d10b973d1ce55a42a6850357e0cd447d9bac5627bb2516b5b19a",
    "Name": "mypod",
    "Created": "2020-09-08T10:35:07.536541534+02:00",
    "CreateCommand": [
        "podman",
        "pod",
        "create",
        "--name",
        "mypod"
    ],
    "State": "Running",
    "Hostname": "mypod",
    "CreateCgroup": false,
    "CgroupParent": "/libpod_parent",
    "CgroupPath":
    "/libpod_parent/db99446fa9c6d10b973d1ce55a42a6850357e0cd447d9bac5627bb25
    16b5b19a",
    "CreateInfra": false,
    "InfraContainerID":
    "891c54f70783dcad596d888040700d93f3ead01921894bc19c10b0a03c738ff7",
    "SharedNamespaces": [
        "uts",
        "ipc",
        "net"
    ],
    "NumContainers": 2,
    "Containers": [
        {
            "Id":
            "891c54f70783dcad596d888040700d93f3ead01921894bc19c10b0a03c738ff7",
            "Name": "db99446fa9c6-infra",
            "State": "running"
        },
        {
            "Id":
            "effc5bbcfef505b522e3bf8fbb5705a39f94a455a66fd81e542bcc27d39727d2d",
            "Name": "myubi",
            "State": "running"
        }
    ]
}

```

Pod에서 컨테이너에 대한 정보를 볼 수 있습니다.

추가 리소스

- [Podman 포트 상위 도움말 페이지](#)
- [podman-pod-stats man page](#)

- [podman-pod-inspect](#) 도움말 페이지

5.3. POD 중지

podman pod stop 명령을 사용하여 하나 이상의 **Pod**를 중지할 수 있습니다.

사전 요구 사항

- **Podman** 툴이 설치되어 있습니다.
- ```
yum module install -y container-tools
```
- 포드가 생성되었습니다. 자세한 내용은 [Pod 생성](#) 섹션을 참조하십시오.

#### 절차

1. 포드 **mypod** 를 중지합니다.

```
$ podman pod stop mypod
```

2. 선택 사항: 연결된 모든 **Pod** 및 컨테이너를 나열합니다.

```
$ podman ps -a --pod
```

| CONTAINER ID             | IMAGE                              | COMMAND            | CREATED            | STATUS |
|--------------------------|------------------------------------|--------------------|--------------------|--------|
| PORTS                    | NAMES                              | POD ID             | PODNAME            |        |
| 5df5c48fea87             | registry.redhat.io/ubi8/ubi:latest | /bin/bash          | About a minute ago | Exited |
| (0) 7 seconds ago        |                                    | myubi              | 223df6b390b4       | mypod  |
| 3afdc93de3e              | registry.access.redhat.com/8/pause |                    | About a minute ago |        |
| Exited (0) 7 seconds ago |                                    | 8a4e6527ac9d-infra | 223df6b390b4       | mypod  |

**mypod** 및 container **my ubi** 가 "Exited" 상태인지 확인할 수 있습니다.

#### 추가 리소스

- [podman-pod-stop man page](#)

## 5.4. POD 제거

**podman pod rm** 명령을 사용하여 중지된 **Pod** 및 컨테이너를 하나 이상 제거할 수 있습니다.

### 사전 요구 사항

- **Podman** 툴이 설치되어 있습니다.
- ```
# yum module install -y container-tools
```
- 포드가 생성되었습니다. 자세한 내용은 [Pod 생성](#) 섹션을 참조하십시오.
 - 포드가 중지되었습니다. 자세한 내용은 [Pod 중지](#) 섹션을 참조하십시오.

절차

1. **Pod mypod, type**을 제거합니다.

```
$ podman pod rm mypod
223df6b390b4ea87a090a4b5207f7b9b003187a6960bd37631ae9bc12c433aff
```

포드를 제거하면 내부의 모든 컨테이너가 자동으로 제거됩니다.

2. 선택 사항: 모든 컨테이너 및 **Pod**가 제거되었는지 확인합니다.

```
$ podman ps
$ podman pod ps
```

추가 리소스

- [podman-pod-rm](#) 도움말 페이지

6장. 컨테이너 네트워크 관리

이 장에서는 컨테이너 네트워크를 관리하는 방법에 대해 설명합니다.

6.1. 컨테이너 네트워크 나열

Podman에는 **rootless** 및 **rootful**라는 두 가지 네트워크 동작이 있습니다.

- **rootless** 네트워킹 - 네트워크가 자동으로 설정되며, 컨테이너에 **IP** 주소가 없습니다.
- **Rootful** 네트워킹 - 컨테이너에 **IP** 주소가 있습니다.

절차

- 모든 네트워크를 **root** 사용자로 나열합니다.

```
# podman network ls
NETWORK ID   NAME      VERSION   PLUGINS
2f259bab93aa podman     0.4.0     bridge,portmap,firewall,tuning
```

- 기본적으로 **Podman**은 브리지된 네트워크를 제공합니다.
- **rootless** 사용자의 네트워크 목록은 **rootful** 사용자의 것과 동일합니다.

추가 리소스

- **podman-network-ls** 매뉴얼 페이지

6.2. 네트워크 검사

podman network ls 명령으로 나열된 지정된 네트워크의 **IP** 범위, 활성화된 플러그인, 네트워크 유형 등을 표시합니다.

절차

•

기본 **podman** 네트워크를 검사합니다.

```
$ podman network inspect podman
[
  {
    "cniVersion": "0.4.0",
    "name": "podman",
    "plugins": [
      {
        "bridge": "cni-podman0",
        "hairpinMode": true,
        "ipMasq": true,
        "ipam": {
          "ranges": [
            [
              {
                "gateway": "10.88.0.1",
                "subnet": "10.88.0.0/16"
              }
            ]
          ],
          "routes": [
            {
              "dst": "0.0.0.0/0"
            }
          ],
          "type": "host-local"
        },
        "isGateway": true,
        "type": "bridge"
      },
      {
        "capabilities": {
          "portMappings": true
        },
        "type": "portmap"
      },
      {
        "type": "firewall"
      },
      {
        "type": "tuning"
      }
    ]
  }
]
```

IP 범위, 활성화된 플러그인, 네트워크 유형 및 기타 네트워크 설정을 확인할 수 있습니다.

추가 리소스

- **podman-network-inspect** 매뉴얼 페이지

6.3. 네트워크 생성

podman network create 명령을 사용하여 새 네트워크를 만듭니다.



참고

기본적으로 **Podman**은 외부 네트워크를 생성합니다. **podman network create --internal** 명령을 사용하여 내부 네트워크를 생성할 수 있습니다. 내부 네트워크의 컨테이너는 호스트의 다른 컨테이너와 통신할 수 있지만 호스트 외부의 네트워크에 연결할 수도 없습니다.

절차

- 이름이 **mynet** 인 외부 네트워크를 만듭니다.

```
# podman network create mynet
/etc/cni/net.d/mynet.conflist
```

검증

- 모든 네트워크를 나열합니다.

```
# podman network ls
NETWORK ID   NAME      VERSION  PLUGINS
2f259bab93aa podman    0.4.0    bridge,portmap,firewall,tuning
11c844f95e28 mynet     0.4.0    bridge,portmap,firewall,tuning,dnsname
```

생성된 **mynet** 네트워크 및 기본 **podman** 네트워크를 확인할 수 있습니다.



참고

Podman 4.0부터 **podman network create** 명령을 사용하여 새 외부 네트워크를 생성하는 경우 기본적으로 **DNS** 플러그인이 활성화됩니다.

추가 리소스

- [podman-network-create man 페이지](#)

6.4. 네트워크에 컨테이너 연결

podman network connect 명령을 사용하여 컨테이너를 네트워크에 연결합니다.

사전 요구 사항

- **podman network create** 명령을 사용하여 네트워크가 생성되었습니다.
- 컨테이너가 생성되었습니다.

절차

- **mycontainer** 라는 컨테이너를 **mynet** 이라는 네트워크에 연결합니다.

```
# podman network connect mynet mycontainer
```

검증

- **mycontainer** 가 **mynet** 네트워크에 연결되어 있는지 확인합니다.

```
# podman inspect --format='{{.NetworkSettings.Networks}}' mycontainer  
map[podman:0xc00042ab40 mynet:0xc00042ac60]
```

mycontainer 가 **mynet** 및 **podman** 네트워크에 연결되어 있음을 확인할 수 있습니다.

추가 리소스

- [podman-network-connect man 페이지](#)

6.5. 네트워크에서 컨테이너 연결 해제

podman network disconnect 명령을 사용하여 네트워크에서 컨테이너의 연결을 끊습니다.

사전 요구 사항

- **podman network create** 명령을 사용하여 네트워크가 생성되었습니다.
- 컨테이너가 네트워크에 연결되어 있습니다.

절차

- **mycontainer** 라는 컨테이너를 **mynet** 이라는 네트워크에서 연결을 끊습니다.

```
# podman network disconnect mynet mycontainer
```

검증

- **mycontainer** 가 **mynet** 네트워크에서 연결이 끊어졌는지 확인합니다.

```
# podman inspect --format='{{.NetworkSettings.Networks}}' mycontainer
map[podman:0xc000537440]
```

mycontainer 가 **mynet** 네트워크에서 연결이 끊어진 것을 알 수 있습니다. **mycontainer** 는 기본 **podman** 네트워크에만 연결되어 있습니다.

추가 리소스

- **podman-network-disconnect man** 페이지

6.6. 네트워크 제거

podman network rm 명령을 사용하여 지정된 네트워크를 제거합니다.

절차

1. 모든 네트워크를 나열합니다.

```
# podman network ls
NETWORK ID   NAME      VERSION  PLUGINS
2f259bab93aa podman    0.4.0    bridge,portmap,firewall,tuning
11c844f95e28 mynet     0.4.0    bridge,portmap,firewall,tuning,dnsname
```

2.

mynet 네트워크를 제거합니다.

```
# podman network rm mynet
mynet
```



참고

삭제된 네트워크에 컨테이너가 연결되어 있으면 **podman network rm -f** 명령을 사용하여 컨테이너 및 **Pod**를 삭제해야 합니다.

검증

- **mynet** 네트워크가 제거되었는지 확인합니다.

```
# podman network ls
NETWORK ID   NAME      VERSION  PLUGINS
2f259bab93aa podman    0.4.0    bridge,portmap,firewall,tuning
```

추가 리소스

- [podman-network-rm 매뉴얼 페이지](#)

6.7. 사용되지 않는 모든 네트워크 제거

podman 네트워크 정리를 사용하여 사용되지 않는 모든 네트워크를 제거합니다. 미사용 네트워크는 컨테이너가 연결되어 있지 않은 네트워크입니다. **podman network prune** 명령은 기본 **podman** 네트워크를 제거하지 않습니다.

절차

- 사용되지 않는 모든 네트워크를 제거합니다.

```
# podman network prune
WARNING! This will remove all networks not used by at least one container.
Are you sure you want to continue? [y/N] y
```

검증

- 모든 네트워크가 제거되었는지 확인합니다.

```
# podman network ls
NETWORK ID   NAME      VERSION  PLUGINS
2f259bab93aa podman    0.4.0    bridge,portmap,firewall,tuning
```

추가 리소스

- [podman-network-prune 매뉴얼 페이지](#)

7장. 컨테이너 간 통신

이 장에서는 컨테이너 간에 통신하는 방법에 대해 설명합니다.

7.1. 네트워크 모드 및 계층

Podman에는 다양한 네트워크 모드가 있습니다.

- **bridge** - 기본 브릿지 네트워크에 다른 네트워크 생성
- **container:<id> - <id> ID** 가 있는 컨테이너와 동일한 네트워크를 사용합니다.
- **host** - 호스트 네트워크 스택 사용
- **network-id** - `podman network create` 명령으로 생성된 사용자 정의 네트워크를 사용합니다.
- **Private** - 컨테이너에 대한 새 네트워크 생성
- **slirp4nets** - **rootless** 컨테이너의 기본 옵션인 **slirp4netns**를 사용하여 사용자 네트워크 스택을 만듭니다.



참고

호스트 모드는 컨테이너에 **D-bus**와 같은 로컬 시스템 서비스(예: 프로세스 간 통신)에 대한 전체 액세스 권한을 부여하므로 비보안으로 간주됩니다.

7.2. 컨테이너의 네트워크 설정 검사

`podman inspect` 명령에 `--format` 옵션을 사용하여 `podman inspect` 출력의 개별 항목을 표시합니다.

절차

1. 컨테이너의 IP 주소를 표시합니다.

```
# podman inspect --format='{{.NetworkSettings.IPAddress}}' containerName
```

2. 컨테이너가 연결된 모든 네트워크를 표시합니다.

```
# podman inspect --format='{{.NetworkSettings.Networks}}' containerName
```

3. 포트 매핑을 표시합니다.

```
# podman inspect --format='{{.NetworkSettings.Ports}}' containerName
```

추가 리소스

- [podman-inspect](#) 도움말 페이지

7.3. 컨테이너와 애플리케이션 간 통신

컨테이너와 애플리케이션 간에 통신할 수 있습니다. 애플리케이션 포트는 수신 대기 또는 열려 있는 상태입니다. 이러한 포트는 컨테이너 네트워크에 자동으로 노출되므로 이러한 네트워크를 사용하여 해당 컨테이너에 연결할 수 있습니다. 기본적으로 웹 서버는 포트 **80**에서 수신 대기합니다. 이 절차를 사용하여 **myubi** 컨테이너는 **web-container** 애플리케이션과 통신합니다.

절차

1. **web-container** 라는 컨테이너를 시작합니다.

```
# podman run -dt --name=web-container docker.io/library/httpd
```

2. 모든 컨테이너를 나열합니다.

```
# podman ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
b8c057333513	docker.io/library/httpd:latest	httpd-foreground	4 seconds ago	Up 5 seconds ago
	web-container			

3.

컨테이너를 검사하고 IP 주소를 표시합니다.

```
# podman inspect --format='{{.NetworkSettings.IPAddress}}' web-container
10.88.0.2
```

4.

myubi 컨테이너를 실행하고 웹 서버가 실행 중인지 확인합니다.

```
# podman run -it --name=myubi ubi8/ubi curl 10.88.0.2:80
<html><body><h1>It works!</h1></body></html>
```

7.4. 컨테이너와 호스트 간의 통신

기본적으로 **podman** 네트워크는 브리지 네트워크입니다. 이는 네트워크 장치가 컨테이너 네트워크를 호스트 네트워크에 브리징하고 있음을 의미합니다.

사전 요구 사항

- **web-container** 가 실행 중입니다. 자세한 내용은 [컨테이너와 애플리케이션 간 완화](#) 섹션을 참조하십시오.

절차

1.

브릿지가 구성되었는지 확인합니다.

```
# podman network inspect podman | grep bridge

"bridge": "cni-podman0",
"type": "bridge"
```

2.

호스트 네트워크 구성을 표시합니다.

```
# ip addr show cni-podman0

6: cni-podman0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default qlen 1000
    link/ether 62:af:a1:0a:ca:2e brd ff:ff:ff:ff:ff:ff
    inet 10.88.0.1/16 brd 10.88.255.255 scope global cni-podman0
        valid_lft forever preferred_lft forever
    inet6 fe80::60af:a1ff:fe0a:ca2e/64 scope link
        valid_lft forever preferred_lft forever
```

-

web-container에 **cni-podman0** 네트워크의 IP가 있고 네트워크가 호스트에 브리지된 것을 확인할 수 있습니다.

3.

web-container를 검사하고 IP 주소를 표시합니다.

```
# podman inspect --format='{{.NetworkSettings.IPAddress}}' web-container
10.88.0.2
```

4.

호스트에서 직접 **web-container**에 액세스합니다.

```
$ curl 10.88.0.2:80
<html><body><h1>It works!</h1></body></html>
```

추가 리소스

•

podman-network man 페이지

7.5. 포트 매핑을 사용하여 컨테이너 간 통신

두 컨테이너 간에 통신하는 가장 편리한 방법은 게시된 포트를 사용하는 것입니다. 포트는 자동 또는 수동이라는 두 가지 방법으로 게시할 수 있습니다.

절차

1.

게시되지 않은 컨테이너를 실행합니다.

```
# podman run -dt --name=web1 ubi8/httpd-24
```

2.

자동으로 게시된 컨테이너를 실행합니다.

```
# podman run -dt --name=web2 -P ubi8/httpd-24
```

3.

수동으로 게시된 컨테이너를 실행하고 컨테이너 포트 **80**을 게시합니다.

```
# podman run -dt --name=web3 -p 9090:80 ubi8/httpd-24
```

4.

모든 컨테이너를 나열합니다.

```
# podman ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS	NAMES	
f12fa79b8b39	registry.access.redhat.com/ubi8/httpd-24:latest	/usr/bin/run-http...	23 seconds ago
	Up 24 seconds ago	web1	
9024d9e815e2	registry.access.redhat.com/ubi8/httpd-24:latest	/usr/bin/run-http...	13 seconds ago
	Up 13 seconds ago	0.0.0.0:43595->8080/tcp, 0.0.0.0:42423->8443/tcp	
		web2	
03bc2a019f1b	registry.access.redhat.com/ubi8/httpd-24:latest	/usr/bin/run-http...	2 seconds ago
	Up 2 seconds ago	0.0.0.0:9090->80/tcp	
		web3	

다음을 확인할 수 있습니다.

- 컨테이너 웹1에는 게시된 포트가 없으며 컨테이너 네트워크 또는 브리지에서만 연결할 수 있습니다.
- 컨테이너 웹2는 각각 애플리케이션 포트 8080 및 8443을 게시하도록 포트 43595 및 42423을 자동으로 매핑했습니다.



참고

[Containerfile](#)에 registry.access.redhat.com/8/httpd-24 이미지에 EXPOSE 8080 및 EXPOSE 8443 명령이 있기 때문에 자동 포트 매핑이 가능합니다.

- 컨테이너 웹3에는 수동으로 게시된 포트가 있습니다. 호스트 포트 9090은 컨테이너 포트 80에 매핑됩니다.

5.

web1 및 web3 컨테이너의 IP 주소를 표시합니다.

```
# podman inspect --format='{{.NetworkSettings.IPAddress}}' web1
# podman inspect --format='{{.NetworkSettings.IPAddress}}' web3
```

6.

<IP>:<port> 표기법을 사용하여 **web1** 컨테이너에 연결합니다.

```
# curl 10.88.0.14:8080
...
<title>Test Page for the HTTP Server on Red Hat Enterprise Linux</title>
...
```

7.

localhost:<port> 표기법을 사용하여 **web2** 컨테이너에 연결합니다.

```
# curl localhost:43595
...
<title>Test Page for the HTTP Server on Red Hat Enterprise Linux</title>
...
```

8.

<IP>:<port> 표기법을 사용하여 **web3** 컨테이너에 도달합니다.

```
# curl 10.88.0.14:9090
...
<title>Test Page for the HTTP Server on Red Hat Enterprise Linux</title>
...
```

7.6. DNS를 사용하여 컨테이너 간 통신

DNS 플러그인이 활성화되면 컨테이너 이름을 사용하여 컨테이너 주소를 지정하십시오.

사전 요구 사항

- **podman network create** 명령을 사용하여 활성화된 **DNS** 플러그인이 있는 네트워크가 생성되었습니다.

절차

1.

mynet 네트워크에 연결된 수신자 컨테이너를 실행합니다.

```
# podman run -d --net mynet --name receiver ubi8 sleep 3000
```

2.

보낸 사람 컨테이너를 실행하고 해당 이름으로 수신자 컨테이너에 연결합니다.

```
# podman run -it --rm --net mynet --name sender alpine ping receiver
```

```
PING rcv01 (10.89.0.2): 56 data bytes
64 bytes from 10.89.0.2: seq=0 ttl=42 time=0.041 ms
64 bytes from 10.89.0.2: seq=1 ttl=42 time=0.125 ms
64 bytes from 10.89.0.2: seq=2 ttl=42 time=0.109 ms
```

CTRL+C 를 사용하여 종료합니다.

발신자 컨테이너가 해당 이름을 사용하여 수신자 컨테이너를 **ping**할 수 있음을 확인할 수 있습니다.

7.7. POD의 두 컨테이너 간 통신

동일한 **Pod**의 모든 컨테이너는 **IP** 주소, **MAC** 주소 및 포트 매핑을 공유합니다. **localhost:port** 표기법을 사용하여 동일한 포드의 컨테이너 간에 통신할 수 있습니다.

절차

1. **web-pod** 라는 **Pod**를 만듭니다.

```
$ podman pod create --name=web-pod
```

2. **Pod**에서 **web-container** 라는 웹 컨테이너를 실행합니다.

```
$ podman container run -d --pod web-pod --name=web-container
docker.io/library/httpd
```

3. 연결된 모든 **Pod** 및 컨테이너를 나열합니다.

```
$ podman ps --pod
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS	NAMES	POD ID	PODNAME	
58653cf0cf09	k8s.gcr.io/pause:3.5		4 minutes ago	Up 3 minutes ago
4e61a300c194	infra	4e61a300c194	web-pod	
b3f4255afdb3	docker.io/library/httpd:latest	httpd-foreground	3 minutes ago	Up 3 minutes ago
	web-container	4e61a300c194	web-pod	

4. **docker.io/library/fedora** 이미지를 기반으로 **web-pod** 에서 컨테이너를 실행합니다.

```
$ podman container run -it --rm --pod web-pod docker.io/library/fedora curl localhost
<html><body><h1>It works!</h1></body></html>
```

컨테이너가 **web-container**에 도달할 수 있음을 확인할 수 있습니다.

7.8. POD에서 통신

포드가 생성될 때 **Pod**에서 컨테이너 포트를 게시해야 합니다.

절차

1. **web-pod** 라는 **Pod**를 만듭니다.

```
# podman pod create --name=web-pod-publish -p 80:80
```

2. 모든 **Pod**를 나열합니다.

```
# podman pod ls
```

POD ID	NAME	STATUS	CREATED	INFRA ID	# OF CONTAINERS
26fe5de43ab3	publish-pod	Created	5 seconds ago	7de09076d2b3	1

3. **web-pod** 내에서 **web-container** 라는 웹 컨테이너를 실행합니다.

```
# podman container run -d --pod web-pod-publish --name=web-container
docker.io/library/httpd
```

4. 컨테이너 나열

```
# podman ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
7de09076d2b3	k8s.gcr.io/pause:3.5		About a minute ago	Up 23 seconds
000.0.0.0:80->80/tcp	26fe5de43ab3-infra			
088befb90e59	docker.io/library/httpd	httpd-foreground	23 seconds ago	Up 23 seconds
000.0.0.0:80->80/tcp	web-container			

5. **web-container** 에 연결할 수 있는지 확인합니다.

```
$ curl localhost:80
<html><body><h1>It works!</h1></body></html>
```

7.9. 컨테이너 네트워크에 POD 연결

Pod를 생성하는 동안 **Pod**의 컨테이너를 네트워크에 연결합니다.

절차

1. **pod-net** 이라는 네트워크를 만듭니다.

```
# podman network create pod-net
/etc/cni/net.d/pod-net.conflist
```

2. **Pod web-pod** 를 생성합니다.

```
# podman pod create --net pod-net --name web-pod
```

3. **web-pod** 내에서 **web-container** 라는 컨테이너를 실행합니다.

```
# podman run -d --pod web-pod --name=web-container docker.io/library/httpd
```

4. 선택 사항: 컨테이너가 연결된 **Pod**를 표시합니다.

```
# podman ps -p
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS	NAMES	POD ID	PODNAME	
b7d6871d018c	registry.access.redhat.com/ubi8/pause:latest		9 minutes ago	Up 6 minutes ago
645835585e24	docker.io/library/httpd:latest	httpd-foreground	6 minutes ago	Up 6 minutes ago

검증

- 컨테이너에 연결된 모든 네트워크를 표시합니다.

```
# podman ps --format="{{.Networks}}"
```

```
pod-net
```

8장. 컨테이너 네트워크 모드 설정

이 장에서는 다양한 네트워크 모드를 설정하는 방법에 대해 설명합니다.

8.1. 고정 IP로 컨테이너 실행

--ip 옵션을 사용하는 **podman run** 명령은 컨테이너 네트워크 인터페이스를 특정 IP 주소(예: 10.88.0.44)로 설정합니다. IP 주소를 올바르게 설정했는지 확인하려면 **podman inspect** 명령을 실행합니다.

절차

- 컨테이너 네트워크 인터페이스를 IP 주소 10.88.0.44로 설정합니다.

```
# podman run -d --name=myubi --ip=10.88.0.44
registry.access.redhat.com/ubi8/ubi
efde5f0a8c723f70dd5cb5dc3d5039df3b962fae65575b08662e0d5b5f9fbe85
```

검증

- IP 주소가 올바르게 설정되었는지 확인합니다.

```
# podman inspect --format='{{.NetworkSettings.IPAddress}}' myubi
10.88.0.44
```

8.2. SYSTEMD 없이 DHCP 플러그인 실행

podman run --network 명령을 사용하여 사용자 정의 네트워크에 연결합니다. 대부분의 컨테이너 이미지는 DHCP 클라이언트가 없지만 **dhcp** 플러그인은 DHCP 서버와 상호 작용할 수 있는 프록시 DHCP 클라이언트 역할을 합니다.



참고

이 절차는 **rootfull** 컨테이너에만 적용됩니다. **rootless** 컨테이너는 **dhcp** 플러그인을 사용하지 않습니다.

절차

1. **dhcp** 플러그인을 수동으로 실행합니다.

```
# /usr/libexec/cni/dhcp daemon &

[1] 4966
```

2. **dhcp** 플러그인이 실행 중인지 확인합니다.

```
# ps -a | grep dhcp

4966 pts/1    00:00:00 dhcp
```

3. **alpine** 컨테이너를 실행합니다.

```
# podman run -it --rm --network=example alpine ip addr show enp1s0

Resolved "alpine" as an alias (/etc/containers/registries.conf.d/000-shortnames.conf)
Trying to pull docker.io/library/alpine:latest...
...
Storing signatures

2: eth0@eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP
    link/ether f6:dd:1b:a7:9b:92 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.22/24 brd 192.168.1.255 scope global eth0
    ...
```

이 예제에서는 다음을 수행합니다.

- **--network=example** 옵션은 연결할 **example**이라는 네트워크를 지정합니다.
- **alpine** 컨테이너 내부의 **ip addr**는 **enp1s0** 명령을 표시합니다. 이 명령은 네트워크 인터페이스 **enp1s0**의 IP 주소를 확인합니다.
- 호스트 네트워크는 **192.168.1.0/24**
- **eth0** 인터페이스는 **alpine** 컨테이너에 대해 **192.168.1.122**의 IP 주소를 리스합니다.



참고

이 구성은 수명이 많은 컨테이너 및 긴 리스가 있는 **DHCP** 서버가 많은 경우 사용 가능한 **DHCP** 주소를 소모할 수 있습니다.

추가 리소스

- [Podman 컨테이너를 사용하여 라우팅 가능한 IP 주소 임대](#)

8.3. SYSTEMD를 사용하여 DHCP 플러그인 실행

systemd 장치 파일을 사용하여 **dhcp** 플러그인을 실행할 수 있습니다.

절차

1. 소켓 유닛 파일을 생성합니다.

```
# cat /usr/lib/systemd/system/io.podman.dhcp.socket
[Unit]
Description=DHCP Client for CNI

[Socket]
ListenStream=%t/cni/dhcp.sock
SocketMode=0600

[Install]
WantedBy=sockets.target
```

2. 서비스 유닛 파일을 생성합니다.

```
# cat /usr/lib/systemd/system/io.podman.dhcp.service
[Unit]
Description=DHCP Client CNI Service
Requires=io.podman.dhcp.socket
After=io.podman.dhcp.socket

[Service]
Type=simple
ExecStart=/usr/libexec/cni/dhcp daemon
TimeoutStopSec=30
KillMode=process
```

```
[Install]
WantedBy=multi-user.target
Also=io.podman.dhcp.socket
```

3. 즉시 서비스를 시작합니다.

```
# systemctl --now enable io.podman.dhcp.socket
```

검증

- 소켓 상태를 확인합니다.

```
# systemctl status io.podman.dhcp.socket
io.podman.dhcp.socket - DHCP Client for CNI
Loaded: loaded (/usr/lib/systemd/system/io.podman.dhcp.socket; enabled; vendor
preset: disabled)
Active: active (listening) since Mon 2022-01-03 18:08:10 CET; 39s ago
Listen: /run/cni/dhcp.sock (Stream)
CGroup: /system.slice/io.podman.dhcp.socket
```

추가 리소스

- [Podman 컨테이너를 사용하여 라우팅 가능한 IP 주소 임대](#)

8.4. MACVLAN 플러그인

대부분의 컨테이너 이미지에는 **DHCP** 클라이언트가 없으며 **dhcp** 플러그인은 **DHCP** 서버와 상호 작용하는 컨테이너의 프록시 **DHCP** 클라이언트 역할을 합니다.

호스트 시스템에는 컨테이너에 대한 네트워크 액세스 권한이 없습니다. 호스트 외부에서 컨테이너로의 네트워크 연결을 허용하려면 컨테이너와 동일한 네트워크에 있는 **IP**가 있어야 합니다. **macvlan** 플러그인을 사용하면 호스트와 동일한 네트워크에 컨테이너를 연결할 수 있습니다.



참고

이 절차는 **rootfull** 컨테이너에만 적용됩니다. **rootless** 컨테이너는 **macvlan** 및 **dhcp** 플러그인을 사용할 수 없습니다.



참고

podman network create --macvlan 명령을 사용하여 **macvlan** 네트워크를 생성할 수 있습니다.

추가 리소스

- [Podman 컨테이너를 사용하여 라우팅 가능한 IP 주소 임대](#)
- [podman-network-create man 페이지](#)

8.5. 네트워크 스택을 CNI에서 NETAVARK로 전환

이전에는 컨테이너가 단일 **CNI(Container Network Interface)** 플러그인에 연결된 경우에만 **DNS**를 사용할 수 있었습니다. **Netavark**는 컨테이너의 네트워크 스택입니다. **Netavark**를 **Podman** 및 기타 **OCI(Open Container Initiative)** 컨테이너 관리 애플리케이션과 함께 사용할 수 있습니다. **Podman**을 위한 고급 네트워크 스택은 고급 **Docker** 기능과 호환됩니다. 이제 여러 네트워크의 컨테이너가 해당 네트워크의 컨테이너에 액세스합니다.

Netavark는 다음을 수행할 수 있습니다.

- 브리지 및 **MACVLAN** 인터페이스를 포함한 네트워크 인터페이스를 생성, 관리 및 제거합니다.
- **NAT**(네트워크 주소 변환) 및 포트 매핑 규칙과 같은 방화벽 설정을 구성합니다.
- **IPv4** 및 **IPv6** 지원.
- 여러 네트워크에서 컨테이너에 대한 지원을 개선합니다.

절차

1. **/etc/containers/container.conf** 파일이 없는 경우 **/usr/share/containers.conf** 파일을 **/etc/containers/** 디렉터리에 복사합니다.

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

2.

`/etc/containers/container.conf` 파일을 편집하고 다음 내용을 **[network]** 섹션에 추가합니다.

```
network_backend="netavark"
```

3.

컨테이너 또는 **Pod**가 있는 경우 스토리지를 다시 초기 상태로 재설정합니다.

```
# podman system reset
```

4.

시스템을 재부팅합니다.

```
# reboot
```

검증

•

네트워크 스택이 **Netavark**로 변경되었는지 확인합니다.

```
# cat /etc/containers/container.conf
...
[network]
network_backend="netavark"
...
```



참고

Podman 4.0.0 이상을 사용하는 경우 **podman info** 명령을 사용하여 네트워크 스택 설정을 확인합니다.

추가 리소스

•

[podman 4.0의 새로운 네트워크 스택: 자주 묻는 질문](#)

•

[podman-system-reset man 페이지](#)

8.6. 네트워크 스택을 NETAVARK에서 CNI로 전환

절차

1. `/etc/containers/container.conf` 파일이 없는 경우 `/usr/share/containers.conf` 파일을 `/etc/containers/` 디렉터리에 복사합니다.

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

2. `/etc/containers/container.conf` 파일을 편집하고 다음 내용을 `[network]` 섹션에 추가합니다.

```
network_backend="cni"
```

3. 컨테이너 또는 Pod가 있는 경우 스토리지를 다시 초기 상태로 재설정합니다.

```
# podman system reset
```

4. 시스템을 재부팅합니다.

```
# reboot
```

검증

- 네트워크 스택이 **CNI**로 변경되었는지 확인합니다.

```
# cat /etc/containers/container.conf
...
[network]
network_backend="cni"
...
```



참고

Podman 4.0.0 이상을 사용하는 경우 `podman info` 명령을 사용하여 네트워크 스택 설정을 확인합니다.

추가 리소스

- [podman 4.0의 새로운 네트워크 스택: 자주 묻는 질문](#)
- `podman-system-reset man` 페이지

9장. PODMAN을 사용하여 OPENSIFT로 컨테이너 이식

이 장에서는 **YAML**("YAML Ain't Markup Language") 형식을 사용하여 컨테이너 및 포드에 대한 이식 가능한 설명을 생성하는 방법을 설명합니다. **YAML**은 구성 데이터를 설명하는 데 사용되는 텍스트 형식입니다.

YAML 파일은 다음과 같습니다.

- 읽기 가능.
- 손쉬운 생성.
- 환경 간에 이식 가능(예: **RHEL**과 **OpenShift** 사이의).
- 프로그래밍 언어 간 이식성.
- 를 편리하게 사용할 수 있습니다(명령줄에 모든 매개 변수를 추가할 필요가 없음).

YAML 파일을 사용하는 이유:

1. 반복적 개발에 유용할 수 있는 최소한의 입력으로 로컬 오케스트레이션된 컨테이너 및 **Pod** 세트를 다시 실행할 수 있습니다.
2. 다른 머신에서 동일한 컨테이너 및 포드를 실행할 수 있습니다. 예를 들어 **OpenShift** 환경에서 애플리케이션을 실행하고 애플리케이션이 올바르게 작동하는지 확인하려면 다음을 수행합니다. **podman generate kube** 명령을 사용하여 **Kubernetes** **YAML** 파일을 생성할 수 있습니다. 그런 다음 **podman play** 명령을 사용하여 생성된 **YAML** 파일을 **Kubernetes** 또는 **OpenShift** 환경으로 전송하기 전에 로컬 시스템에서 **Pod** 및 컨테이너 생성을 테스트할 수 있습니다. **podman play** 명령을 사용하여 **OpenShift** 또는 **Kubernetes** 환경에서 원래 생성된 포드 및 컨테이너를 재 생성할 수도 있습니다.

9.1. PODMAN을 사용하여 KUBERNETES YAML 파일 생성

다음 절차에서는 하나의 컨테이너가 있는 **Pod**를 생성하고 **podman generate kube** 명령을 사용하여

Kubernetes YAML 파일을 생성하는 방법을 설명합니다.

사전 요구 사항

- 포드가 생성되었습니다. 자세한 내용은 [Pod 생성](#) 섹션을 참조하십시오.

절차

1. 연결된 모든 Pod 및 컨테이너를 나열합니다.

```
$ podman ps -a --pod
CONTAINER ID IMAGE COMMAND CREATED
STATUS PORTS NAMES POD
5df5c48fea87 registry.access.redhat.com/ubi8/ubi:latest /bin/bash Less than a
second ago Up Less than a second ago myubi 223df6b390b4
3afdc93de3e k8s.gcr.io/pause:3.1 Less than a second ago Up
Less than a second ago 223df6b390b4-infra 223df6b390b4
```

2. Pod 이름 또는 ID를 사용하여 Kubernetes YAML 파일을 생성합니다.

```
$ podman generate kube mypod > mypod.yaml
```

`podman generate` 명령은 컨테이너에 연결될 수 있는 LVM(논리 볼륨 관리자) 논리 볼륨 또는 물리 볼륨을 반영하지 않습니다.

3. `mypod.yaml` 파일을 표시합니다.

```
$ cat mypod.yaml
# Generation of Kubernetes YAML is still under development!
#
# Save the output of this file and use kubectl create -f to import
# it into Kubernetes.
#
# Created with podman-1.6.4
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: "2020-06-09T10:31:56Z"
  labels:
    app: mypod
    name: mypod
spec:
  containers:
    - command:
```

```

- /bin/bash
env:
- name: PATH
  value: /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
- name: TERM
  value: xterm
- name: HOSTNAME
- name: container
  value: oci
image: registry.access.redhat.com/ubi8/ubi:latest
name: myubi
resources: {}
securityContext:
  allowPrivilegeEscalation: true
  capabilities: {}
  privileged: false
  readOnlyRootFilesystem: false
tty: true
workingDir: /
status: {}

```

추가 리소스

- [podman-generate-kube 매뉴얼 페이지](#)
- [Podman: 로컬 컨테이너 런타임에서 Pod 및 컨테이너 관리](#)

9.2. OPENSIFT 환경에서 KUBERNETES YAML 파일 생성

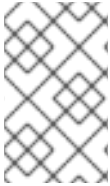
OpenShift 환경에서 `oc create` 명령을 사용하여 애플리케이션을 설명하는 **YAML** 파일을 생성합니다.

절차

- **myapp** 애플리케이션에 대한 **YAML** 파일을 생성합니다.

```
$ oc create myapp --image=me/myapp:v1 -o yaml --dry-run > myapp.yaml
```

`oc create` 명령은 **myapp** 이미지를 생성하고 실행합니다. 오브젝트는 `--dry-run` 옵션을 사용하여 출력되고 **myapp.yaml** 출력 파일로 리디렉션됩니다.



참고

Kubernetes 환경에서는 동일한 플래그로 **kubecti create** 명령을 사용할 수 있습니다.

9.3. PODMAN을 사용하여 컨테이너 및 포트 시작

생성된 **YAML** 파일을 사용하면 모든 환경에서 컨테이너 및 포트를 자동으로 시작할 수 있습니다. **YAML** 파일은 **Podman**에서 생성할 수 없습니다. **podman play kube** 명령을 사용하면 **YAML** 입력 파일을 기반으로 **Pod** 및 컨테이너를 재생성할 수 있습니다.

절차

1.

mypod.yaml 파일에서 **Pod** 및 컨테이너를 생성합니다.

```
$ podman play kube mypod.yaml
Pod:
b8c5b99ba846ccff76c3ef257e5761c2d8a5ca4d7ffa3880531aec79c0dacb22
Container:
848179395ebd33dd91d14ffbde7ae273158d9695a081468f487af4e356888ece
```

2.

모든 **Pod**를 나열합니다.

```
$ podman pod ps
POD ID      NAME      STATUS      CREATED      # OF CONTAINERS  INFRA ID
b8c5b99ba846 mypod     Running     19 seconds ago  2                aa4220eaf4bb
```

3.

연결된 모든 **Pod** 및 컨테이너를 나열합니다.

```
$ podman ps -a --pod
CONTAINER ID IMAGE                                COMMAND                  CREATED        STATUS
PORTS NAMES          POD
848179395ebd registry.access.redhat.com/ubi8/ubi:latest /bin/bash About a minute ago Up About a minute ago myubi b8c5b99ba846
aa4220eaf4bb k8s.gcr.io/pause:3.1                About a minute ago Up About a minute ago b8c5b99ba846-infra b8c5b99ba846
```

podman ps 명령의 포트 ID는 **podman pod ps** 명령의 포트 ID와 일치합니다.

추가 리소스

- **podman-play-kube** 매뉴얼 페이지
- **Podman은 Kubernetes 및 CRI-O로 쉽게 전환할 수 있습니다.**

9.4. OPENSIFT 환경에서 컨테이너 및 포트 시작

oc create 명령을 사용하여 **OpenShift** 환경에서 포트와 컨테이너를 생성할 수 있습니다.

절차

- **OpenShift** 환경의 **YAML** 파일에서 **Pod**를 생성합니다.

```
$ oc create -f mypod.yaml
```



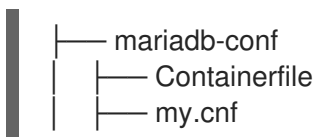
참고

Kubernetes 환경에서는 동일한 플래그로 **kubectl create** 명령을 사용할 수 있습니다.

9.5. PODMAN을 사용하여 수동으로 컨테이너 및 POD 실행

다음 절차에서는 **Podman**을 사용하여 **MariaDB** 데이터베이스와 결합된 **WordPress** 콘텐츠 관리 시스템을 수동으로 생성하는 방법을 보여줍니다.

다음 디렉터리 레이아웃을 가정합니다.



절차

1. **mariadb-conf/Containerfile** 파일을 표시합니다.

```
$ cat mariadb-conf/Containerfile
FROM docker.io/library/mariadb
COPY my.cnf /etc/mysql/my.cnf
```

2.

mariadb-conf/my.cnf 파일을 표시합니다.

```
[client-server]
# Port or socket location where to connect
port = 3306
socket = /run/mysqld/mysqld.sock

# Import all .cnf files from the configuration directory
[mariadb]
skip-host-cache
skip-name-resolve
bind-address = 127.0.0.1

!includedir /etc/mysql/mariadb.conf.d/
!includedir /etc/mysql/conf.d/
```

3.

mariadb-conf/Containerfile 을 사용하여 **docker.io/library /RuntimeConfig** 이미지를 빌드합니다.

```
$ cd mariadb-conf
$ podman build -t mariadb-conf .
$ cd ..
STEP 1: FROM docker.io/library/mariadb
Trying to pull docker.io/library/mariadb:latest...
Getting image source signatures
Copying blob 7b1a6ab2e44d done
...
Storing signatures
STEP 2: COPY my.cnf /etc/mysql/my.cnf
STEP 3: COMMIT mariadb-conf
--> ffae584aa6e
Successfully tagged localhost/mariadb-conf:latest
ffae584aa6e733ee1cdf89c053337502e1089d1620ff05680b6818a96eec3c17
```

4.

선택 사항: 모든 이미지를 나열합니다.

```
$ podman images
LIST IMAGES
REPOSITORY                                TAG      IMAGE ID      CREATED
SIZE
localhost/mariadb-conf                    latest   b66fa0fa0ef2  57 seconds ago
416 MB
```

5.

wordpresspod 라는 Pod를 생성하고 컨테이너와 호스트 시스템 간의 포트 매핑을 구성합니다.

```
$ podman pod create --name wordpresspod -p 8080:80
```

6.

wordpresspod 포트에 mydb 컨테이너를 만듭니다.

```
$ podman run --detach --pod wordpresspod \
  -e MYSQL_ROOT_PASSWORD=1234 \
  -e MYSQL_DATABASE=mywpdb \
  -e MYSQL_USER=mywpuser \
  -e MYSQL_PASSWORD=1234 \
  --name mydb localhost/mariadb-conf
```

7.

wordpresspod 포트에 myweb 컨테이너를 만듭니다.

```
$ podman run --detach --pod wordpresspod \
  -e WORDPRESS_DB_HOST=127.0.0.1 \
  -e WORDPRESS_DB_NAME=mywpdb \
  -e WORDPRESS_DB_USER=mywpuser \
  -e WORDPRESS_DB_PASSWORD=1234 \
  --name myweb docker.io/wordpress
```

8.

선택 사항: 연결된 모든 Pod 및 컨테이너를 나열합니다.

```
$ podman ps --pod -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS	NAMES	POD ID
9ea56f771915	k8s.gcr.io/pause:3.5		Less than a second ago Up
Less than a second ago	0.0.0.0:8080->80/tcp	4b7f054a6f01-infra	4b7f054a6f01
wordpresspod			
60e8dbbabac5	localhost/mariadb-conf:latest	mariadb	Less than a second ago Up
Less than a second ago	0.0.0.0:8080->80/tcp	mydb	4b7f054a6f01
wordpresspod			
045d3d506e50	docker.io/library/wordpress:latest	apache2-foregroun...	Less than a second ago Up
Less than a second ago	0.0.0.0:8080->80/tcp	myweb	
4b7f054a6f01	wordpresspod		

검증

•

Pod가 실행 중인지 확인합니다. <http://localhost:8080/wp-admin/install.php> 페이지를 방문하거나 curl 명령을 사용합니다.

```
$ curl http://localhost:8080/wp-admin/install.php
<!DOCTYPE html>
<html xml:lang="ko-KR">
<head>
...
</head>
<body class="wp-core-ui">
```

```
<p id="logo">WordPress</p>
<h1>Welcome</h1>
...
```

추가 리소스

- [Podman 플레이 kube를 사용하여 Kubernetes Pod 빌드](#)
- [podman-play-kube 매뉴얼 페이지](#)

9.6. PODMAN을 사용하여 YAML 파일 생성

`podman generate kube` 명령을 사용하여 Kubernetes YAML 파일을 생성할 수 있습니다.

사전 요구 사항

- `wordpresspod` 라는 포트가 생성되었습니다. 자세한 내용은 [Pod 생성](#) 섹션을 참조하십시오.

절차

1. 연결된 모든 Pod 및 컨테이너를 나열합니다.

```
$ podman ps --pod -a
CONTAINER ID IMAGE COMMAND CREATED
STATUS PORTS NAMES POD ID PODNAME
9ea56f771915 k8s.gcr.io/pause:3.5 Less than a second ago Up
Less than a second ago 0.0.0.0:8080->80/tcp 4b7f054a6f01-infra 4b7f054a6f01
wordpresspod
60e8dbbabac5 localhost/mariadb-conf:latest mariadb Less than a second
ago Up Less than a second ago 0.0.0.0:8080->80/tcp mydb 4b7f054a6f01
wordpresspod
045d3d506e50 docker.io/library/wordpress:latest apache2-foregroun... Less than a
second ago Up Less than a second ago 0.0.0.0:8080->80/tcp myweb
4b7f054a6f01 wordpresspod
```

2. Pod 이름 또는 ID를 사용하여 Kubernetes YAML 파일을 생성합니다.

```
$ podman generate kube wordpresspod >> wordpresspod.yaml
```

검증

- **wordpresspod.yaml** 파일을 표시합니다.

```
$ cat wordpresspod.yaml
...
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: "2021-12-09T15:09:30Z"
  labels:
    app: wordpresspod
    name: wordpresspod
spec:
  containers:
    - args:
        value: podman
      - name: MYSQL_PASSWORD
        value: "1234"
      - name: MYSQL_MAJOR
        value: "8.0"
      - name: MYSQL_VERSION
        value: 8.0.27-1debian10
      - name: MYSQL_ROOT_PASSWORD
        value: "1234"
      - name: MYSQL_DATABASE
        value: mywpdb
      - name: MYSQL_USER
        value: mywpuser
      image: mariadb
      name: mydb
      ports:
        - containerPort: 80
          hostPort: 8080
          protocol: TCP
    - args:
        - name: WORDPRESS_DB_NAME
          value: mywpdb
        - name: WORDPRESS_DB_PASSWORD
          value: "1234"
        - name: WORDPRESS_DB_HOST
          value: 127.0.0.1
        - name: WORDPRESS_DB_USER
          value: mywpuser
      image: docker.io/library/wordpress:latest
      name: myweb
```

추가 리소스

- [Podman 플레이 kube를 사용하여 Kubernetes Pod 빌드](#)
- [podman-play-kube 매뉴얼 페이지](#)

9.7. PODMAN을 사용하여 컨테이너 및 POD 자동 실행

podman play kube 명령을 사용하여 생성된 **YAML** 파일을 **Kubernetes** 또는 **OpenShift** 환경으로 전송하기 전에 로컬 시스템에서 **Pod** 및 컨테이너 생성을 테스트할 수 있습니다.

podman play kube 명령은 **docker compose** 명령과 유사하게 **YAML** 파일을 사용하여 **Pod**의 여러 컨테이너로 여러 **Pod**를 자동으로 빌드하고 실행할 수도 있습니다. 다음 조건이 충족되면 이미지가 자동으로 빌드됩니다.

1. **YAML** 파일에 사용된 이미지와 이름이 동일한 디렉터리가 있습니다.
2. 이 디렉터리에는 컨테이너 파일이 포함되어 있습니다.

사전 요구 사항

- **wordpresspod** 라는 포트가 생성되었습니다. 자세한 내용은 **Podman을 사용하여 수동으로 실행 중인 컨테이너 및 포트** 섹션을 참조하십시오.
- **YAML** 파일이 생성되었습니다. 자세한 내용은 **Podman을 사용하여 YAML 파일 생성** 섹션을 참조하십시오.
- 처음부터 전체 시나리오를 반복하려면 로컬에 저장된 이미지를 삭제합니다.

```
$ podman rmi localhost/mariadb-conf
$ podman rmi docker.io/library/wordpress
$ podman rmi docker.io/library/mysql
```

절차

1. **wordpress.yaml** 파일을 사용하여 **wordpress** 포트를 만듭니다.

```
STEP 1/2: FROM docker.io/library/mariadb
STEP 2/2: COPY my.cnf /etc/mysql/my.cnf
COMMIT localhost/mariadb-conf:latest
--> 428832c45d0
Successfully tagged localhost/mariadb-conf:latest
428832c45d07d78bb9cb34e0296a7dc205026c2fe4d636c54912c3d6bab7f399
Trying to pull docker.io/library/wordpress:latest...
Getting image source signatures
```

```
Copying blob 99c3c1c4d556 done
```

```
...
```

```
Storing signatures
```

```
Pod:
```

```
3e391d091d190756e655219a34de55583eed3ef59470aadd214c1fc48cae92ac
```

```
Containers:
```

```
6c59ebe968467d7fdb961c74a175c88cb5257fed7fb3d375c002899ea855ae1f
```

```
29717878452ff56299531f79832723d3a620a403f4a996090ea987233df0bc3d
```

podman play kube 명령은 다음과 같습니다.

- `docker.io/library/ResourceOverride` 이미지를 기반으로 `localhost/ systemd-conf:latest` 이미지를 자동으로 빌드합니다.
- `docker.io/library/wordpress:latest` 이미지를 가져옵니다.
- `wordpresspod-mysql` 및 `wordpresspod -myweb` 이라는 두 개의 컨테이너가 있는 `wordpresspod`라는 포드를 만듭니다.

2.

모든 컨테이너 및 Pod를 나열합니다.

```
$ podman ps --pod -a
CONTAINER ID IMAGE COMMAND CREATED STATUS
PORTS NAMES POD ID PODNAME
a1dbf7b5606c k8s.gcr.io/pause:3.5 3 minutes ago Up 2 minutes
ago 0.0.0.0:8080->80/tcp 3e391d091d19-infra 3e391d091d19 wordpresspod
6c59ebe96846 localhost/mariadb-conf:latest mariadb 2 minutes ago
Exited (1) 2 minutes ago 0.0.0.0:8080->80/tcp wordpresspod-mysql 3e391d091d19
wordpresspod
29717878452f docker.io/library/wordpress:latest apache2-foregroun... 2 minutes ago
Up 2 minutes ago 0.0.0.0:8080->80/tcp wordpresspod-myweb 3e391d091d19
wordpresspod
```

검증

- Pod가 실행 중인지 확인합니다. <http://localhost:8080/wp-admin/install.php> 페이지를 방문하거나 `curl` 명령을 사용합니다.

```
$ curl http://localhost:8080/wp-admin/install.php
<!DOCTYPE html>
<html xml:lang="ko-KR">
<head>
...
```

```
</head>
<body class="wp-core-ui">
<p id="logo">WordPress</p>
  <h1>Welcome</h1>
...
```

추가 리소스

- [Podman 플레이 kube를 사용하여 Kubernetes Pod 빌드](#)
- [podman-play-kube 매뉴얼 페이지](#)

9.8. PODMAN을 사용하여 POD 자동 중지 및 제거

`podman play kube --down` 명령은 모든 Pod 및 해당 컨테이너를 중지하고 제거합니다.



참고

블롭이 사용 중인 경우 제거되지 않습니다.

사전 요구 사항

- **wordpresspod** 라는 포트가 생성되었습니다. 자세한 내용은 [Podman을 사용하여 수동으로 실행 중인 컨테이너 및 포트](#) 섹션을 참조하십시오.
- **YAML** 파일이 생성되었습니다. 자세한 내용은 [Podman을 사용하여 YAML 파일 생성](#) 섹션을 참조하십시오.
- **Pod**가 실행 중입니다. 자세한 내용은 [Podman을 사용하여 컨테이너 및 포트 자동 실행](#) 섹션을 참조하십시오.

절차

- **wordpresspod.yaml** 파일에서 생성한 모든 포트 및 컨테이너를 제거합니다.

```
$ podman play kube --down wordpresspod.yaml
Pods stopped:
3e391d091d190756e655219a34de55583eed3ef59470aadd214c1fc48cae92ac
```

Pods removed:

3e391d091d190756e655219a34de55583eed3ef59470aadd214c1fc48cae92ac

검증

- **wordpresspod.yaml** 파일에서 생성한 모든 포트 및 컨테이너가 제거되었는지 확인합니다.

\$ podman ps --pod -a

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS	NAMES	POD ID PODNAME

추가 리소스

- [Podman 플레이 kube를 사용하여 Kubernetes Pod 빌드](#)
- [podman-play-kube 매뉴얼 페이지](#)

10장. PODMAN을 사용하여 컨테이너를 SYSTEMD로 포팅

Podman(Pod Manager)은 데몬이 없는 간단한 툴인 완전히 지원되는 컨테이너 엔진입니다. **Podman**은 다른 컨테이너 엔진에서 쉽게 전환하고 포트, 컨테이너 및 이미지를 관리할 수 있는 **Docker-CLI** 비교 명령줄을 제공합니다.

Podman은 원래 전체 **Linux** 시스템을 가져오거나 시작 순서, 종속성 확인, 실패한 서비스 복구 등의 서비스를 관리하도록 설계되지 않았습니다. **systemd**와 같은 전체 초기화 시스템의 작업입니다. **Red Hat**은 컨테이너를 **systemd**와 통합하는 리더가 되어 **Podman**에서 빌드한 **OCI** 및 **Docker** 형식 컨테이너를 **Linux** 시스템에서 다른 서비스 및 기능을 관리하는 것과 동일한 방식으로 관리할 수 있습니다. **systemd** 초기화 서비스를 사용하여 **Pod** 및 컨테이너와 작업할 수 있습니다. **podman generate systemd** 명령을 사용하여 컨테이너 및 포트에 대한 **systemd** 장치 파일을 생성할 수 있습니다.

systemd 유닛 파일을 사용하면 다음을 수행할 수 있습니다.

- **systemd** 서비스로 시작하도록 컨테이너 또는 포드를 설정합니다.
- 컨테이너화된 서비스가 실행되는 순서를 정의하고 종속성을 확인합니다(예: 다른 서비스가 실행 중인지 확인, 파일을 사용할 수 있는지 또는 리소스가 마운트되었는지 확인).
- **systemctl** 명령을 사용하여 **systemd** 시스템의 상태를 제어합니다.

이 장에서는 **systemd** 유닛 파일을 사용하여 컨테이너 및 포트에 대한 이식 가능한 설명을 생성하는 방법에 대해 설명합니다.

10.1. SYSTEMD 서비스 활성화

서비스를 활성화할 때 다양한 옵션이 있습니다.

절차

- 서비스를 활성화합니다.
 - 사용자가 로그인 여부에 관계없이 시스템을 시작할 때 서비스를 활성화하려면 다음을 입력합니다.

```
# systemctl enable <service>
```

systemd 유닛 파일을 **/etc/systemd/system** 디렉토리에 복사해야 합니다.

- 사용자 로그인 시 서비스를 시작하고 사용자가 로그아웃할 때 중지하려면 다음을 입력합니다.

```
$ systemctl --user enable <service>
```

systemd 유닛 파일을 **\$HOME/.config/systemd/user** 디렉토리에 복사해야 합니다.

- 사용자가 시스템 시작 시 서비스를 시작하고 로그아웃 후에도 지속할 수 있도록 하려면 다음을 입력합니다.

```
# loginctl enable-linger <username>
```

추가 리소스

- **systemctl man** 페이지
- **loginctl** 매뉴얼 페이지
- 부팅 시 **systemd** 서비스가 시작되도록 설정

10.2. PODMAN을 사용하여 SYSTEMD 장치 파일 생성

Podman을 사용하면 **systemd**에서 컨테이너 프로세스를 제어하고 관리할 수 있습니다. **podman generate systemd** 명령을 사용하여 기존 컨테이너 및 **Pod**에 대한 **systemd** 장치 파일을 생성할 수 있습니다. 생성된 유닛 파일이 **Podman**에 업데이트되어 **podman generate systemd**를 통해 최신 버전의 유닛 파일을 가져올 수 있으므로 **podman generate systemd**를 사용하는 것이 좋습니다.

절차

1. 컨테이너를 만듭니다(예: **myubi**).

```
$ podman create --name myubi registry.access.redhat.com/ubi8:latest sleep infinity
0280afe98bb75a5c5e713b28de4b7c5cb49f156f1cce4a208f13fee2f75cb453
```

2.

컨테이너 이름 또는 ID를 사용하여 **systemd** 유닛 파일을 생성하고
`~/.config/systemd/user/container-myubi.service` 파일로 보냅니다.

```
$ podman generate systemd --name myubi > ~/.config/systemd/user/container-
myubi.service
```

검증 단계

•

생성된 **systemd** 장치 파일의 내용을 표시합니다.

```
$ cat ~/.config/systemd/user/container-myubi.service
# container-myubi.service
# autogenerated by Podman 3.3.1
# Wed Sep 8 20:34:46 CEST 2021

[Unit]
Description=Podman container-myubi.service
Documentation=man:podman-generate-systemd(1)
Wants=network-online.target
After=network-online.target
RequiresMountsFor=/run/user/1000/containers

[Service]
Environment=PODMAN_SYSTEMD_UNIT=%n
Restart=on-failure
TimeoutStopSec=70
ExecStart=/usr/bin/podman start myubi
ExecStop=/usr/bin/podman stop -t 10 myubi
ExecStopPost=/usr/bin/podman stop -t 10 myubi
PIDFile=/run/user/1000/containers/overlay-
containers/9683103f58a32192c84801f0be93446cb33c1ee7d9cdda225b78049d7c5deea4/
userdata/common.pid
Type=forking

[Install]
WantedBy=multi-user.target default.target
```

○

Restart=on-failure 행은 재시작 정책을 설정하고 서비스를 정상적으로 시작하거나 중지할 수 없을 때 또는 프로세스가 0이 아닌 종료될 때 **systemd**를 다시 시작하도록 지시합니다.

○

ExecStart 행은 컨테이너를 시작하는 방법을 설명합니다.

○

ExecStop 행은 컨테이너를 중지하고 제거하는 방법을 설명합니다.

추가 리소스

●

[Podman을 사용하여 컨테이너 실행 및 공유 가능한 systemd 서비스](#)

10.3. PODMAN을 사용하여 SYSTEMD 장치 파일 자동 생성

기본적으로 **Podman**은 기존 컨테이너 또는 포트에 대한 유닛 파일을 생성합니다. **podman generate systemd --new**를 사용하여 이식 가능한 **systemd** 장치 파일을 생성할 수 있습니다. **new** 플래그는 컨테이너를 생성, 시작 및 제거하는 유닛 파일을 생성하도록 **Podman**에 지시합니다.

절차

1.

시스템에서 사용할 이미지를 가져옵니다. 예를 들어 **httpd-24** 이미지를 가져오려면 다음을 수행합니다.

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

2.

선택 사항: 시스템에서 사용 가능한 모든 이미지를 나열합니다.

```
# podman images
REPOSITORY          TAG          IMAGE ID      CREATED      SIZE
registry.access.redhat.com/ubi8/httpd-24 latest      8594be0a0b57 2 weeks ago
462 MB
```

3.

httpd 컨테이너를 생성합니다.

```
# podman create --name httpd -p 8080:8080 registry.access.redhat.com/ubi8/httpd-24
cdb9f981cf143021b1679599d860026b13a77187f75e46cc0eac85293710a4b1
```

4.

선택 사항: 컨테이너가 생성되었는지 확인합니다.

```
# podman ps -a
CONTAINER ID  IMAGE                                COMMAND                                CREATED
STATUS       PORTS                                NAMES
cdb9f981cf14 registry.access.redhat.com/ubi8/httpd-24:latest /usr/bin/run-http... 5
minutes ago  Created  0.0.0.0:8080->8080/tcp httpd
```

5.

httpd 컨테이너의 **systemd** 장치 파일을 생성합니다.

```
# podman generate systemd --new --files --name httpd
/root/container-httpd.service
```

6.

생성된 **container-httpd.service** **systemd** 장치 파일의 내용을 표시합니다.

```
# cat /root/container-httpd.service
# container-httpd.service
# autogenerated by Podman 3.3.1
# Wed Sep 8 20:41:44 CEST 2021

[Unit]
Description=Podman container-httpd.service
Documentation=man:podman-generate-systemd(1)
Wants=network-online.target
After=network-online.target
RequiresMountsFor=%t/containers

[Service]
Environment=PODMAN_SYSTEMD_UNIT=%n
Restart=on-failure
TimeoutStopSec=70
ExecStartPre=/bin/rm -f %t/%n.ctr-id
ExecStart=/usr/bin/podman run --cidfile=%t/%n.ctr-id --sdnotify=common --
cgroups=no-common --rm -d --replace --name httpd -p 8080:8080
registry.access.redhat.com/ubi8/httpd-24
ExecStop=/usr/bin/podman stop --ignore --cidfile=%t/%n.ctr-id
ExecStopPost=/usr/bin/podman rm -f --ignore --cidfile=%t/%n.ctr-id
Type=notify
NotifyAccess=all

[Install]
WantedBy=multi-user.target default.target
```

참고

new 옵션을 사용하여 생성한 유닛 파일은 컨테이너와 포드가 존재할 것으로 예상하지 않습니다. 따라서 **podman start** 명령 대신 서비스를 시작할 때 **podman run** 명령을 수행합니다(**ExecStart** 행 참조). 예를 들어 [Podman을 사용하여 systemd 장치 파일 생성](#) 섹션을 참조하십시오.

-

podman run 명령은 다음 명령줄 옵션을 사용합니다.

-

con mon-pidfile 옵션은 호스트에서 실행 중인 **common** 프로세스의 프로세스 ID를 저장하는 경로를 가리킵니다. **common** 프로세스는 컨테이너와 동일한 종료

상태로 종료되므로 **systemd**에서 올바른 서비스 상태를 보고하고 필요한 경우 컨테이너를 다시 시작할 수 있습니다.

- **cidfile** 옵션은 컨테이너 **ID**를 저장하는 경로를 가리킵니다.
- **%t** 는 런타임 디렉토리 루트의 경로입니다(예: **/run/user/\$UserID**).
- **%n** 은 서비스의 전체 이름입니다.

7.

루트 사용자로 설치하기 위해 장치 파일을 **/usr/lib/systemd/system** 에 복사합니다.

```
# cp -Z container-httpd.service /etc/systemd/system
```

8.

container-httpd.service 를 활성화하고 시작합니다 :

```
# systemctl daemon-reload
# systemctl enable --now container-httpd.service
Created symlink /etc/systemd/system/multi-user.target.wants/container-httpd.service
→ /etc/systemd/system/container-httpd.service.
Created symlink /etc/systemd/system/default.target.wants/container-httpd.service →
/etc/systemd/system/container-httpd.service.
```

검증 단계

- **container-httpd.service** :의 상태를 확인합니다.

```
# systemctl status container-httpd.service
● container-httpd.service - Podman container-httpd.service
   Loaded: loaded (/etc/systemd/system/container-httpd.service; enabled; vendor
  preset: disabled)
   Active: active (running) since Tue 2021-08-24 09:53:40 EDT; 1min 5s ago
     Docs: man:podman-generate-systemd(1)
   Process: 493317 ExecStart=/usr/bin/podman run --common-pidfile /run/container-
  httpd.pid --cidfile /run/container-httpd.ctr-id --cgroups=no-common -d --repla>
   Process: 493315 ExecStartPre=/bin/rm -f /run/container-httpd.pid /run/container-
  httpd.ctr-id (code=exited, status=0/SUCCESS)
    Main PID: 493435 (common)
  ...
```

추가 리소스

- **Podman 2.0으로 Systemd 통합 개선**
- **부팅 시 systemd 서비스가 시작되도록 설정**

10.4. SYSTEMD를 사용하여 컨테이너 자동 시작

systemctl 명령을 사용하여 **systemd** 시스템 및 서비스 관리자의 상태를 제어할 수 있습니다. 이 섹션에서는 루트가 아닌 사용자로 서비스를 활성화, 시작, 중지하는 방법에 대한 일반적인 절차를 보여줍니다. 서비스를 루트 사용자로 설치하려면 **--user** 옵션을 생략합니다.

절차

1. **systemd** 관리자 설정을 다시 로드합니다.

```
# systemctl --user daemon-reload
```

2. 서비스 **container.service** 를 활성화하고 부팅 시 시작합니다.

```
# systemctl --user enable container.service
```

3. 즉시 서비스를 시작합니다.

```
# systemctl --user start container.service
```

4. 서비스의 상태를 확인합니다.

```
$ systemctl --user status container.service
● container.service - Podman container.service
   Loaded: loaded (/home/user/.config/systemd/user/container.service; enabled; vendor preset: enabled)
   Active: active (running) since Wed 2020-09-16 11:56:57 CEST; 8s ago
     Docs: man:podman-generate-systemd(1)
   Process: 80602 ExecStart=/usr/bin/podman run --common-pidfile //run/user/1000/container.service-pid --cidfile //run/user/1000/container.service-cid -d ubi8-minimal:>
   Process: 80601 ExecStartPre=/usr/bin/rm -f //run/user/1000/container.service-pid //run/user/1000/container.service-cid (code=exited, status=0/SUCCESS)
  Main PID: 80617 (common)
    CGroup: /user.slice/user-1000.slice/user@1000.service/container.service
            └─ 2870 /usr/bin/podman
```

```

└─80612 /usr/bin/slrp4netns --disable-host-loopback --mtu 65520 --enable-
sandbox --enable-seccomp -c -e 3 -r 4 --netns-type=path /run/user/1000/netns/cni->
└─80614 /usr/bin/fuse-overlayfs -o
lowerdir=/home/user/.local/share/containers/storage/overlay/l/YJSPGXM2OCDZPLMLX
JOW3NRF6Q:/home/user/.local/share/contain>
└─80617 /usr/bin/conmon --api-version 1 -c
cbc75d6031508dfd3d78a74a03e4ace1732b51223e72a2ce4aa3bfe10a78e4fa -u
cbc75d6031508dfd3d78a74a03e4ace1732b51223e72>
└─cbc75d6031508dfd3d78a74a03e4ace1732b51223e72a2ce4aa3bfe10a78e4fa
└─80626 /usr/bin/coreutils --coreutils-prog-shebang=sleep /usr/bin/sleep 1d

```

systemctl is-enabled container.service 명령을 사용하여 서비스가 활성화되어 있는지 확인할 수 있습니다.

검증 단계

- 실행 중이거나 종료된 컨테이너를 나열합니다.

```

# podman ps
CONTAINER ID  IMAGE                                COMMAND CREATED     STATUS
PORTS NAMES
f20988d59920  registry.access.redhat.com/ubi8-minimal:latest top    12 seconds ago
Up 11 seconds ago    funny_zhukovsky

```

참고

container.service 를 중지하려면 다음을 입력합니다.

```
# systemctl --user stop container.service
```

추가 리소스

- [systemctl man 페이지](#)
- [Podman을 사용하여 컨테이너 실행 및 공유 가능한 systemd 서비스](#)
- [systemd를 사용하여 서비스 관리](#)

10.5. SYSTEMD를 사용하여 POD 자동 시작

여러 컨테이너를 **systemd** 서비스로 시작할 수 있습니다. **systemctl** 명령은 포트에서만 사용해야 하며 내부 **infra-container**와 함께 포트 서비스에서 관리하므로 **systemctl** 을 통해 개별적으로 컨테이너를 시작하거나 중지해서는 안 됩니다.

절차

- 빈 Pod (예: **systemd-pod**)를 생성합니다.

```
$ podman pod create --name systemd-pod
11d4646ba41b1ffa51c108cbdf97cfab3213f7bd9b3e1ca52fe81b90fed5577
```

- 선택 사항: 모든 Pod를 나열합니다.

```
$ podman pod ps
POD ID      NAME          STATUS  CREATED      # OF CONTAINERS  INFRA ID
11d4646ba41b systemd-pod  Created 40 seconds ago 1              8a428b257111
11d4646ba41b1ffa51c108cbdf97cfab3213f7bd9b3e1ca52fe81b90fed5577
```

- 빈 포트에 두 개의 컨테이너를 만듭니다. 예를 들어 **systemd-pod** 에서 **container0** 및 **container1** 을 생성하려면 다음을 수행합니다.

```
$ podman create --pod systemd-pod --name container0
registry.access.redhat.com/ubi8 top
$ podman create --pod systemd-pod --name container1
registry.access.redhat.com/ubi8 top
```

- 선택 사항: 연결된 모든 Pod 및 컨테이너를 나열합니다.

```
$ podman ps -a --pod
CONTAINER ID IMAGE                                COMMAND CREATED    STATUS
PORTS NAMES          POD ID    PODNAME
24666f47d9b2 registry.access.redhat.com/ubi8:latest top    3 minutes ago Created
container0    3130f724e229 systemd-pod
56eb1bf0cdfe k8s.gcr.io/pause:3.2                4 minutes ago Created
3130f724e229-infra 3130f724e229 systemd-pod
62118d170e43 registry.access.redhat.com/ubi8:latest top    3 seconds ago Created
container1    3130f724e229 systemd-pod
```

- 새 Pod에 대한 **systemd** 장치 파일을 생성합니다.

```
$ podman generate systemd --files --name systemd-pod
/home/user1/pod-systemd-pod.service
/home/user1/container-container0.service
```

```
/home/user1/container-container1.service
```

systemd -pod 포드용의 **systemd** 유닛 파일인 **systemd** 유닛 파일 3개가 생성되고 컨테이너 **container0** 및 **container1**에 대해 두 개의 **systemd** 유닛 파일이 생성됩니다.

6.

pod-systemd-pod.service 유닛 파일을 표시합니다.

```
$ cat pod-systemd-pod.service
# pod-systemd-pod.service
# autogenerated by Podman 3.3.1
# Wed Sep 8 20:49:17 CEST 2021

[Unit]
Description=Podman pod-systemd-pod.service
Documentation=man:podman-generate-systemd(1)
Wants=network-online.target
After=network-online.target
RequiresMountsFor=
Requires=container-container0.service container-container1.service
Before=container-container0.service container-container1.service

[Service]
Environment=PODMAN_SYSTEMD_UNIT=%n
Restart=on-failure
TimeoutStopSec=70
ExecStart=/usr/bin/podman start bcb128965b8e-infra
ExecStop=/usr/bin/podman stop -t 10 bcb128965b8e-infra
ExecStopPost=/usr/bin/podman stop -t 10 bcb128965b8e-infra
PIDFile=/run/user/1000/containers/overlay-
containers/1dfdcd20e35043939ea3f80f002c65c00d560e47223685dbc3230e26fe001b29/u
serdata/common.pid
Type=forking

[Install]
WantedBy=multi-user.target default.target
```

•

[Unit] 섹션의 **Requires** 행은 **container -container0.service** 및 **container-container1.service** 유닛 파일에 대한 종속성을 정의합니다. 두 유닛 파일이 모두 활성화됩니다.

•

[Service] 섹션의 **ExecStart** 및 **ExecStop** 행은 각각 **infra-container**를 시작하고 중지합니다.

7.

container-container0.service 유닛 파일을 표시합니다.

```
$ cat container-container0.service
```

```
# container-container0.service
# autogenerated by Podman 3.3.1
# Wed Sep  8 20:49:17 CEST 2021

[Unit]
Description=Podman container-container0.service
Documentation=man:podman-generate-systemd(1)
Wants=network-online.target
After=network-online.target
RequiresMountsFor=/run/user/1000/containers
BindsTo=pod-systemd-pod.service
After=pod-systemd-pod.service

[Service]
Environment=PODMAN_SYSTEMD_UNIT=%n
Restart=on-failure
TimeoutStopSec=70
ExecStart=/usr/bin/podman start container0
ExecStop=/usr/bin/podman stop -t 10 container0
ExecStopPost=/usr/bin/podman stop -t 10 container0
PIDFile=/run/user/1000/containers/overlay-
containers/4bccd7c8616ae5909b05317df4066fa90a64a067375af5996fdef9152f6d51f5/us
erdata/common.pid
Type=forking

[Install]
WantedBy=multi-user.target default.target
```

- [Unit] 섹션의 **BindsTo** 행은 **pod-systemd-pod.service** 유닛 파일에 대한 종속성을 정의합니다.
- [Service] 섹션의 **ExecStart** 및 **ExecStop** 행은 각각 **container0** 을 시작하고 중지합니다.

8. **container-container1.service** 유닛 파일을 표시합니다.

```
$ cat container-container1.service
```

9. 루트가 아닌 사용자로 설치하기 위해 생성된 모든 파일을 **\$HOME/.config/systemd/user** 에 복사합니다.

```
$ cp pod-systemd-pod.service container-container0.service container-
container1.service $HOME/.config/systemd/user
```

10. 서비스를 활성화하고 사용자가 로그인할 때 시작합니다.

```
$ systemctl enable --user pod-systemd-pod.service
Created symlink /home/user1/.config/systemd/user/multi-user.target.wants/pod-systemd-pod.service → /home/user1/.config/systemd/user/pod-systemd-pod.service.
Created symlink /home/user1/.config/systemd/user/default.target.wants/pod-systemd-pod.service → /home/user1/.config/systemd/user/pod-systemd-pod.service.
```

서비스는 사용자 로그아웃 시 중지됩니다.

검증 단계

- 서비스가 활성화되었는지 확인합니다.

```
$ systemctl is-enabled pod-systemd-pod.service
enabled
```

추가 리소스

- [podman-create man 페이지](#)
- [podman-generate-systemd 매뉴얼 페이지](#)
- [systemctl man 페이지](#)
- [Podman을 사용하여 컨테이너 실행 및 공유 가능한 systemd 서비스](#)
- [부팅 시 systemd 서비스가 시작되도록 설정](#)

10.6. PODMAN을 사용하여 컨테이너 자동 업데이트

podman auto-update 명령을 사용하면 자동 업데이트 정책에 따라 컨테이너를 자동으로 업데이트할 수 있습니다. **podman auto-update** 명령은 레지스트리에서 컨테이너 이미지가 업데이트되면 서비스를 업데이트합니다. 자동 업데이트를 사용하려면 **--label "io.containers.autoupdate=image"** 레이블로 컨테이너를 만들고 **podman generate systemd --new** 명령으로 생성된 **systemd** 장치에서 실행해야 합니다.

Podman은 "io.containers.autoupdate" 레이블이 "image" 로 설정된 컨테이너를 검색하고 컨테이너 레지스트리에 통신합니다. 이미지가 변경되면 Podman은 해당 **systemd** 장치를 다시 시작하여 이전 컨테

이너를 중지하고 새 이미지를 사용하여 새 컨테이너를 만듭니다. 결과적으로 컨테이너, 환경 및 모든 종속성이 다시 시작됩니다.

사전 요구 사항

- **container-tools** 모듈이 설치되어 있어야 합니다.

```
# yum module install -y container-tools
```

절차

1. **registry.access.redhat.com/ubi8/ubi-init** 이미지를 기반으로 **my ubi** 컨테이너를 시작합니다.

```
# podman run --label "io.containers.autoupdate=image" \
--name myubi -dt registry.access.redhat.com/ubi8/ubi-init top
bc219740a210455fa27deacc96d50a9e20516492f1417507c13ce1533dbdc9d
```

2. 선택 사항: 실행 중이거나 종료된 컨테이너를 나열합니다.

```
# podman ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
76465a5e2933	registry.access.redhat.com/8/ubi-init:latest	top	24 seconds ago	Up
	23 seconds ago	myubi		

3. **myubi** 컨테이너에 대한 **systemd** 장치 파일을 생성합니다.

```
# podman generate systemd --new --files --name myubi
/root/container-myubi.service
```

4. **root** 사용자로 설치하기 위해 장치 파일을 **/usr/lib/systemd/system** 에 복사합니다.

```
# cp -Z ~/container-myubi.service /usr/lib/systemd/system
```

5. **systemd** 관리자 설정을 다시 로드합니다.

```
# systemctl daemon-reload
```

6.

컨테이너 상태를 시작하고 확인합니다.

```
# systemctl start container-myubi.service
# systemctl status container-myubi.service
```

7.

컨테이너를 자동 업데이트합니다.

```
# podman auto-update
```

추가 리소스

- [Podman 2.0으로 Systemd 통합 개선](#)
- [Podman을 사용하여 컨테이너 실행 및 공유 가능한 systemd 서비스](#)
- [부팅 시 systemd 서비스가 시작되도록 설정](#)

10.7. SYSTEMD를 사용하여 컨테이너 자동 업데이트

[Podman을 사용하여 컨테이너 자동 확장](#) 섹션에서 언급한 것처럼 `podman auto-update` 명령을 사용하여 컨테이너를 업데이트할 수 있습니다. 사용자 지정 스크립트에 통합되며 필요한 경우 호출할 수 있습니다. 컨테이너를 자동 업데이트하는 또 다른 방법은 사전 설치된 `podman-auto-update.timer` 및 `podman-auto-update.service` systemd 서비스를 사용하는 것입니다. `podman-auto-update.timer` 은 특정 날짜 또는 시간에 자동 업데이트를 트리거하도록 구성할 수 있습니다. `podman-auto-update.service` 는 `systemctl` 명령으로 추가 시작하거나 다른 systemd 서비스에서 종속성으로 사용할 수 있습니다. 따라서 개별 요구 및 사용 사례를 충족하기 위해 다양한 방법으로 시간 및 이벤트를 기반으로 자동 업데이트를 트리거할 수 있습니다.

사전 요구 사항

- `container-tools` 모듈이 설치되어 있어야 합니다.

```
# yum module install -y container-tools
```

절차

1.

`podman-auto-update.service` 유닛 파일을 표시합니다.

```
# cat /usr/lib/systemd/system/podman-auto-update.service
```

```
[Unit]
Description=Podman auto-update service
Documentation=man:podman-auto-update(1)
Wants=network.target
After=network-online.target
```

```
[Service]
Type=oneshot
ExecStart=/usr/bin/podman auto-update
```

```
[Install]
WantedBy=multi-user.target default.target
```

2.

podman-auto-update.timer 유닛 파일을 표시합니다.

```
# cat /usr/lib/systemd/system/podman-auto-update.timer
```

```
[Unit]
Description=Podman auto-update timer
```

```
[Timer]
OnCalendar=daily
Persistent=true
```

```
[Install]
WantedBy=timers.target
```

이 예에서는 **podman auto-update** 명령이 매일 자정에 시작됩니다.

3.

시스템을 시작할 때 **podman-auto-update.timer** 서비스를 활성화합니다.

```
# systemctl enable podman-auto-update.timer
```

4.

systemd 서비스를 시작합니다.

```
# systemctl start podman-auto-update.timer
```

5.

선택 사항: 모든 타이머를 나열합니다.

```
# systemctl list-timers --all
NEXT          LEFT    LAST          PASSED    UNIT
ACTIVATES
```

Wed 2020-12-09 00:00:00 CET 9h left n/a
update.timer podman-auto-update.service

n/a

podman-auto-

podman-auto-update.timer에서 **podman-auto-update.service** 를 활성화하는 것을 확인할 수 있습니다.

추가 리소스

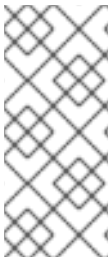
- [Podman 2.0으로 Systemd 통합 개선](#)
- [Podman을 사용하여 컨테이너 실행 및 공유 가능한 systemd 서비스](#)
- [부팅 시 systemd 서비스가 시작되도록 설정](#)

11장. UBI 컨테이너에 소프트웨어 추가

Red Hat UBI(Universal Base Images)는 **RHEL** 콘텐츠의 하위 집합에서 빌드됩니다. **UBI**는 또한 **UBI**와 함께 사용하도록 자유롭게 설치할 수 있는 **RHEL** 패키지의 하위 집합을 제공합니다. 실행 중인 컨테이너에 소프트웨어를 추가하거나 업데이트하려면 **RPM** 패키지 및 업데이트를 포함하는 **yum** 리포지토리를 사용할 수 있습니다. **UBI**는 **Python**, **Perl**, **Node.js**, **Ruby** 등과 같은 사전 빌드된 언어 런타임 컨테이너 이미지 세트를 제공합니다.

UBI 리포지토리에서 실행 중인 **UBI** 컨테이너에 패키지를 추가하려면 다음을 수행합니다.

- **UBI init** 및 **UBI** 표준 이미지에서 **yum** 명령을 사용합니다.
- **UBI** 최소 이미지에서 **microdnf** 명령을 사용하십시오.



참고

실행 중인 컨테이너에서 직접 소프트웨어 패키지를 설치하고 작업하면 패키지가 일시적으로 추가됩니다. 변경 사항은 컨테이너 이미지에 저장되지 않습니다. 패키지를 영구적으로 변경하려면 [Buildah를 사용하여 컨테이너 파일에서 이미지 빌드](#) 섹션을 참조하십시오.



참고

UBI 컨테이너에 소프트웨어를 추가하면 서브스크립션된 **RHEL** 호스트 또는 서브스크립션 취소(또는 **RHEL**이 아닌) 시스템에서 **UBI**를 업데이트하는 절차가 다릅니다.

11.1. 서브스크립션 호스트의 UBI 컨테이너에 소프트웨어 추가

등록된 **RHEL** 호스트에서 **UBI** 컨테이너를 실행하는 경우 모든 **UBI** 리포지토리 및 함께 표준 **UBI** 컨테이너 내에서 **RHEL Base** 및 **AppStream** 리포지토리가 활성화됩니다.

추가 리소스

- [UBI\(Universal Base Image\): 이미지, 리포지토리, 패키지 및 소스 코드](#)

11.2. 표준 UBI 컨테이너에 소프트웨어 추가

표준 **UBI** 컨테이너 내부에 소프트웨어를 추가하려면 비**UBI yum** 리포지토리를 비활성화하여 빌드한 컨테이너를 재배포할 수 있는지 확인합니다.

절차

1. **registry.access.redhat.com/ubi8/ubi** 이미지를 가져와서 실행합니다.

```
$ podman run -it --name myubi registry.access.redhat.com/ubi8/ubi
```

2. 패키지를 **myubi** 컨테이너에 추가합니다.

- **UBI** 리포지토리에 있는 패키지를 추가하려면 **UBI** 리포지토리를 제외한 모든 **yum** 리포지토리를 비활성화합니다. 예를 들어 **bzip2** 패키지를 추가하려면 다음을 수행합니다.

```
# yum install --disablerepo=* --enablerepo=ubi-8-appstream --enablerepo=ubi-8-baseos bzip2
```

- **UBI** 리포지토리에 없는 패키지를 추가하려면 리포지토리를 비활성화하지 마십시오. 예를 들어 **zsh** 패키지를 추가하려면 다음을 수행합니다.

```
# yum install zsh
```

- 다른 호스트 리포지토리에 있는 패키지를 추가하려면 필요한 리포지토리를 명시적으로 활성화합니다. 예를 들어 **codeready-builder-for-rhel-8-x86_64-rpms** 리포지토리에서 **python38-devel** 패키지를 설치하려면 다음을 수행합니다.

```
# yum install --enablerepo=codeready-builder-for-rhel-8-x86_64-rpms python38-devel
```

검증 단계

1. 컨테이너 내에서 활성화된 모든 리포지토리를 나열합니다.

```
# yum repolist
```

2. 필요한 리포지토리가 나열되었는지 확인합니다.

3. 설치된 패키지를 모두 나열합니다.

```
# rpm -qa
```

4. 필수 패키지가 나열되어 있는지 확인합니다.

참고

Red Hat UBI 리포지토리에 없는 **Red Hat** 패키지를 설치하면 서브스크립션된 **RHEL** 시스템 외부에서 컨테이너를 배포하는 기능이 제한될 수 있습니다.

11.3. 최소 UBI 컨테이너에 소프트웨어 추가

UBI yum 리포지토리는 기본적으로 **UBI Minimal** 이미지 내에서 활성화됩니다.

절차

1. **registry.access.redhat.com/ubi8/ubi-minimal** 이미지를 가져와서 실행합니다.

```
$ podman run -it --name myubimin registry.access.redhat.com/ubi8/ubi-minimal
```

2. **myubimin** 컨테이너에 패키지를 추가합니다.

- **UBI** 리포지토리에 있는 패키지를 추가하려면 리포지토리를 비활성화하지 마십시오. 예를 들어 **bzip2** 패키지를 추가하려면 다음을 수행합니다.

```
# microdnf install bzip2
```

- 다른 호스트 리포지토리에 있는 패키지를 추가하려면 필요한 리포지토리를 명시적으로 활성화합니다. 예를 들어 **codeready-builder-for-rhel-8-x86_64-rpms** 리포지토리에서 **python38-devel** 패키지를 설치하려면 다음을 수행합니다.

```
# microdnf install --enablerepo=codeready-builder-for-rhel-8-x86_64-rpms python38-devel
```

검증 단계

1. 컨테이너 내에서 활성화된 모든 리포지토리를 나열합니다.

```
# microdnf repolist
```

2. 필요한 리포지토리가 나열되었는지 확인합니다.

3. 설치된 패키지를 모두 나열합니다.

```
# rpm -qa
```

4. 필수 패키지가 나열되어 있는지 확인합니다.

참고

Red Hat UBI 리포지토리에 없는 Red Hat 패키지를 설치하면 서브스크립션된 RHEL 시스템 외부에서 컨테이너를 배포하는 기능이 제한될 수 있습니다.

11.4. 서브스크립션된 호스트의 UBI 컨테이너에 소프트웨어 추가

서브스크립션이 취소된 RHEL 시스템에서 소프트웨어 패키지를 추가할 때 리포지토리를 비활성화할 필요가 없습니다.

절차

- UBI 표준 또는 UBI init 이미지를 기반으로 실행 중인 컨테이너에 패키지를 추가합니다. 리포지토리를 비활성화하지 마십시오. **podman run** 명령을 사용하여 컨테이너를 실행합니다. 그런 다음 컨테이너 내부에서 **yum install** 명령을 사용합니다.

- 예를 들어 UBI 표준 기반 컨테이너에 **bzip2** 패키지를 추가하려면 다음을 수행합니다.

```
$ podman run -it --name myubi registry.access.redhat.com/ubi8/ubi
# yum install bzip2
```

1. 예를 들어 UBI init 기반 컨테이너에 **bzip2** 패키지를 추가하려면 다음을 수행합니다.

```
$ podman run -it --name myubimin registry.access.redhat.com/ubi8/ubi-
minimal
# microdnf install bzip2
```

검증 단계

1.

활성화된 모든 리포지토리를 나열합니다.

-

UBI 표준 또는 **UBI init** 이미지를 기반으로 컨테이너 내의 활성화된 리포지토리를 모두 나열하려면 다음을 수행합니다.

```
# yum repolist
```

-

UBI 최소 컨테이너를 기반으로 컨테이너 내의 활성화된 모든 리포지토리를 나열하려면 다음을 수행합니다.

```
# microdnf repolist
```

2.

필요한 리포지토리가 나열되었는지 확인합니다.

3.

설치된 패키지를 모두 나열합니다.

```
# rpm -qa
```

4.

필수 패키지가 나열되어 있는지 확인합니다.

11.5. UBI 기반 이미지 빌드

Buildah 유틸리티를 사용하여 컨테이너 파일에서 **UBI** 기반 웹 서버 컨테이너를 생성할 수 있습니다. 이미지에 재배포할 수 있는 **Red Hat** 소프트웨어만 포함되도록 하려면 비**UBI yum** 리포지토리를 모두 비활성화해야 합니다.

참고

UBI 최소 이미지의 경우 **yum** 대신 **microdnf** 를 사용하십시오.

```
RUN microdnf update -y && rm -rf /var/cache/yum
RUN microdnf install httpd -y && microdnf clean all
```

절차

1.

컨테이너 파일 만들기:

```
FROM registry.access.redhat.com/ubi8/ubi
USER root
LABEL maintainer="John Doe"
# Update image
RUN yum update --disablerepo=* --enablerepo=ubi-8-appstream --enablerepo=ubi-8-
baseos -y && rm -rf /var/cache/yum
RUN yum install --disablerepo=* --enablerepo=ubi-8-appstream --enablerepo=ubi-8-
baseos httpd -y && rm -rf /var/cache/yum
# Add default Web page and expose port
RUN echo "The Web Server is Running" > /var/www/html/index.html
EXPOSE 80
# Start the service
CMD ["-D", "FOREGROUND"]
ENTRYPOINT ["/usr/sbin/httpd"]
```

2.

컨테이너 이미지를 빌드합니다.

```
# buildah bud -t johndoe/webserver .
STEP 1: FROM registry.access.redhat.com/ubi8/ubi:latest
STEP 2: USER root
STEP 3: LABEL maintainer="John Doe"
STEP 4: RUN yum update --disablerepo=* --enablerepo=ubi-8-appstream --
enablerepo=ubi-8-baseos -y
...
Writing manifest to image destination
Storing signatures
--> f9874f27050
f9874f270500c255b950e751e53d37c6f8f6dba13425d42f30c2a8ef26b769f2
```

검증 단계

1.

웹 서버를 실행합니다.

```
# podman run -d --name=myweb -p 80:80 johndoe/webserver
bbe98c71d18720d966e4567949888dc4fb86eec7d304e785d5177168a5965f64
```

2.

웹 서버를 테스트합니다.

```
# curl http://localhost/index.html
The Web Server is Running
```

11.6. APPLICATION STREAM 런타임 이미지 사용

애플리케이션 스트림을 **기본으로 하는** 런타임 이미지는 컨테이너 빌드의 기반으로 사용할 수 있는 컨테이너 이미지 세트를 제공합니다.

지원되는 런타임 이미지는 **Python, Ruby, s2-core, s2i-base, .NET Core, PHP**입니다. 런타임 이미지는 **Red Hat Container Catalog**에서 사용할 수 있습니다.

참고

이러한 **UBI** 이미지에는 레거시 이미지와 동일한 기본 소프트웨어가 포함되어 있으므로 **Red Hat Software Collections 컨테이너 이미지 사용 가이드**에서 해당 이미지에 대해 알아볼 수 있습니다.

추가 리소스

- [Red Hat Container Catalog](#)
- [Red Hat 컨테이너 이미지 업데이트](#)

11.7. UBI 컨테이너 이미지 소스 코드 가져오기

소스 코드는 다운로드 가능한 컨테이너 이미지 형태로 모든 **Red Hat UBI** 기반 이미지에서 사용할 수 있습니다. 컨테이너로 패키징되어 있어도 소스 컨테이너 이미지를 실행할 수 없습니다. **Red Hat** 소스 컨테이너 이미지를 시스템에 설치하려면 **podman pull** 명령이 아닌 **skopeo** 명령을 사용합니다.

소스 컨테이너 이미지의 이름은 나타내는 바이너리 컨테이너를 기반으로 지정됩니다. 예를 들어 특정 표준 **RHEL UBI 8** 컨테이너 **registry.access.redhat.com/ubi8:8.1-397**의 경우 소스 컨테이너 이미지 (**registry.access.redhat.com/ubi8:8.1-397-source**)를 가져오기 위해 **- source**를 추가합니다.

절차

1. **skopeo copy** 명령을 사용하여 소스 컨테이너 이미지를 로컬 디렉터리에 복사합니다.

```
$ skopeo copy \
docker://registry.access.redhat.com/ubi8:8.1-397-source \
dir:$HOME/TEST
...
Copying blob 477bc8106765 done
Copying blob c438818481d3 done
```

```
...
Writing manifest to image destination
Storing signatures
```

2.

skopeo inspect 명령을 사용하여 소스 컨테이너 이미지를 검사합니다.

```
$ skopeo inspect dir:$HOME/TEST
{
  "Digest":
"sha256:7ab721ef3305271bbb629a6db065c59bbeb87bc53e7cbf88e2953a1217ba7322",
  "RepoTags": [],
  "Created": "2020-02-11T12:14:18.612461174Z",
  "DockerVersion": "",
  "Labels": null,
  "Architecture": "amd64",
  "Os": "linux",
  "Layers": [

"sha256:1ae73d938ab9f11718d0f6a4148eb07d38ac1c0a70b1d03e751de8bf3c2c87fa",
"sha256:9fe966885cb8712c47efe5ecc2eaa0797a0d5ffb8b119c4bd4b400cc9e255421",
"sha256:61b2527a4b836a4efbb82dfd449c0556c0f769570a6c02e112f88f8bbcd90166",
...
"sha256:cc56c782b513e2bdd2cc2af77b69e13df4ab624ddb856c4d086206b46b9b9e5f",
"sha256:dcf9396fdada4e6c1ce667b306b7f08a83c9e6b39d0955c481b8ea5b2a465b32",
"sha256:feb6d2ae252402ea6a6fca8a158a7d32c7e4572db0e6e5a5eab15d4e0777951e"
  ],
  "Env": null
}
```

3.

모든 콘텐츠의 압축을 풉니다.

```
$ cd $HOME/TEST
$ for f in $(ls); do tar xvf $f; done
```

4.

결과를 확인합니다.

```
$ find blobs/ rpm_dir/
blobs/
blobs/sha256
blobs/sha256/10914f1fff060ce31388f5ab963871870535aaaa551629f5ad182384d60fdf82
rpm_dir/
rpm_dir/gzip-1.9-4.el8.src.rpm
```

결과가 올바르면 이미지를 사용할 준비가 된 것입니다.

참고

관련 소스 컨테이너를 사용할 수 있도록 컨테이너 이미지를 릴리스한 후 몇 시간이 걸릴 수 있습니다.

추가 리소스

- **Skopeo-copy** 도움말 페이지
- **Skopeo-inspect** 도움말 페이지

12장. 컨테이너에서 SKOPEO, BUILDDAH, PODMAN 실행

이 장에서는 컨테이너에서 **Skopeo**, **Buildah** 및 **Podman**을 실행하는 방법을 설명합니다.

Skopeo를 사용하면 모든 계층이 있는 전체 이미지를 다운로드하지 않고도 원격 레지스트리의 이미지를 검사할 수 있습니다. **Skopeo**를 사용하여 이미지 복사, 이미지 서명, 이미지 동기화, 다양한 형식 및 계층 압축에서 이미지를 변환할 수도 있습니다.

Buildah는 **OCI** 컨테이너 이미지를 쉽게 빌드할 수 있습니다. **Buildah**를 사용하면 처음부터 또는 이미지를 시작점으로 사용하여 작업 중인 컨테이너를 생성할 수 있습니다. 작업 컨테이너에서 또는 **Containerfile**의 지침을 사용하여 이미지를 생성할 수 있습니다. 작동하는 컨테이너의 루트 파일 시스템을 마운트 및 마운트 해제할 수 있습니다.

Podman을 사용하면 컨테이너 및 이미지, 해당 컨테이너에 마운트된 볼륨 및 컨테이너 그룹에서 만든 포드를 관리할 수 있습니다. **Podman**은 컨테이너 라이프사이클 관리를 위한 **libpod** 라이브러리를 기반으로 합니다. **libpod** 라이브러리는 컨테이너, 포드, 컨테이너 이미지 및 볼륨을 관리하는 **API**를 제공합니다.

컨테이너에서 **Buildah**, **Skopeo** 및 **Podman**을 실행하는 이유:

- **CI/CD 시스템:**
 - **podman** 및 **Skopeo**: **Kubernetes** 내부에서 **CI/CD** 시스템을 실행하거나 **OpenShift**를 사용하여 컨테이너 이미지를 빌드하고 다양한 컨테이너 레지스트리에 해당 이미지를 배포할 수 있습니다. **Skopeo**를 **Kubernetes** 워크플로에 통합하려면 컨테이너에서 실행해야 합니다.
 - **Buildah**: 이미지를 지속적으로 빌드하는 **Kubernetes** 또는 **OpenShift CI/CD** 시스템 내에서 **OCI**/컨테이너 이미지를 빌드하려고 합니다. 이전에는 **Docker** 소켓을 사용하여 컨테이너 엔진에 연결하고 **Docker** 빌드 명령을 수행했습니다. 이는 암호가 안전하지 않은 암호 없이 시스템에 루트 액세스 권한을 제공하는 것과 동일합니다. 이러한 이유로 **Red Hat**은 컨테이너에서 **Buildah** 사용을 권장합니다.
- **다른 버전:**
 - 모두: 호스트에서 이전 **OS**를 실행 중이지만 최신 버전의 **Skopeo**, **Buildah** 또는 **Podman**을 실행하려고 합니다. 해결책은 컨테이너에서 컨테이너 틀을 실행하는 것입니다.

예를 들어, 이는 기본적으로 최신 버전에 액세스할 수 없는 **RHEL 7** 컨테이너 호스트의 **RHEL 8**에 제공된 최신 버전의 컨테이너 도구를 실행하는 데 유용합니다.

- **HPC 환경:**
 - 모두: **HPC** 환경의 일반적인 제한 사항은 루트가 아닌 사용자가 호스트에 패키지를 설치할 수 없다는 것입니다. 컨테이너에서 **Skopeo**, **Buildah** 또는 **Podman**을 실행하는 경우 이러한 특정 작업을 루트가 아닌 사용자로 수행할 수 있습니다.

12.1. 컨테이너에서 SKOPEO 실행

다음 절차에서는 **Skopeo**를 사용하여 원격 컨테이너 이미지를 검사하는 방법을 설명합니다. 컨테이너에서 **Skopeo**를 실행하면 컨테이너 루트 파일 시스템이 호스트 루트 파일 시스템과 격리됩니다. 호스트와 컨테이너 간에 파일을 공유하거나 복사하려면 파일과 디렉터리를 마운트해야 합니다.

사전 요구 사항

- **container-tools** 모듈이 설치되어 있어야 합니다.

```
# yum module install -y container-tools
```

절차

1. **registry.redhat.io** 레지스트리에 로그인합니다.

```
$ podman login registry.redhat.io
Username: myuser@mycompany.com
Password: *****
Login Succeeded!
```

2. **registry.redhat.io/rhel8/skopeo** 컨테이너 이미지를 가져옵니다.

```
$ podman pull registry.redhat.io/rhel8/skopeo
```

3. **Skopeo**를 사용하여 원격 컨테이너 이미지 **registry.access.redhat.com/ubi8/ubi**를 검사합니다.

```
$ podman run --rm registry.redhat.io/rhel8/skopeo skopeo inspect
docker://registry.access.redhat.com/ubi8/ubi
```

```
{
  "Name": "registry.access.redhat.com/ubi8/ubi",
  ...
  "Labels": {
    "architecture": "x86_64",
    ...
    "name": "ubi8",
    ...
    "summary": "Provides the latest release of Red Hat Universal Base Image 8.",
    "url":
"https://access.redhat.com/containers/#/registry.access.redhat.com/ubi8/images/8.2-347",
    ...
  },
  "Architecture": "amd64",
  "Os": "linux",
  "Layers": [
    ...
  ],
  "Env": [
    "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
    "container=oci"
  ]
}
```

rm 옵션은 컨테이너가 종료된 후 **registry.redhat.io/rhel8/skopeo** 이미지를 제거합니다.

추가 리소스

- [컨테이너에서 skopeo를 실행하는 방법](#)

12.2. 자격 증명을 사용하여 컨테이너에서 SKOPEO 실행

컨테이너 레지스트리를 사용하려면 데이터에 액세스하고 변경하기 위한 인증이 필요합니다. **Skopeo**는 자격 증명을 지정하는 다양한 방법을 지원합니다.

이 방법을 사용하면 명령줄에 **--cred USERNAME[:PASSWORD]** 옵션을 사용하여 자격 증명을 지정할 수 있습니다.

사전 요구 사항

- **container-tools** 모듈이 설치되어 있어야 합니다.

```
# yum module install -y container-tools
```

절차

- Skopeo를 사용하여 원격 컨테이너 이미지를 잠긴 레지스트리에 대해 검사합니다.

```
$ podman run --rm registry.redhat.io/rhel8/skopeo inspect --creds
$USER:$PASSWORD docker://$IMAGE
```

추가 리소스

- [컨테이너에서 skopeo를 실행하는 방법](#)

12.3. AUTHFILES를 사용하여 컨테이너에서 SKOPEO 실행

인증 파일(authfile)을 사용하여 자격 증명을 지정할 수 있습니다. **skopeo login** 명령은 특정 레지스트리에 로그인하고 인증 토큰을 **authfile**에 저장합니다. **authfile**을 사용하면 자격 증명을 반복적으로 입력할 필요가 없습니다.

동일한 호스트에서 실행하는 경우 **Skopeo**, **Buildah** 및 **Podman**과 같은 모든 컨테이너 툴은 동일한 **authfile**을 공유합니다. 컨테이너에서 **Skopeo**를 실행하는 경우 컨테이너의 **authfile**을 볼륨 마운트하여 호스트의 **authfile**을 공유하거나 컨테이너 내에서 다시 인증해야 합니다.

사전 요구 사항

- **container-tools** 모듈이 설치되어 있어야 합니다.

```
# yum module install -y container-tools
```

절차

- Skopeo를 사용하여 원격 컨테이너 이미지를 잠긴 레지스트리에 대해 검사합니다.

```
$ podman run --rm -v $AUTHFILE:/auth.json registry.redhat.io/rhel8/skopeo inspect
docker://$IMAGE
```

v \$AUTHFILE:/auth.json 옵션은 컨테이너의 **/auth.json**에 **authfile**을 마운트합니다. **Skopeo**는 이제 호스트의 **authfile**의 인증 토큰에 액세스하여 레지스트리에 대한 보안 액세스 권한을 얻을 수 있습니다.

다른 **Skopeo** 명령은 이와 유사하게 작동합니다. 예를 들면 다음과 같습니다.

- **skopeo-copy** 명령을 사용하여 **--source-creds** 및 **--dest-creds** 옵션을 사용하여 소스 및 대상 이미지의 명령줄에 자격 증명을 지정합니다. 또한 **/auth.json authfile**을 읽습니다.
- 소스 및 대상 이미지에 별도의 **authfile**을 지정하려면 **--source-authfile** 및 **--dest-authfile** 옵션을 사용하고 해당 **authfile**을 호스트에서 컨테이너로 볼륨 마운트하십시오.

추가 리소스

- [컨테이너에서 skopeo를 실행하는 방법](#)

12.4. 호스트에서 또는 호스트로 컨테이너 이미지 복사

Skopeo, **Buildah** 및 **Podman**은 동일한 로컬 컨테이너 이미지 스토리지를 공유합니다. 컨테이너를 호스트 컨테이너 스토리지에 복사하거나 호스트 컨테이너 스토리지에서 복사하려면 컨테이너를 **Skopeo** 컨테이너에 마운트해야 합니다.



참고

호스트 컨테이너 스토리지의 경로는 **root(/var/lib/containers/storage)**와 루트가 아닌 사용자(**\$HOME/.local/share/containers/storage**)마다 다릅니다.

사전 요구 사항

- **container-tools** 모듈이 설치되어 있어야 합니다.

```
# yum module install -y container-tools
```

절차

1. **registry.access.redhat.com/ubi8/ubi** 이미지를 로컬 컨테이너 스토리지에 복사합니다.

```
$ podman run --privileged --rm -v
$HOME/.local/share/containers/storage:/var/lib/containers/storage
registry.redhat.io/rhel8/skopeo skopeo copy
docker://registry.access.redhat.com/ubi8/ubi containers-
storage:registry.access.redhat.com/ubi8/ubi
```

- **--privileged** 옵션은 모든 보안 메커니즘을 비활성화합니다. 신뢰할 수 있는 환경에서는 이 옵션만 사용하는 것이 좋습니다.
- 보안 메커니즘을 비활성화하지 않도록 하려면 이미지를 **tarball** 또는 기타 경로 기반 이미지 전송으로 내보내고 **Skopeo** 컨테이너에 마운트합니다.
 - **\$ podman save --format oci-archive -o oci.tar \$IMAGE**
 - **\$ podman run --rm -v oci.tar:/oci.tar registry.redhat.io/rhel8/skopeo copy oci-archive:/oci.tar \$DESTINATION**

2.

선택 사항: 로컬 스토리지에 이미지를 나열합니다.

```
$ podman images
REPOSITORY                                TAG    IMAGE ID    CREATED    SIZE
registry.access.redhat.com/ubi8/ubi        latest ecbc6f53bba0 8 weeks ago 211 MB
```

추가 리소스

- [컨테이너에서 skopeo를 실행하는 방법](#)

12.5. 컨테이너에서 BUILDDAH 실행

이 절차에서는 컨테이너에서 **Buildah**를 실행하고 이미지를 기반으로 작동하는 컨테이너를 생성하는 방법을 설명합니다.

사전 요구 사항

- **container-tools** 모듈이 설치되어 있어야 합니다.

```
# yum module install -y container-tools
```

절차

1. **registry.redhat.io** 레지스트리에 로그인합니다.

```
$ podman login registry.redhat.io
Username: myuser@mycompany.com
Password: *****
Login Succeeded!
```

2.

registry.redhat.io/rhel8/buildah 이미지를 가져와 실행합니다.

```
# podman run --rm --device /dev/fuse -it registry.redhat.io/rhel8/buildah /bin/bash
```

- **rm** 옵션은 컨테이너가 종료된 후 **registry.redhat.io/rhel8/buildah** 이미지를 제거합니다.
- **device** 옵션은 컨테이너에 호스트 장치를 추가합니다.

3.

registry.access.redhat.com/ubi8 이미지를 사용하여 새 컨테이너를 생성합니다.

```
# buildah from registry.access.redhat.com/ubi8
...
ubi8-working-container
```

4.

ubi8-working-container 컨테이너 내부에서 **ls /** 명령을 실행합니다.

```
# buildah run --isolation=chroot ubi8-working-container ls /
bin boot dev etc home lib lib64 lost+found media mnt opt proc root run sbin
srv
```

5.

선택 사항: 로컬 스토리지의 모든 이미지를 나열합니다.

```
# buildah images
REPOSITORY          TAG    IMAGE ID    CREATED    SIZE
registry.access.redhat.com/ubi8 latest ecbc6f53bba0 5 weeks ago 211 MB
```

6.

선택 사항: 작업 중인 컨테이너 및 해당 기본 이미지를 나열합니다.

```
# buildah containers
CONTAINER ID  BUILDER  IMAGE ID    IMAGE NAME                CONTAINER NAME
0aaba7192762  *       ecbc6f53bba0 registry.access.redhat.com/ub... ubi8-working-
container
```

7.

선택 사항: `registry.access.redhat.com/ubi8` 이미지를 `registry.example.com`에 있는 로컬 레지스트리로 푸시합니다.

```
# buildah push ecbc6f53bba0 registry.example.com:5000/ubi8/ubi
```

추가 리소스

- [컨테이너에서 Buildah를 실행하는 모범 사례](#)

12.6. 권한 있는 PODMAN 컨테이너 및 권한이 없는 PODMAN 컨테이너

기본적으로 **Podman** 컨테이너는 권한이 없으며 호스트에서 운영 체제 일부를 수정할 수 없습니다. 기본적으로 컨테이너는 장치에 대한 제한된 액세스만 허용하기 때문입니다.

다음 목록은 권한 있는 컨테이너의 중요한 속성을 강조합니다. `podman run --privileged <image_name>` 명령을 사용하여 권한 있는 컨테이너를 실행할 수 있습니다.

- 권한 있는 컨테이너에는 컨테이너를 시작하는 사용자와 동일한 액세스 권한이 부여됩니다.
- 권한 있는 컨테이너는 호스트에서 컨테이너를 분리하는 보안 기능을 비활성화합니다. 삭제된 기능, 제한된 장치, 읽기 전용 마운트 지점, 접근 방식/**SELinux** 분리 및 **Seccomp** 필터는 모두 비활성화됩니다.
- 권한이 있는 컨테이너는 해당 컨테이너를 시작한 계정보다 많은 권한을 가질 수 없습니다.

추가 리소스

- [컨테이너 엔진에 --privileged 플래그를 사용하는 방법](#)
- [podman-run 도움말 페이지](#)

12.7. 확장된 권한으로 PODMAN 실행

rootless 환경에서 워크로드를 실행할 수 없는 경우 이러한 워크로드를 **root** 사용자로 실행해야 합니다. 확장된 권한으로 컨테이너를 실행하는 것은 모든 보안 기능을 비활성화하기 때문에 적절하게 수행되

어야 합니다.

사전 요구 사항

- **container-tools** 모듈이 설치되어 있어야 합니다.

```
# yum module install -y container-tools
```

절차

- **Podman** 컨테이너에서 **Podman** 컨테이너를 실행합니다.
- ```
$ podman run --privileged --name=privileged_podman
registry.access.redhat.com//podman podman run ubi8 echo hello
Resolved "ubi8" as an alias (/etc/containers/registries.conf.d/001-rhel-
shortnames.conf)
Trying to pull registry.access.redhat.com/ubi8:latest...
...
Storing signatures
hello
```
- **registry.access.redhat.com/ubi8/podman** 이미지를 기반으로 **privileged\_podman**이라는 외부 컨테이너를 실행합니다.
  - **privileged** 옵션은 호스트에서 컨테이너를 분리하는 보안 기능을 비활성화합니다.
  - **podman run ubi8 echo hello** 명령을 실행하여 **ubi8** 이미지를 기반으로 내부 컨테이너를 생성합니다.
  - **ubi8**의 짧은 이미지 이름이 별칭으로 확인되었습니다. 결과적으로 **registry.access.redhat.com/ubi8:latest** 이미지를 가져옵니다.

검증

- 모든 컨테이너를 나열합니다.

```
$ podman ps -a
```

| CONTAINER ID | IMAGE | COMMAND | CREATED | STATUS |
|--------------|-------|---------|---------|--------|
| PORTS        | NAMES |         |         |        |

```
52537876caf4 registry.access.redhat.com/ubi8/podman podman run ubi8 e...
30 seconds ago Exited (0) 13 seconds ago privileged_podman
```

#### 추가 리소스

- [컨테이너 내부에서 Podman 사용 방법](#)
- [podman-run 도움말 페이지](#)

## 12.8. 더 적은 권한으로 PODMAN 실행

**privileged** 옵션 없이 중첩된 두 개의 **Podman** 컨테이너를 실행할 수 있습니다. **privileged** 옵션 없이 컨테이너를 실행하는 것이 더 안전한 옵션입니다.

이는 가능한 한 가장 안전한 방법으로 다양한 버전의 **Podman**을 시도하려는 경우 유용할 수 있습니다.

#### 사전 요구 사항

- **container-tools** 모듈이 설치되어 있어야 합니다.

```
yum module install -y container-tools
```

#### 절차

- 중첩된 두 개의 컨테이너를 실행합니다.
 

```
$ podman run --name=unprivileged_podman --security-opt label=disable --user
podman --device /dev/fuse registry.access.redhat.com/ubi8/podman podman run ubi8
echo hello
```
- **registry.access.redhat.com/ubi8/ podman** 이미지를 기반으로 **privileged\_podman**이라는 외부 컨테이너를 실행합니다.
- **security -opt label=disable** 옵션은 호스트 **Podman**에서 **SELinux** 분리를 비활성화합니다. **SELinux**는 컨테이너화된 프로세스가 컨테이너 내부에서 실행하는 데 필요한 모든 파일 시스템을 마운트할 수 있도록 허용하지 않습니다.
-

**user podman** 옵션을 사용하면 외부 컨테이너 내부의 **Podman**이 사용자 네임스페이스 내에서 자동으로 실행됩니다.

- **device /dev/fuse** 옵션은 컨테이너 내부의 **fuse-overlayfs** 패키지를 사용합니다. 이 옵션은 컨테이너 내부의 **Podman**에서 사용할 수 있도록 **/dev/fuse** 를 외부 컨테이너에 추가합니다.
- **podman run ubi8 echo hello** 명령을 실행하여 **ubi8** 이미지를 기반으로 내부 컨테이너를 생성합니다.
- **ubi8**의 짧은 이미지 이름이 별칭으로 확인되었습니다. 결과적으로 **registry.access.redhat.com/ubi8:latest** 이미지를 가져옵니다.

#### 검증

- 모든 컨테이너를 나열합니다.

```
$ podman ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS
PORTS NAMES
a47b26290f43 podman run ubi8 e... 30 seconds ago Exited (0) 13 seconds
ago unprivileged_podman
```

## 12.9. PODMAN 컨테이너 내부에 컨테이너 빌드

다음 절차에서는 **Podman**을 사용하여 컨테이너에서 컨테이너를 실행하는 방법을 설명합니다. 이 예에서는 **Podman**을 사용하여 이 컨테이너 내에서 다른 컨테이너를 빌드하고 실행하는 방법을 보여줍니다. 컨테이너는 간단한 텍스트 기반 게임인 "**Moon-buggy**"를 실행합니다.

#### 사전 요구 사항

- **container-tools** 모듈이 설치되어 있어야 합니다.
- ```
# yum module install -y container-tools
```
- **registry.redhat.io** 레지스트리에 로그인되어 있습니다.

```
# podman login registry.redhat.io
```

절차

1. **registry.redhat.io/rhel8/podman** 이미지를 기반으로 하여 컨테이너를 실행합니다.

```
# podman run --privileged --name podman_container -it
registry.redhat.io/rhel8/podman /bin/bash
```

- **registry.redhat.io/rhel8/podman** 이미지를 기반으로 **podman_container** 라는 외부 컨테이너를 실행합니다.
- **it** 옵션은 컨테이너 내에서 대화형 **bash** 셸을 실행하도록 지정합니다.
- **privileged** 옵션은 호스트에서 컨테이너를 분리하는 보안 기능을 비활성화합니다.

2. **podman_container** 컨테이너 내에 **Containerfile** 을 생성합니다.

```
# vi Containerfile
FROM registry.access.redhat.com/ubi8/ubi
RUN yum install -y https://dl.fedoraproject.org/pub/epel/epel-release-latest-8.noarch.rpm
RUN yum -y install moon-buggy && yum clean all
CMD ["/usr/bin/moon-buggy"]
```

Containerfile 의 명령은 다음과 같은 빌드 명령을 수행합니다.

- **registry.access.redhat.com/ubi8/ubi** 이미지에서 컨테이너를 빌드합니다.
- **epel-release-latest-8.noarch.rpm** 패키지를 설치합니다.
- **north -buggy** 패키지를 설치합니다.
- 컨테이너 명령을 설정합니다.

3. **Containerfile** 을 사용하여 **galaxy -buggy** 라는 새 컨테이너 이미지를 빌드합니다.

```
# podman build -t moon-buggy .
```

4.

선택 사항: 모든 이미지를 나열합니다.

```
# podman images
REPOSITORY          TAG      IMAGE ID      CREATED      SIZE
localhost/moon-buggy latest   c97c58abb564  13 seconds ago  1.67 GB
registry.access.redhat.com/ubi8/ubi latest   4199acc83c6a  132seconds ago  213 MB
```

5.

war -buggy 컨테이너를 기반으로 새 컨테이너를 실행합니다.

```
# podman run -it --name moon moon-buggy
```

6.

선택 사항: **bootstrap -buggy** 이미지에 태그를 지정합니다.

```
# podman tag moon-buggy registry.example.com/moon-buggy
```

7.

선택 사항: **registries -buggy** 이미지를 레지스트리로 푸시합니다.

```
# podman push registry.example.com/moon-buggy
```

추가 리소스

-

[기술 프리뷰: 컨테이너 내에서 컨테이너 실행](#)

13장. BUILDDAH를 사용하여 컨테이너 이미지 빌드

Buildah는 **OCI 런타임 사양**을 충족하는 **OCI** 컨테이너 이미지를 쉽게 빌드합니다. **Buildah**를 사용하면 처음부터 또는 이미지를 시작점으로 사용하여 작업 중인 컨테이너를 생성할 수 있습니다. 작업 컨테이너에서 또는 **Containerfile**의 지침을 사용하여 이미지를 생성할 수 있습니다. 작동하는 컨테이너의 루트 파일 시스템을 마운트 및 마운트 해제할 수 있습니다.

13.1. BUILDDAH 툴

Buildah 사용은 다음과 같은 방법으로 **docker** 명령으로 이미지를 빌드하는 것과는 다릅니다.

데몬 없음

Buildah에는 컨테이너 런타임이 필요하지 않습니다.

기본 이미지 또는 스크래치

다른 컨테이너를 기반으로 이미지를 빌드하거나 빈 이미지(**scratch**)로 시작할 수 있습니다.

빌드 툴은 외부입니다.

Buildah에는 이미지 자체 내에 빌드 툴이 포함되지 않습니다. 결과적으로 **Buildah**:

- 빌드된 이미지의 크기를 줄입니다.
- 결과 이미지에서 소프트웨어를 제외하고(예: **gcc**, **make**, **yum**)을 제외하여 이미지의 보안을 강화합니다.
- 이미지 크기가 줄어들어 더 적은 리소스를 사용하여 이미지를 전송할 수 있습니다.

호환성

Buildah는 **Dockerfile**을 사용하여 컨테이너 이미지 빌드를 지원하므로 **Docker**에서 **Buildah**로 쉽게 전환할 수 있습니다.



참고

컨테이너 스토리지에 대한 기본 위치 **Buildah**는 **CRI-O** 컨테이너 엔진이 이미지 로컬 사본을 저장하는 데 사용하는 위치와 동일합니다. 결과적으로 **CRI-O** 또는 **Buildah**를 통해 레지스트리에서 가져오거나 **buildah** 명령으로 커밋한 이미지는 동일한 디렉터리 구조에 저장됩니다. 그러나 **CRI-O** 및 **Buildah**가 현재 이미지를 공유할 수 있더라도 컨테이너를 공유할 수 없습니다.

추가 리소스

- [Buildah - OCI\(Open Container Initiative\) 컨테이너 이미지 빌드를 용이하게 하는 툴](#)
- [Buildah 자습서 1: OCI 컨테이너 이미지 빌드](#)
- [Buildah 자습서 2: 컨테이너 레지스트리에서 Buildah 사용](#)
- [Buildah로 빌드: Dockerfiles, 명령줄 또는 스크립트](#)
- [rootless Buildah 작동 방식: 권한이 없는 환경에서 컨테이너 빌드](#)

13.2. BUILDDAH 설치

yum 명령을 사용하여 **Buildah** 툴을 설치합니다.

절차

- **Buildah** 툴을 설치합니다.

```
# yum -y install buildah
```

검증

- 도움말 메시지를 표시합니다.

```
# buildah -h
```

13.3. BUILDAH로 이미지 가져오기

buildah from 명령을 사용하여 처음부터 또는 지정된 이미지에 따라 시작 지점으로 새 작업 컨테이너를 생성합니다.

절차

- **registry.redhat.io/ubi8/ubi** 이미지를 기반으로 새 작업 컨테이너를 생성합니다.

```
# buildah from registry.access.redhat.com/ubi8/ubi
Getting image source signatures
Copying blob...
Writing manifest to image destination
Storing signatures
ubi-working-container
```

검증

1. 로컬 스토리지의 모든 이미지를 나열합니다.

```
# buildah images
REPOSITORY                                TAG      IMAGE ID      CREATED      SIZE
registry.access.redhat.com/ubi8/ubi      latest   272209ff0ae5  2 weeks ago  234 MB
```

2. 작업 중인 컨테이너 및 해당 기본 이미지를 나열합니다.

```
# buildah containers
CONTAINER ID  BUILDER  IMAGE ID      IMAGE NAME                                CONTAINER NAME
01eab9588ae1  *        272209ff0ae5  registry.access.redhat.com/ubi8/ubi      ubi-working-
container
```

추가 리소스

- **Buildah-from** 도움말 페이지
- **buildah-images** 도움말 페이지
- **buildah-containers** 매뉴얼 페이지

13.4. 컨테이너 내에서 명령 실행

buildah run 명령을 사용하여 컨테이너에서 명령을 실행합니다.

사전 요구 사항

- 가져온 이미지는 로컬 시스템에서 사용할 수 있습니다.

절차

- 운영 체제 버전을 표시합니다.

```
# buildah run ubi-working-container cat /etc/redhat-release
Red Hat Enterprise Linux release 8.4 (Ootpa)
```

추가 리소스

- buildah-run** 도움말 페이지

13.5. BUILDHAH를 사용하여 CONTAINERFILE에서 이미지 빌드

buildah bud 명령을 사용하여 컨테이너 파일의 지침을 사용하여 이미지를 빌드합니다.



참고

buildah bud 명령은 컨텍스트 디렉터리에 있는 경우 **Containerfile** 을 사용합니다. **buildah bud** 명령은 **Dockerfile** 을 사용합니다. 그렇지 않으면 **--file** 옵션으로 파일을 지정할 수 있습니다. **Containerfile** 및 **Dockerfile** 내에서 사용할 수 있는 사용 가능한 명령은 동일합니다.

절차

- 컨테이너 파일 만들기:

```
# cat Containerfile
FROM registry.access.redhat.com/ubi8/ubi
ADD myecho /usr/local/bin
ENTRYPOINT "/usr/local/bin/myecho"
```

2. **myecho** 스크립트를 생성합니다.

```
# cat myecho
echo "This container works!"
```

3. **myecho** 스크립트의 액세스 권한을 변경합니다.

```
# chmod 755 myecho
```

4. 현재 디렉터리에서 **Containerfile** 을 사용하여 **myecho** 이미지를 빌드합니다.

```
# buildah bud -t myecho .
STEP 1: FROM registry.access.redhat.com/ubi8/ubi
STEP 2: ADD myecho /usr/local/bin
STEP 3: ENTRYPOINT "/usr/local/bin/myecho"
STEP 4: COMMIT myecho
...
Storing signatures
```

검증

1. 모든 이미지를 나열합니다.

```
# buildah images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
localhost/myecho	latest	b28cd00741b3	About a minute ago	234 MB

2. **localhost/my echo** 이미지를 기반으로 **my echo** 컨테이너를 실행합니다.

```
# podman run --name=myecho localhost/myecho
This container works!
```

3. 모든 컨테이너를 나열합니다.

```
# podman ps -a
```

CONTAINER ID	IMAGE	COMMAND	STATUS	CREATED	PORTS
0d97517428d	localhost/myecho	myecho	Exited (0) 13 seconds ago	12 seconds ago	



참고

podman history 명령을 사용하여 이미지에 사용된 각 계층에 대한 정보를 표시할 수 있습니다.

추가 리소스

- **buildah-bud** 도움말 페이지

13.6. BUILDHAH를 사용하여 컨테이너 및 이미지 검사

buildah inspect 명령을 사용하여 컨테이너 또는 이미지에 대한 정보를 표시합니다.

사전 요구 사항

- 이미지는 **Containerfile**의 지침을 사용하여 빌드되었습니다. 자세한 내용은 [Buildah를 사용하여 Containerfile에서 이미지 빌드 섹션](#)을 참조하십시오.

절차

- 이미지를 검사합니다.
 - **myecho** 이미지를 검사하려면 다음을 입력합니다.

```
# buildah inspect localhost/myecho
{
  "Type": "buildah 0.0.1",
  "FromImage": "localhost/myecho:latest",
  "FromImageID":
    "b28cd00741b38c92382ee806e1653eae0a56402bcd2c8d31bdcd36521bc267a4",
  "FromImageDigest":
    "sha256:0f5b06cbd51b464fab93ce4fe852a9038cdd7c7b7661cd7efef8f9ae8a59585"
,
  "Config":
    ...
    "Entrypoint": [
      "/bin/sh",
      "-c",
      "\"/usr/local/bin/myecho\""
    ],
    ...
}
```

○

myecho 이미지에서 작업 컨테이너를 검사하려면 다음을 수행합니다.

i.

localhost/myecho 이미지를 기반으로 작동하는 컨테이너를 생성합니다.

```
# buildah from localhost/myecho
```

ii.

myecho-working-container 컨테이너를 검사합니다.

```
# buildah inspect ubi-working-container
{
  "Type": "buildah 0.0.1",
  "FromImage": "registry.access.redhat.com/ubi8/ubi:latest",
  "FromImageID":
    "272209ff0ae5fe54c119b9c32a25887e13625c9035a1599feba654aa7638262d",
  "FromImageDigest":
    "sha256:77623387101abefbf83161c7d5a0378379d0424b2244009282acb39d42f1f
    e13",
  "Config":
    ...
  "Container": "ubi-working-container",
  "ContainerID":
    "01eab9588ae1523746bb706479063ba103f6281ebaeeccb5dc42b70e450d5ad0",
  "ProcessLabel": "system_u:system_r:container_t:s0:c162,c1000",
  "MountLabel": "system_u:object_r:container_file_t:s0:c162,c1000",
  ...
}
```

추가 리소스

●

Buildah-inspect 도움말 페이지

13.7. BUILDDAH 마운트를 사용하여 컨테이너 수정

buildah inspect 명령을 사용하여 컨테이너 또는 이미지에 대한 정보를 표시합니다.

사전 요구 사항

●

Containerfile의 지침을 사용하여 빌드된 이미지. 자세한 내용은 [Buildah를 사용하여 Containerfile에서 이미지 빌드](#) 섹션을 참조하십시오.

절차

1.

registry.access.redhat.com/ubi8/ubi 이미지를 기반으로 작업 중인 컨테이너를 생성하고 컨테이너 이름을 **mycontainer** 변수에 저장합니다.

```
# mycontainer=$(buildah from localhost/myecho)
```

```
# echo $mycontainer
myecho-working-container
```

2.

myecho-working-container 컨테이너를 마운트하고 마운트 지점 경로를 **mymount** 변수에 저장합니다.

```
# mymount=$(buildah mount $mycontainer)
```

```
# echo $mymount
/var/lib/containers/storage/overlay/c1709df40031dda7c49e93575d9c8eebcaa5d8129033
a58e5b6a95019684cc25/merged
```

3.

myecho 스크립트를 수정하고 실행 가능하게 만듭니다.

```
# echo 'echo "We modified this container."' >> $mymount/usr/local/bin/myecho
# chmod +x $mymount/usr/local/bin/myecho
```

4.

myecho -working-container 컨테이너에서 **myecho 2** 이미지를 생성합니다.

```
# buildah commit $mycontainer containers-storage:myecho2
```

검증

1.

로컬 스토리지의 모든 이미지를 나열합니다.

```
# buildah images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
docker.io/library/myecho2	latest	4547d2c3e436	4 minutes ago	234 MB
localhost/myecho	latest	b28cd00741b3	56 minutes ago	234 MB

2.

docker.io/library/my echo2 이미지를 기반으로 **my echo2** 컨테이너를 실행합니다.

```
# podman run --name=myecho2 docker.io/library/myecho2
This container works!
We even modified it.
```

추가 리소스

- [buildah-mount 도움말 페이지](#)
- [buildah-commit 도움말 페이지](#)

13.8. BUILDDAH 복사 및 BUILDDAH 구성을 사용하여 컨테이너 수정

buildah 복사 명령을 사용하여 마운트하지 않고 파일을 컨테이너로 복사합니다. 그런 다음 **buildah config** 명령을 사용하여 컨테이너를 구성하여 기본적으로 생성한 스크립트를 실행할 수 있습니다.

사전 요구 사항

- **Containerfile**의 지침을 사용하여 빌드된 이미지. 자세한 내용은 [Buildah를 사용하여 Containerfile에서 이미지 빌드 섹션](#)을 참조하십시오.

절차

1. **newecho** 라는 스크립트를 생성하고 이를 실행 가능하게 만듭니다.

```
# cat newecho
echo "I changed this container"
# chmod 755 newecho
```

2. 새 작업 컨테이너를 생성합니다.

```
# buildah from myecho:latest
myecho-working-container-2
```

3. **newecho** 스크립트를 컨테이너 내부의 **/usr/local/bin** 디렉토리에 복사합니다.

```
# buildah copy myecho-working-container-2 newecho /usr/local/bin
```

4. **new echo** 스크립트를 새 진입점으로 사용하도록 구성을 변경합니다.

```
# buildah config --entrypoint "/bin/sh -c /usr/local/bin/newecho" myecho-working-container-2
```

5.

선택 사항: **myecho-working-container-2** 컨테이너를 실행하면 **newecho** 스크립트가 실행됩니다.

```
# buildah run myecho-working-container-2 -- sh -c '/usr/local/bin/newecho'
I changed this container
```

6.

myecho-working-container-2 컨테이너를 **mynewecho** 라는 새 이미지에 커밋합니다:

```
# buildah commit myecho-working-container-2 containers-storage:mynewecho
```

검증

•

로컬 스토리지의 모든 이미지를 나열합니다.

```
# buildah images
REPOSITORY                                TAG    IMAGE ID    CREATED    SIZE
docker.io/library/mynewecho               latest fa2091a7d8b6 8 seconds ago 234 MB
```

추가 리소스

•

Buildah-copy 도움말 페이지

•

buildah-config 도움말 페이지

•

buildah-commit 도움말 페이지

•

buildah-run 도움말 페이지

13.9. BUILDHAH를 사용하여 처음부터 이미지 생성

기본 이미지로 시작하는 대신 최소 컨테이너 메타데이터 양을 보유하는 새 컨테이너를 생성할 수 있습니다.

스크래치 컨테이너에서 이미지를 생성할 때 다음을 고려하십시오.

•

종속성이 없는 실행 파일을 스크래치 이미지에 복사하고 몇 가지 구성 설정을 사용하여 최소한의 컨테이너를 작업할 수 있습니다.

- **yum** 또는 **rpm** 과 같은 도구를 사용하려면 **RPM** 데이터베이스를 초기화하고 컨테이너에 **release** 패키지를 추가해야 합니다.
- 많은 패키지를 추가하는 경우 이미지 대신 표준 **UBI** 또는 최소 **UBI** 이미지를 사용하는 것이 좋습니다.

절차

이 절차에서는 웹 서비스 **httpd**를 컨테이너에 추가하고 을 실행하도록 구성합니다.

1. 비어 있는 컨테이너를 생성합니다.

```
# buildah from scratch
working-container
```

2. **working-container** 컨테이너를 마운트하고 마운트 지점 경로를 **scratchmnt** 변수에 저장합니다.

```
# scratchmnt=$(buildah mount working-container)

# echo $scratchmnt
/var/lib/containers/storage/overlay/be2eaecf9f74b6acfe4d0017dd5534fde06b2fa8de9ed
875691f6ccc791c1836/merged
```

3. 스크래치 이미지 내에서 **RPM** 데이터베이스를 초기화하고 **redhat-release** 패키지를 추가합니다.

```
# yum install -y --releasever=8 --installroot=$scratchmnt redhat-release
```

4. **httpd** 서비스를 스크래치 디렉터리에 설치합니다.

```
# yum install -y --setopt=reposdir=/etc/yum.repos.d \
  --installroot=$scratchmnt \
  --setopt=cachedir=/var/cache/dnf httpd
```

5.

\$scratchmnt/var/www/html/index.html 파일을 생성합니다.

```
# mkdir -p $scratchmnt/var/www/html
# echo "Your httpd container from scratch works!" >
$scratchmnt/var/www/html/index.html
```

6.

컨테이너에서 직접 **httpd** 데몬을 실행하도록 **working-container** 를 구성합니다.

```
# buildah config --cmd "/usr/sbin/httpd -DFOREGROUND" working-container
# buildah config --port 80/tcp working-container
# buildah commit working-container localhost/myhttpd:latest
```

검증

1.

로컬 스토리지의 모든 이미지를 나열합니다.

```
# podman images
REPOSITORY          TAG   IMAGE ID   CREATED   SIZE
localhost/myhttpd    latest 08da72792f60 2 minutes ago 121 MB
```

2.

localhost/myhttpd 이미지를 실행하고 컨테이너와 호스트 시스템 간의 포트 매핑을 구성합니다.

```
# podman run -p 8080:80 -d --name myhttpd 08da72792f60
```

3.

웹 서버를 테스트합니다.

```
# curl localhost:8080
Your httpd container from scratch works!
```

추가 리소스

- [buildah-config](#) 도움말 페이지
- [buildah-commit](#) 도움말 페이지

13.10. 프라이빗 레지스트리로 컨테이너 내보내기

buildah push 명령을 사용하여 로컬 스토리지에서 퍼블릭 또는 프라이빗 리포지토리로 이미지를 내보냅니다.

사전 요구 사항

- 이미지는 **Containerfile**의 지침을 사용하여 빌드되었습니다. 자세한 내용은 [Buildah를 사용하여 Containerfile에서 이미지 빌드](#) 섹션을 참조하십시오.

절차

1. 머신에 로컬 레지스트리를 생성합니다.
- ```
podman run -d -p 5000:5000 registry:2
```
2. **myecho:latest** 이미지를 **localhost** 레지스트리로 푸시합니다.

```
buildah push --tls-verify=false myecho:latest localhost:5000/myecho:latest
Getting image source signatures
Copying blob sha256:e4efd0...
...
Writing manifest to image destination
Storing signatures
```

#### 검증

1. **localhost** 리포지토리의 모든 이미지를 나열합니다.
- ```
# curl http://localhost:5000/v2/_catalog
{"repositories":["myecho2"]}

# curl http://localhost:5000/v2/myecho2/tags/list
{"name":"myecho","tags":["latest"]}
```
2. **docker://localhost:5000/myecho:latest** 이미지를 검사합니다.

```
# skopeo inspect --tls-verify=false docker://localhost:5000/myecho:latest | less
{
  "Name": "localhost:5000/myecho",
  "Digest": "sha256:8999ff6050...",
  "RepoTags": [
    "latest"
  ],
}
```

```
"Created": "2021-06-28T14:44:05.919583964Z",
"DockerVersion": "",
"Labels": {
  "architecture": "x86_64",
  "authoritative-source-url": "registry.redhat.io",
  ...
}
```

3.

localhost:5000/myecho 이미지를 가져옵니다.

```
# podman pull --tls-verify=false localhost:5000/myecho2
# podman run localhost:5000/myecho2
This container works!
```

추가 리소스

- [buildah-push 도움말 페이지](#)

13.11. DOCKER HUB로 컨테이너 푸시

Docker Hub 자격 증명을 사용하여 **buildah** 명령으로 **Docker Hub**에서 이미지를 푸시하고 가져옵니다.

사전 요구 사항

- **Containerfile**의 지침을 사용하여 빌드된 이미지. 자세한 내용은 [Buildah를 사용하여 Containerfile에서 이미지 빌드](#) 섹션을 참조하십시오.

절차

1.

docker.io/library/myecho:latest를 **Docker Hub**로 푸시합니다. 사용자 이름 및 암호를 **Docker Hub** 인증 정보로 바꿉니다.

```
# buildah push --creds username:password \
  docker.io/library/myecho:latest docker://testaccountXX/myecho:latest
```

검증

- **docker.io/testaccountXX/myecho:latest** 이미지를 가져와서 실행합니다.

○

Podman 툴 사용:

```
# podman run docker.io/testaccountXX/myecho:latest
This container works!
```

○

Buildah 및 Podman 툴 사용:

```
# buildah from docker.io/testaccountXX/myecho:latest
myecho2-working-container-2
# podman run myecho-working-container-2
```

추가 리소스

●

buildah-push 도움말 페이지

13.12. BUILDAH로 이미지 제거

buildah rmi 명령을 사용하여 로컬에 저장된 컨테이너 이미지를 제거합니다. ID 또는 이름으로 이미지를 제거할 수 있습니다.

절차

1.

로컬 시스템의 모든 이미지를 나열합니다.

```
# buildah images
REPOSITORY                                TAG    IMAGE ID    CREATED    SIZE
localhost/johndoe/webserver               latest dc5fcc610313 46 minutes ago 263 MB
docker.io/library/mynewecho               latest fa2091a7d8b6 17 hours ago 234 MB
docker.io/library/myecho2                 latest 4547d2c3e436 6 days ago 234 MB
localhost/myecho                          latest b28cd00741b3 6 days ago 234 MB
localhost/ubi-micro-httpd                 latest c6a7678c4139 12 days ago 152 MB
registry.access.redhat.com/ubi8/ubi       latest 272209ff0ae5 3 weeks ago 234 MB
```

2.

localhost/myecho 이미지를 삭제합니다.

```
# buildah rmi localhost/myecho
```

●

여러 이미지를 제거하려면 다음을 수행합니다.

```
# buildah rmi docker.io/library/mynewecho docker.io/library/myecho2
```

- 시스템에서 모든 이미지를 제거하려면 다음을 수행합니다.

```
# buildah rmi -a
```

- 여러 이름(태그)이 연결된 이미지를 제거하려면 **-f** 옵션을 추가하여 제거합니다.

```
# buildah rmi -f localhost/ubi-micro-httpd
```

검증

- 이미지가 제거되었는지 확인합니다.

```
# buildah images
```

추가 리소스

- [buildah-rmi 도움말 페이지](#)

13.13. BUILDDAH를 사용하여 컨테이너 제거

buildah rm 명령을 사용하여 컨테이너를 제거합니다. 컨테이너 **ID** 또는 이름을 사용하여 제거할 컨테이너를 지정할 수 있습니다.

사전 요구 사항

- 하나 이상의 컨테이너가 중지되었습니다.

절차

1. 모든 컨테이너를 나열합니다.

```
# buildah containers
CONTAINER ID  BUILDER  IMAGE ID  IMAGE NAME  CONTAINER NAME
05387e29ab93  *        c37e14066ac7  docker.io/library/myecho:latest  myecho-working-container
```

2.

myecho-working-container 컨테이너를 제거합니다.

```
# buildah rm myecho-working-container  
05387e29ab93151cf52e9c85c573f3e8ab64af1592b1ff9315db8a10a77d7c22
```

검증

- 컨테이너가 제거되었는지 확인합니다.

```
# buildah containers
```

추가 리소스

- **Buildah-rm** 도움말 페이지

14장. 컨테이너 모니터링

이 장에서는 컨테이너의 상태 결정, 시스템 및 포트 정보 표시, **Podman** 이벤트 모니터링 등 **Podman** 환경을 관리할 수 있는 유용한 **Podman** 명령에 중점을 둡니다.

14.1. 컨테이너에서 상태 점검 수행

상태 점검을 통해 컨테이너 내부에서 실행 중인 프로세스의 상태 또는 준비 상태를 확인할 수 있습니다. 상태 점검은 5가지 기본 구성 요소로 구성됩니다.

- 명령
- retries
- 간격
- start-period
- Timeout

상태 점검 구성 요소에 대한 설명은 다음과 같습니다.

명령

Podman은 대상 컨테이너 내부에서 명령을 실행하고 종료 코드를 기다립니다.

나머지 4개의 구성 요소는 상태 점검 예약과 관련이 있으며 선택 사항입니다.

retries

컨테이너가 "unhealthy"로 표시되기 전에 발생해야 하는 연속 실패 상태 점검 수를 정의합니다. 상태 점검이 성공하면 재시도 카운터가 재설정됩니다.

간격

healthcheck 명령을 실행할 때까지의 시간을 설명합니다. 시스템이 상태 점검을 실행하는 데 많은 시간을 할애하게 됩니다. 큰 간격은 시간 제한을 따라잡는 데 어려움을 겪습니다.

start-period

컨테이너가 시작되는 시간과 상태 점검 실패를 무시하려는 시기 사이의 시간을 설명합니다.

Timeout

상태 점검을 완료한 후 완료해야 하는 시간을 설명합니다.

컨테이너 내부에서 상태 점검이 실행됩니다. **Healthcheck**는 서비스의 상태 상태인 것을 알고 성공 및 실패한 상태 점검을 구분할 수 있는 경우에만 의미가 있습니다.

절차

1.

상태 확인을 정의합니다.

```
$ podman run -dt --name hc1 -p 8080:8080 --health-cmd='curl http://localhost:8080 || exit 1' --health-interval=0 registry.access.redhat.com/ubi8/httpd-24
```

-

health -cmd 옵션은 컨테이너의 **healthcheck** 명령을 설정합니다.

-

값이 0인 **-health-interval=0** 옵션은 **healthcheck**를 수동으로 실행하려고 함을 나타냅니다.

2.

상태 점검을 수동으로 실행합니다.

```
$ podman healthcheck run hc1
Healthy
```

3.

필요한 경우 마지막 명령의 종료 상태를 확인할 수 있습니다.

```
$ echo $?
0
```

value 값은 성공을 의미합니다.

추가 리소스

- [podman-run 도움말 페이지](#)
- [Podman을 사용하여 컨테이너 필수 및 가용성 모니터링](#)

14.2. PODMAN 시스템 정보 표시

podman system 명령을 사용하면 **Podman** 시스템을 관리할 수 있습니다. 이 섹션에서는 **Podman** 시스템 정보를 표시하는 방법에 대한 정보를 제공합니다.

절차

- **Podman** 시스템 정보를 표시합니다.
 - **Podman** 디스크 사용량을 표시하려면 다음을 입력합니다.

```
$ podman system df
TYPE          TOTAL    ACTIVE   SIZE     RECLAIMABLE
Images        3        2        1.085GB  233.4MB (0%)
Containers    2        0        28.17kB  28.17kB (100%)
Local Volumes 3        0        0B       0B (0%)
```

- 공간 사용량에 대한 자세한 정보를 표시하려면 다음을 입력합니다.

```
$ podman system df -v
Images space usage:

REPOSITORY                                TAG      IMAGE ID   CREATED   SIZE
registry.access.redhat.com/ubi8           latest   b1e63aaae5cf 13 days   233.4MB
233.4MB  0B      0
registry.access.redhat.com/ubi8/httpd-24 latest    0d04740850e8 13 days   461.5MB
461.5MB  0B      461.5MB   1
registry.redhat.io/rhel8/podman           latest   dce10f591a2d 13 days   390.6MB
233.4MB  157.2MB  1

Containers space usage:

CONTAINER ID  IMAGE          COMMAND                  LOCAL VOLUMES  SIZE
CREATED      STATUS        NAMES
311180ab99fb  0d04740850e8  /usr/bin/run-httpd      0              28.17kB  16 hours
exited      hc1
```

```
bedb6c287ed6 dce10f591a2d podman run ubi8 echo hello 0 0B 11
hours configured dazzling_tu
```

Local Volumes space usage:

VOLUME NAME	LINKS	SIZE
76de0efa83a3dae1a388b9e9e67161d28187e093955df185ea228ad0b3e435d0	0	0B
8a1b4658aecc9ff38711a2c7f2da6de192c5b1e753bb7e3b25e9bf3bb7da8b13	0	0B
d9cab4f6ccbcf2ac3cd750d2efff9d2b0f29411d430a119210dd242e8be20e26	0	0B

○

호스트에 대한 정보, 현재 스토리지 통계 및 **Podman** 빌드를 표시하려면 다음을 입력합니다.

```
$ podman system info
host:
  arch: amd64
  buildahVersion: 1.22.3
  cgroupControllers: []
  cgroupManager: cgroupfs
  cgroupVersion: v1
  conmon:
    package: conmon-2.0.29-1.module+el8.5.0+12381+e822eb26.x86_64
    path: /usr/bin/conmon
    version: 'conmon version 2.0.29, commit:
7d0fa63455025991c2fc641da85922fde889c91b'
  cpus: 2
  distribution:
    distribution: '"rhel"'
    version: "8.5"
  eventLogger: file
  hostname: localhost.localdomain
  idMappings:
    gidmap:
      - container_id: 0
        host_id: 1000
        size: 1
      - container_id: 1
        host_id: 100000
        size: 65536
    uidmap:
      - container_id: 0
        host_id: 1000
        size: 1
      - container_id: 1
        host_id: 100000
        size: 65536
  kernel: 4.18.0-323.el8.x86_64
  linkmode: dynamic
  memFree: 352288768
  memTotal: 2819129344
  ociRuntime:
```

```

name: runc
package: runc-1.0.2-1.module+el8.5.0+12381+e822eb26.x86_64
path: /usr/bin/runc
version: |-
  runc version 1.0.2
  spec: 1.0.2-dev
  go: go1.16.7
  libseccomp: 2.5.1
os: linux
remoteSocket:
  path: /run/user/1000/podman/podman.sock
security:
  apparmorEnabled: false
  capabilities:
CAP_NET_RAW,CAP_CHOWN,CAP_DAC_OVERRIDE,CAP_FOWNER,CAP_FSETID,
CAP_KILL,CAP_NET_BIND_SERVICE,CAP_SETFCAP,CAP_SETGID,CAP_SETPCAP
,CAP_SETUID,CAP_SYS_CHROOT
  rootless: true
  seccompEnabled: true
  seccompProfilePath: /usr/share/containers/seccomp.json
  selinuxEnabled: true
servicelsRemote: false
slirp4netns:
  executable: /usr/bin/slirp4netns
  package: slirp4netns-1.1.8-1.module+el8.5.0+12381+e822eb26.x86_64
  version: |-
    slirp4netns version 1.1.8
    commit: d361001f495417b880f20329121e3aa431a8f90f
    libslirp: 4.4.0
    SLIRP_CONFIG_VERSION_MAX: 3
    libseccomp: 2.5.1
  swapFree: 3113668608
  swapTotal: 3124752384
  uptime: 11h 24m 12.52s (Approximately 0.46 days)
registries:
  search:
    - registry.fedoraproject.org
    - registry.access.redhat.com
    - registry.centos.org
    - docker.io
store:
  configFile: /home/user/.config/containers/storage.conf
  containerStore:
    number: 2
    paused: 0
    running: 0
    stopped: 2
  graphDriverName: overlay
  graphOptions:
    overlay.mount_program:
      Executable: /usr/bin/fuse-overlayfs
      Package: fuse-overlayfs-1.7.1-1.module+el8.5.0+12381+e822eb26.x86_64
      Version: |-
        fusermount3 version: 3.2.1
        fuse-overlayfs: version 1.7.1
        FUSE library version 3.2.1

```

```

    using FUSE kernel interface version 7.26
graphRoot: /home/user/.local/share/containers/storage
graphStatus:
  Backing Filesystem: xfs
  Native Overlay Diff: "false"
  Supports d_type: "true"
  Using metacopy: "false"
imageStore:
  number: 3
runRoot: /run/user/1000/containers
volumePath: /home/user/.local/share/containers/storage/volumes
version:
  APIVersion: 3.3.1
  Built: 1630360721
  BuiltTime: Mon Aug 30 23:58:41 2021
  GitCommit: ""
  GoVersion: go1.16.7
  OsArch: linux/amd64
  Version: 3.3.1

```

○

사용되지 않은 모든 컨테이너, 이미지 및 볼륨 데이터를 제거하려면 다음을 입력합니다.

```

$ podman system prune
WARNING! This will remove:
- all stopped containers
- all stopped pods
- all dangling images
- all build cache
Are you sure you want to continue? [y/N] y

```

■

podman system prune 명령은 사용되지 않는 모든 컨테이너(**dangling** 및 **unreferenced**), **Pod** 및 선택적으로 로컬 스토리지의 볼륨을 제거합니다.

■

사용하지 않는 모든 이미지를 삭제하려면 **--all** 옵션을 사용합니다. 사용하지 않는 이미지는 이미지와 이미지를 기반으로 한 컨테이너가 없는 모든 이미지를 유추합니다.

■

volume 옵션을 사용하여 볼륨을 정리합니다. 현재 볼륨을 사용하는 컨테이너가 없는 경우 중요한 데이터가 삭제되지 않도록 볼륨은 기본적으로 제거되지 않습니다.

추가 리소스

●

podman-system-df man 페이지

●

podman-system-info 매뉴얼 페이지

- [podman-system-prune](#) 매뉴얼 페이지

14.3. PODMAN 이벤트 유형

Podman에서 발생하는 이벤트를 모니터링할 수 있습니다. 여러 이벤트 유형이 존재하며 각 이벤트 유형은 다른 상태를 보고합니다.

컨테이너 이벤트 유형은 다음 상태를 보고합니다.

- **attach**
- **checkpoint**
- **cleanup**
- 커밋
- **create**
- **exec**
- 내보내기
- **import**
- **init**
- 강제 종료

- **Mount**
- **pause**
- **prune**
- 제거
- 재시작
- 복원
- **start**
- 중지
- **sync**
- **unmount**
- 일시 중지 해제

Pod 이벤트 유형은 다음 상태를 보고합니다.

- **create**
- 강제 종료

- **pause**
- 제거
- **start**
- 중지
- 일시 중지 해제

이미지 이벤트 유형은 다음 상태를 보고합니다.

- **prune**
- **push**
- **pull**
- 저장
- 제거
- **tag**
- **untag**

시스템 유형에서 다음 상태를 보고합니다.

- 새로 고침
- **renumber**

볼륨 유형은 다음 상태를 보고합니다.

- **create**
- **prune**
- 제거

추가 리소스

- **podman-events** 도움말 페이지

14.4. PODMAN 이벤트 모니터링

Podman에서 발생하는 이벤트를 모니터링하고 인쇄할 수 있습니다. 각 이벤트에는 타임스탬프, 유형, 상태, 이름(해당되는 경우) 및 이미지가 포함됩니다(해당하는 경우).

절차

- **Podman 이벤트 표시:**

○

모든 **Podman** 이벤트를 표시하려면 다음을 입력합니다.

```
$ podman events
2020-05-14 10:33:42.312377447 -0600 CST container create 34503c192940
(image=registry.access.redhat.com/ubi8/ubi:latest, name=keen_colden)
2020-05-14 10:33:46.958768077 -0600 CST container init 34503c192940
(image=registry.access.redhat.com/ubi8/ubi:latest, name=keen_colden)
2020-05-14 10:33:46.973661968 -0600 CST container start 34503c192940
(image=registry.access.redhat.com/ubi8/ubi:latest, name=keen_colden)
2020-05-14 10:33:50.833761479 -0600 CST container stop 34503c192940
```

```
(image=registry.access.redhat.com/ubi8/ubi:latest, name=keen_colden)
2020-05-14 10:33:51.047104966 -0600 CST container cleanup 34503c192940
(image=registry.access.redhat.com/ubi8/ubi:latest, name=keen_colden)
```

로깅을 종료하려면 **CTRL+c**를 누릅니다.

○

Podman 생성 이벤트만 표시하려면 다음을 입력합니다.

```
$ podman events --filter event=create
2020-05-14 10:36:01.375685062 -0600 CST container create 20dc581f6fbf
(image=registry.access.redhat.com/ubi8/ubi:latest)
2019-03-02 10:36:08.561188337 -0600 CST container create 58e7e002344c
(image=registry.access.redhat.com/ubi8/ubi-minimal:latest)
2019-03-02 10:36:29.978806894 -0600 CST container create d81e30f1310f
(image=registry.access.redhat.com/ubi8/ubi-init:latest)
```

추가 리소스

●

podman-events 도움말 페이지

15장. 컨테이너 체크포인트 생성 및 복원

CRIU(checkpoint/Restore In Userspace)는 실행 중인 컨테이너 또는 개별 애플리케이션에 체크포인트를 설정하고 해당 상태를 디스크에 저장할 수 있는 소프트웨어입니다. 저장된 데이터를 사용하여 검사할 때 동일한 시점에 재부팅한 후 컨테이너를 복원할 수 있습니다.

15.1. 로컬에서 컨테이너 체크포인트 생성 및 복원

이 예제는 각 요청 후에 증가된 단일 정수를 반환하는 **Python** 기반 웹 서버를 기반으로 합니다.

절차

1.

Python 기반 서버를 생성합니다.

```
# cat counter.py
#!/usr/bin/python3

import http.server

counter = 0

class handler(http.server.BaseHTTPRequestHandler):
    def do_GET(s):
        global counter
        s.send_response(200)
        s.send_header('Content-type', 'text/html')
        s.end_headers()
        s.wfile.write(b'%d\n' % counter)
        counter += 1

server = http.server.HTTPServer(('', 8088), handler)
server.serve_forever()
```

2.

다음 정의를 사용하여 컨테이너를 생성합니다.

```
# cat Containerfile
FROM registry.access.redhat.com/ubi8/ubi

COPY counter.py /home/counter.py

RUN useradd -ms /bin/bash counter

RUN yum -y install python3 && chmod 755 /home/counter.py
```

```
USER counter
ENTRYPOINT /home/counter.py
```

컨테이너는 **UBI 8(Universal Base Image)**을 기반으로 하며 **Python** 기반 서버를 사용합니다.

3. 컨테이너를 빌드합니다.

```
# podman build . --tag counter
```

files counter.py 및 **Containerfile** 은 컨테이너 빌드 프로세스(**podman build**)에 대한 입력입니다. 빌드된 이미지는 로컬에 저장되며 태그 카운터로 태그가 지정됩니다.

4. 컨테이너를 **root**로 시작합니다.

```
# podman run --name criu-test --detach counter
```

5. 실행 중인 컨테이너를 모두 나열하려면 다음을 입력합니다.

```
# podman ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
e4f82fd84d48 localhost/counter:latest 5 seconds ago Up 4 seconds ago criu-test
```

6. 컨테이너의 **IP** 주소를 표시합니다.

```
# podman inspect criu-test --format "{{.NetworkSettings.IPAddress}}"
10.88.0.247
```

7. 컨테이너로 요청을 보냅니다.

```
# curl 10.88.0.247:8080
0
# curl 10.88.0.247:8080
1
```

8. 컨테이너의 **checkpoint**를 만듭니다.

```
# podman container checkpoint criu-test
```

9. 시스템을 재부팅합니다.

10. 컨테이너를 복원합니다.

```
# podman container restore --keep criu-test
```

11. 컨테이너로 요청을 보냅니다.

```
# curl 10.88.0.247:8080
2
# curl 10.88.0.247:8080
3
# curl 10.88.0.247:8080
4
```

이제 결과가 0 에서 다시 시작되지 않지만 이전 값을 계속 진행합니다.

이렇게 하면 재부팅을 통해 전체 컨테이너 상태를 쉽게 저장할 수 있습니다.

추가 리소스

- [Podman에 checkpoint/restore 지원 추가](#)

15.2. 컨테이너 복원을 사용하여 시작 시간 단축

컨테이너 마이그레이션을 사용하면 특정 시간이 필요한 컨테이너 시작 시간을 줄일 수 있습니다. **checkpoint**를 사용하면 동일한 호스트 또는 다른 호스트에서 컨테이너를 여러 번 복원할 수 있습니다. 이 예는 컨테이너 [검사점 생성 및 복원의 컨테이너를 로컬에서 기반으로 합니다](#).

절차

1. 컨테이너의 **checkpoint**를 만들고 **checkpoint** 이미지를 **tar.gz** 파일로 내보냅니다.

```
# podman container checkpoint criu-test --export /tmp/chkpt.tar.gz
```

2. **tar.gz** 파일에서 컨테이너를 복원합니다.

```
# podman container restore --import /tmp/chkpt.tar.gz --name counter1
# podman container restore --import /tmp/chkpt.tar.gz --name counter2
# podman container restore --import /tmp/chkpt.tar.gz --name counter3
```

name (-n) 옵션은 내보낸 체크포인트에서 복원된 컨테이너의 새 이름을 지정합니다.

3.

각 컨테이너의 **ID**와 이름을 표시합니다.

```
# podman ps -a --format "{{.ID}} {{.Names}}"
a8b2e50d463c counter3
faabc5c27362 counter2
2ce648af11e5 counter1
```

4.

각 컨테이너의 **IP** 주소를 표시합니다.

```
# podman inspect counter1 --format "{{.NetworkSettings.IPAddress}}"
10.88.0.248

# podman inspect counter2 --format "{{.NetworkSettings.IPAddress}}"
10.88.0.249

# podman inspect counter3 --format "{{.NetworkSettings.IPAddress}}"
10.88.0.250
```

5.

각 컨테이너에 요청을 보냅니다.

```
# curl 10.88.0.248:8080
4
# curl 10.88.0.249:8080
4
# curl 10.88.0.250:8080
4
```

동일한 체크포인트에서 복원된 다른 컨테이너를 사용하여 작업하므로 결과는 모든 경우에 4입니다.

이 방법을 사용하면 처음에 확인된 컨테이너의 상태 저장 복제본을 빠르게 시작할 수 있습니다.

추가 리소스

•

RHEL에서 Podman을 사용한 컨테이너 마이그레이션

15.3. 시스템 간 컨테이너 마이그레이션

다음 절차에서는 컨테이너에서 실행 중인 애플리케이션의 상태를 손실하지 않고 한 시스템에서 다른 시스템으로 실행 중인 컨테이너를 마이그레이션하는 방법을 보여줍니다. 이 예는 컨테이너 [검사점 생성 및 복원의 컨테이너를 로컬에서 기반으로 합니다](#).

카운터 가 지정된 섹션 .

사전 요구 사항

컨테이너를 레지스트리로 푸시하면 로컬에서 사용할 수 없는 경우 레지스트리에서 컨테이너를 자동으로 다운로드하므로 다음 단계는 필요하지 않습니다. 이 예제에서는 레지스트리를 사용하지 않습니다. 이전에 빌드되고 태그가 지정된 컨테이너를 내보내야 합니다([로컬 컨테이너 검사점 생성 및 복원](#) 섹션 참조).

•

이전에 빌드한 컨테이너를 내보냅니다.

```
# podman save --output counter.tar counter
```

•

내보낸 컨테이너 이미지를 대상 시스템(*other_host*)에 복사합니다.

```
# scp counter.tar other_host:
```

•

대상 시스템에서 내보낸 컨테이너를 가져옵니다.

```
# ssh other_host podman load --input counter.tar
```

이제 이 컨테이너 마이그레이션의 대상 시스템에 로컬 컨테이너 스토리지에 저장된 동일한 컨테이너 이미지가 있습니다.

절차

1.

컨테이너를 **root**로 시작합니다.

```
# podman run --name criu-test --detach counter
```

2.

컨테이너의 **IP** 주소를 표시합니다.

```
# podman inspect criu-test --format "{{.NetworkSettings.IPAddress}}"
10.88.0.247
```

3.

컨테이너로 요청을 보냅니다.

```
# curl 10.88.0.247:8080
0
# curl 10.88.0.247:8080
1
```

4.

컨테이너의 **checkpoint**를 만들고 **checkpoint** 이미지를 **tar.gz** 파일로 내보냅니다.

```
# podman container checkpoint criu-test --export /tmp/chkpt.tar.gz
```

5.

checkpoint 아카이브를 대상 호스트에 복사합니다.

```
# scp /tmp/chkpt.tar.gz other_host:/tmp/
```

6.

대상 호스트 (**other_host**)에서 **checkpoint**를 복원합니다.

```
# podman container restore --import /tmp/chkpt.tar.gz
```

7.

대상 호스트의 컨테이너(**other_host**)로 요청을 보냅니다.

```
# curl 10.88.0.247:8080
2
```

그 결과 **stateful** 컨테이너가 상태를 손실하지 않고 한 시스템에서 다른 시스템으로 마이그레이션되었습니다.

추가 리소스

-

[RHEL에서 Podman을 사용한 컨테이너 마이그레이션](#)

16장. HPC 환경에서 PODMAN 사용

Podman을 Open MPI(Message Passing Interface)와 함께 사용하여 HPC(고성능 컴퓨팅) 환경에서 컨테이너를 실행할 수 있습니다.

16.1. MPI로 PODMAN 사용

이 예제는 Open MPI에서 가져온 [ring.c](#) 프로그램을 기반으로 합니다. 이 예에서 값은 링과 유사한 방식으로 모든 프로세스에 전달됩니다. 메시지가 순위 0을 통과할 때마다 값이 감소됩니다. 각 프로세스에 0 메시지가 수신되면 해당 메시지를 다음 프로세스로 전달한 다음 종료합니다. 0을 먼저 전달하면 모든 프로세스가 0 메시지를 가져오고 정상적으로 종료될 수 있습니다.

절차

1. 열기 MPI를 설치합니다.

```
$ sudo yum install openmpi
```

2. 환경 모듈을 활성화하려면 다음을 입력합니다.

```
$ . /etc/profile.d/modules.sh
```

3. mpi/openmpi-x86_64 모듈을 로드합니다.

```
$ module load mpi/openmpi-x86_64
```

선택적으로 mpi/openmpi-x86_64 모듈을 자동으로 로드하려면 다음 행을 .bashrc 파일에 추가합니다.

```
$ echo "module load mpi/openmpi-x86_64" >> .bashrc
```

4. mpirun 과 podman 을 결합하려면 다음 정의를 사용하여 컨테이너를 생성합니다.

```
$ cat Containerfile
FROM registry.access.redhat.com/ubi8/ubi

RUN yum -y install openmpi-devel wget && \
    yum clean all
```

```
RUN wget https://raw.githubusercontent.com/open-mpi/ompi/master/test/simple/ring.c
&& \
  /usr/lib64/openmpi/bin/mpicc ring.c -o /home/ring && \
  rm -f ring.c
```

5. 컨테이너를 빌드합니다.

```
$ podman build --tag=mpi-ring .
```

6. 컨테이너를 시작합니다. CPU 4개가 있는 시스템에서 이 명령은 컨테이너 4개를 시작합니다.

```
$ mpirun \
  --mca orte_tmpdir_base /tmp/podman-mpirun \
  podman run --env-host \
  -v /tmp/podman-mpirun:/tmp/podman-mpirun \
  --userns=keep-id \
  --net=host --pid=host --ipc=host \
  mpi-ring /home/ring
Rank 2 has cleared MPI_Init
Rank 2 has completed ring
Rank 2 has completed MPI_Barrier
Rank 3 has cleared MPI_Init
Rank 3 has completed ring
Rank 3 has completed MPI_Barrier
Rank 1 has cleared MPI_Init
Rank 1 has completed ring
Rank 1 has completed MPI_Barrier
Rank 0 has cleared MPI_Init
Rank 0 has completed ring
Rank 0 has completed MPI_Barrier
```

결과적으로 **mpirun** 은 4개의 **Podman** 컨테이너를 시작하고 각 컨테이너에서 링 바이너리 인스턴스 하나를 실행합니다. 4개의 프로세스는 모두 **MPI**를 통해 서로 통신합니다.

추가 리소스

- [HPC 환경의 podman](#)

16.2. MPIRUN 옵션

다음 **mpirun** 옵션을 사용하여 컨테이너를 시작합니다.

- **--mCA orte_tmpdir_base /tmp/podman-mpirun** 행은 Open MPI에 /tmp/podman-mpirun

이 아닌 **/tmp/podman-mpirun** 에 모든 임시 파일을 생성하도록 지시합니다. 둘 이상의 노드를 사용하는 경우 이 디렉터리의 이름은 다른 노드에서 다르게 지정됩니다. 이를 위해서는 전체 **/tmp** 디렉터리를 보다 복잡한 컨테이너에 마운트해야 합니다.

mpirun 명령은 시작할 명령인 **podman** 명령을 지정합니다. 다음 **podman** 옵션을 사용하여 컨테이너를 시작합니다.

- 명령을 실행하여 컨테이너를 실행합니다.
- **--env-host** 옵션은 호스트의 모든 환경 변수를 컨테이너에 복사합니다.
- **-v /tmp/podman-mpirun:/tmp/podman-mpirun** 행은 Podman에 Open MPI가 컨테이너에서 사용할 수 있는 임시 디렉터리와 파일을 생성하는 위치를 마운트하도록 지시합니다.
- **--userns=keep-id** 행은 컨테이너 내부와 외부의 사용자 ID 매핑을 확인합니다.
- **--net=host --pid=host --ipc=host** 행은 동일한 네트워크, PID 및 IPC 네임스페이스를 설정합니다.
- **MPI-ring** 은 컨테이너의 이름입니다.
- **/home/ring** 은 컨테이너의 MPI 프로그램입니다.

추가 리소스

- [HPC 환경의 podman](#)

17장. 특수 컨테이너 이미지 실행

이 장에서는 컨테이너 이미지의 몇 가지 특수 유형에 대해 설명합니다. 일부 컨테이너 이미지에는 미리 설정된 옵션 및 인수를 사용하여 해당 컨테이너를 실행할 수 있는 **runlabels** 라는 레이블이 빌드되어 있습니다. **podman container runlabel <label>** 명령을 사용하면 컨테이너 이미지의 <label>에 정의된 명령을 실행할 수 있습니다. 지원되는 레이블은 설치, 실행 및 제거입니다.

17.1. 호스트에 대한 권한 열기

권한이 있는 컨테이너와 권한이 없는 컨테이너 간에는 몇 가지 차이점이 있습니다. 예를 들어 **toolbox** 컨테이너는 권한 있는 컨테이너입니다. 다음은 컨테이너에서 호스트에 열거나 열지 않을 수 있는 권한의 예입니다.

- **권한:** 권한 있는 컨테이너는 호스트에서 컨테이너를 분리하는 보안 기능을 비활성화합니다. **podman run --privileged <image_name>** 명령을 사용하여 권한 있는 컨테이너를 실행할 수 있습니다. 예를 들어 **root** 사용자가 소유한 호스트에서 마운트된 파일과 디렉토리를 삭제할 수 있습니다.
- **프로세스 테이블:** **podman run --pid=host <image_name>** 명령을 사용하여 컨테이너에 대한 호스트 **PID** 네임스페이스를 사용할 수 있습니다. 그런 다음 권한 있는 컨테이너 내에서 **ps -e** 명령을 사용하여 호스트에서 실행 중인 모든 프로세스를 나열할 수 있습니다. 호스트의 프로세스 **ID**를 권한 있는 컨테이너에서 실행되는 명령으로 전달할 수 있습니다(예: **kill <PID>**).
- **네트워크 인터페이스:** 기본적으로 컨테이너에는 외부 네트워크 인터페이스 1개와 루프백 네트워크 인터페이스가 1개뿐입니다. **podman run --net=host <image_name>** 명령을 사용하여 컨테이너 내에서 직접 호스트 네트워크 인터페이스에 액세스할 수 있습니다.
- **프로세스 간 통신:** 호스트의 **IPC** 기능은 권한 있는 컨테이너 내에서 액세스할 수 있습니다. **ipcs**와 같은 명령을 실행하여 호스트에서 활성 메시지 대기열, 공유 메모리 세그먼트 및 세마포어 세트에 대한 정보를 확인할 수 있습니다.

17.2. RUNLABELS가 있는 컨테이너 이미지

일부 **Red Hat** 이미지에는 해당 이미지 작업을 위해 사전 설정된 명령줄을 제공하는 레이블이 포함되어 있습니다. **podman container runlabel <label>** 명령을 사용하여 이미지의 <label>에 정의된 명령을 실행할 수 있습니다.

기존 **runlabels**는 다음과 같습니다.

- **install:** 이미지를 실행하기 전에 호스트 시스템을 설정합니다. 그러면 일반적으로 컨테이너가 나중에 실행될 때 액세스할 수 있는 호스트에 파일과 디렉터리가 생성됩니다.
- **run:** 컨테이너를 실행할 때 사용할 **podman** 명령줄 옵션을 식별합니다. 일반적으로 옵션은 호스트에 권한을 열고 컨테이너에서 호스트에서 영구적으로 유지해야 하는 호스트 콘텐츠를 마운트합니다.
- **uninstall:** 컨테이너 실행을 마친 후 호스트 시스템을 정리합니다.

17.3. RUNLABELS로 RSYSLOG 실행

rsyslogd 데몬의 컨테이너화된 버전을 실행하도록 **rhel8/rsyslog** 컨테이너 이미지가 생성됩니다. **rsyslog** 이미지에는 다음 **runlabels**가 포함되어 있습니다. **install**, **run** 및 **uninstall**입니다. 다음 절차에서는 **rsyslog** 이미지를 설치, 실행 및 제거하는 단계를 수행합니다.

절차

1. **rsyslog** 이미지를 가져옵니다.

```
# podman pull registry.redhat.io/rhel{ProductNumberLink}/rsyslog
```

2. **rsyslog**에 대한 설치 **runlabel** 표시 :

```
# podman container runlabel install --display rhel{ProductNumberLink}/rsyslog
command: podman run --rm --privileged -v /:/host -e HOST=/host -e
IMAGE=registry.redhat.io/rhel{ProductNumberLink}/rsyslog:latest -e NAME=rsyslog
registry.redhat.io/rhel{ProductNumberLink}/rsyslog:latest /bin/install.sh
```

이는 명령이 호스트에 대한 권한을 열고 컨테이너의 **/host**에 호스트 루트 파일 시스템을 마운트하며 **install.sh** 스크립트를 실행하는 것을 보여줍니다.

3. **rsyslog**에 대한 설치 **runlabel**을 실행합니다 :

```
# podman container runlabel install rhel{ProductNumberLink}/rsyslog
command: podman run --rm --privileged -v /:/host -e HOST=/host -e
IMAGE=registry.redhat.io/rhel{ProductNumberLink}/rsyslog:latest -e NAME=rsyslog
registry.redhat.io/rhel{ProductNumberLink}/rsyslog:latest /bin/install.sh
Creating directory at /host/etc/pki/rsyslog
Creating directory at /host/etc/rsyslog.d
```

```
Installing file at /host/etc/rsyslog.conf
Installing file at /host/etc/sysconfig/rsyslog
Installing file at /host/etc/logrotate.d/syslog
```

이렇게 하면 **rsyslog** 이미지가 나중에 사용할 파일을 호스트 시스템에 생성합니다.

4.

rsyslog 의 **run label**을 표시합니다 :

```
# podman container runlabel run --display rhel{ProductNumberLink}/rsyslog
command: podman run -d --privileged --name rsyslog --net=host --pid=host -v
/etc/pki/rsyslog:/etc/pki/rsyslog -v /etc/rsyslog.conf:/etc/rsyslog.conf -v
/etc/sysconfig/rsyslog:/etc/sysconfig/rsyslog -v /etc/rsyslog.d:/etc/rsyslog.d -v /var/log:/var/log
-v /var/lib/rsyslog:/var/lib/rsyslog -v /run:/run -v /etc/machine-id:/etc/machine-id -v
/etc/localtime:/etc/localtime -e
IMAGE=registry.redhat.io/rhel{ProductNumberLink}/rsyslog:latest -e NAME=rsyslog --
restart=always registry.redhat.io/rhel{ProductNumberLink}/rsyslog:latest /bin/rsyslog.sh
```

이는 명령이 호스트에 대한 권한을 열고 **rsyslog d** 데몬을 실행하기 위해 **rsyslog** 컨테이너를 시작할 때 컨테이너 내부의 호스트에서 특정 파일 및 디렉토리를 마운트함을 나타냅니다.

5.

rsyslog 에 대해 **runlabel**을 실행합니다 :

```
# podman container runlabel run rhel{ProductNumberLink}/rsyslog
command: podman run -d --privileged --name rsyslog --net=host --pid=host -v
/etc/pki/rsyslog:/etc/pki/rsyslog -v /etc/rsyslog.conf:/etc/rsyslog.conf -v
/etc/sysconfig/rsyslog:/etc/sysconfig/rsyslog -v /etc/rsyslog.d:/etc/rsyslog.d -v /var/log:/var/log
-v /var/lib/rsyslog:/var/lib/rsyslog -v /run:/run -v /etc/machine-id:/etc/machine-id -v
/etc/localtime:/etc/localtime -e
IMAGE=registry.redhat.io/rhel{ProductNumberLink}/rsyslog:latest -e NAME=rsyslog --
restart=always registry.redhat.io/rhel{ProductNumberLink}/rsyslog:latest /bin/rsyslog.sh
28a0d719ff179adcea81eb63cc90fcd09f1755d5edb121399068a4ea59bd0f53
```

rsyslog 컨테이너는 권한을 열고 호스트에서 필요한 사항을 마운트하며, **rsyslogd** 데몬을 백그라운드(-d)에서 실행합니다. **rsyslogd** 데몬은 로그 메시지 수집 및 **/var/log** 디렉토리의 파일에 메시지를 보냅니다.

6.

rsyslog 에 대한 제거 **runlabel**을 표시 :

```
# podman container runlabel uninstall --display rhel{ProductNumberLink}/rsyslog
command: podman run --rm --privileged -v /:/host -e HOST=/host -e
IMAGE=registry.redhat.io/rhel{ProductNumberLink}/rsyslog:latest -e NAME=rsyslog
registry.redhat.io/rhel{ProductNumberLink}/rsyslog:latest /bin/uninstall.sh
```

7.

rsyslog 에 대한 제거 **runlabel**을 실행합니다 :

```
# podman container runlabel uninstall rhel{ProductNumberLink}/rsyslog
command: podman run --rm --privileged -v /:/host -e HOST=/host -e
IMAGE=registry.redhat.io/rhel{ProductNumberLink}/rsyslog:latest -e NAME=rsyslog
registry.redhat.io/rhel{ProductNumberLink}/rsyslog:latest /bin/uninstall.sh
```



참고

이 경우 **uninstall.sh** 스크립트는 **/etc/logrotate.d/syslog** 파일을 제거합니다. 구성 파일을 정리하지 않습니다.

18장. CONTAINER-TOOLS API 사용

새로운 **REST** 기반 **Podman 2.0 API**는 **varlink** 라이브러리를 사용한 **Podman**의 이전 원격 **API**를 대체합니다. 새 **API**는 **rootful** 환경과 **rootless** 환경에서 모두 작동합니다.

Podman v2.0 RESTful API는 **Podman** 및 **Docker** 호환 **API**를 지원하는 **Libpod API**로 구성됩니다. 이 새로운 **REST API**를 사용하면 **cURL**, **Postman**, **Google**의 고급 **REST** 클라이언트 등의 플랫폼에서 **Podman**을 호출할 수 있습니다.

18.1. ROOT 모드에서 SYSTEMD를 사용하여 PODMAN API 활성화

다음 절차에서는 다음을 수행하는 방법을 설명합니다.

1. **systemd**를 사용하여 **Podman API** 소켓을 활성화합니다.
2. **Podman** 클라이언트를 사용하여 기본 명령을 수행합니다.

전제 조건

- **podman-remote** 패키지가 설치되어 있습니다.

```
# yum install podman-remote
```

절차

1. 즉시 서비스를 시작합니다.
- ```
systemctl enable --now podman.socket
```
2. **docker-podman** 패키지를 사용하여 **var/lib/docker.sock**에 대한 링크를 활성화하려면 다음을 수행합니다.

```
yum install podman-docker
```

#### 검증 단계

1. **Podman의 시스템 정보를 표시합니다.**

```
podman-remote info
```

2. **링크를 확인합니다.**

```
ls -al /var/run/docker.sock
lrwxrwxrwx. 1 root root 23 Nov 4 10:19 /var/run/docker.sock -> /run/podman/podman.sock
```

#### 추가 리소스

- [podman v2.0 RESTful API](#)
- [첫 번째 조회 Podman 2.0 API](#)
- [스니크 미리보기: podman의 새로운 REST API](#)

## 18.2. ROOTLESS 모드에서 SYSTEMD를 사용하여 PODMAN API 활성화

다음 절차에서는 **systemd**를 사용하여 **Podman API** 소켓 및 **podman API** 서비스를 활성화하는 방법을 설명합니다.

#### 사전 요구 사항

- **podman-remote** 패키지가 설치되어 있습니다.

```
yum install podman-remote
```

#### 절차

1. **서비스를 즉시 활성화하고 시작합니다.**

```
$ systemctl --user enable --now podman.socket
```

2. **선택 사항: Docker를 사용하여 rootless Podman 소켓과 상호 작용하는 프로그램을 활성화하려면 다음을 수행합니다.**

```
$ export DOCKER_HOST=unix:///run/user/<uid>/podman/podman.sock
```

## 검증 단계

1.

소켓 상태를 확인합니다.

```
$ systemctl --user status podman.socket
● podman.socket - Podman API Socket
 Loaded: loaded (/usr/lib/systemd/user/podman.socket; enabled; vendor preset:
 enabled)
 Active: active (listening) since Mon 2021-08-23 10:37:25 CEST; 9min ago
 Docs: man:podman-system-service(1)
 Listen: /run/user/1000/podman/podman.sock (Stream)
 CGroup: /user.slice/user-1000.slice/user@1000.service/podman.socket
```

**podman.socket** 이 활성화되어 있으며 **/run/user/<uid>/podman.podman.sock** 에서 수신 대기 중입니다. 여기서 **<uid>** 는 사용자의 ID입니다.

2.

Podman의 시스템 정보를 표시합니다.

```
$ podman-remote info
```

## 추가 리소스

- [podman v2.0 RESTful API](#)
- [첫 번째 조회 Podman 2.0 API](#)
- [스니크 미리보기: podman의 새로운 REST API](#)
- [Python 및 Bash를 사용하여 Podman RESTful API 탐색](#)

## 18.3. PODMAN API 수동 실행

다음 절차에서는 **Podman API** 실행 방법을 설명합니다. 이는 특히 **Docker** 호환성 계층을 사용할 때 **API** 호출을 디버깅하는 데 유용합니다.

## 전제 조건

- **podman-remote** 패키지가 설치되어 있습니다.

```
yum install podman-remote
```

## 절차

1. **REST API**에 대한 서비스를 실행합니다.

```
podman system service -t 0 --log-level=debug
```

- 값 0은 시간 초과를 의미하지 않습니다. **rootful** 서비스의 기본 끝점은 **unix:/run/podman/podman.sock** 입니다.
- **log -level <level>** 옵션은 로깅 수준을 설정합니다. 표준 로깅 수준은 **debug,info,warn,error,fatal, panic** 입니다.

2. 다른 터미널에서 **Podman**의 시스템 정보를 표시합니다. **podman-remote** 명령은 일반 **podman** 명령과 달리 **Podman** 소켓을 통해 통신합니다.

```
podman-remote info
```

3. **Podman API**의 문제를 해결하고 요청 및 응답을 표시하려면 **curl command**을 사용합니다. **JSON** 형식의 **Linux** 서버에 **Podman** 설치에 대한 정보를 가져오려면 다음을 수행합니다.

```
curl -s --unix-socket /run/podman/podman.sock http://d/v1.0.0/libpod/info | jq
{
 "host": {
 "arch": "amd64",
 "buildahVersion": "1.15.0",
 "cgroupVersion": "v1",
 "conmon": {
 "package": "conmon-2.0.18-1.module+el8.3.0+7084+c16098dd.x86_64",
 "path": "/usr/bin/conmon",
 "version": "conmon version 2.0.18, commit:
7fd3f71a218f8d3a7202e464252aeb1e942d17eb"
 },
 ...
 },
 "version": {
 "APIVersion": 1,
 "Version": "2.0.0",
 "GoVersion": "go1.14.2",
```

```
"GitCommit": "",
"BuiltTime": "Thu Jan 1 01:00:00 1970",
"Built": 0,
"OsArch": "linux/amd64"
}
}
```

jq 유틸리티는 명령줄 JSON 프로세서입니다.

4.

registry.access.redhat.com/ubi8/ubi 컨테이너 이미지를 가져옵니다.

```
curl -XPOST --unix-socket /run/podman/podman.sock -v
'http://d/v1.0.0/images/create?fromImage=registry.access.redhat.com%2Fubi8%2Fubi'
* Trying /run/podman/podman.sock...
* Connected to d (/run/podman/podman.sock) port 80 (#0)
> POST /v1.0.0/images/create?fromImage=registry.access.redhat.com%2Fubi8%2Fubi
HTTP/1.1
> Host: d
> User-Agent: curl/7.61.1
> Accept: /
>
< HTTP/1.1 200 OK
< Content-Type: application/json
< Date: Tue, 20 Oct 2020 13:58:37 GMT
< Content-Length: 231
<
{"status":"pulling image () from registry.access.redhat.com/ubi8/ubi:latest,
registry.redhat.io/ubi8/ubi:latest","error":"","progress":"","progressDetail":
{},"id":"ecbc6f53bba0d1923ca9e92b3f747da8353a070fccbae93625bd8b47dbee772e"}
* Connection #0 to host d left intact
```

5.

가져온 이미지를 표시합니다.

```
curl --unix-socket /run/podman/podman.sock -v 'http://d/v1.0.0/libpod/images/json' |
jq
* Trying /run/podman/podman.sock...
% Total % Received % Xferd Average Speed Time Time Time Current
Dload Upload Total Spent Left Speed
0 0 0 0 0 0 0 0 --:--:-- --:--:-- --:--:-- 0* Connected to d
(/run/podman/podman.sock) port 80 (0) > GET /v1.0.0/libpod/images/json HTTP/1.1 >
Host: d > User-Agent: curl/7.61.1 > Accept: / > < HTTP/1.1 200 OK < Content-Type:
application/json < Date: Tue, 20 Oct 2020 13:59:55 GMT < Transfer-Encoding: chunked
< { [12498 bytes data] 100 12485 0 12485 0 0 2032k 0 --:--:-- --:--:-- --:--:-- 2438k *
Connection #0 to host d left intact [{ "Id":
"ecbc6f53bba0d1923ca9e92b3f747da8353a070fccbae93625bd8b47dbee772e",
"RepoTags": ["registry.access.redhat.com/ubi8/ubi:latest",
"registry.redhat.io/ubi8/ubi:latest"], "Created": "2020-09-01T19:44:12.470032Z",
"Size": 210838671, "Labels": { "architecture": "x86_64", "build-date": "2020-09-
01T19:43:46.041620", "com.redhat.build-host": "cpt-
1008.osbs.prod.upshift.rdu2.redhat.com", ... "maintainer": "Red Hat, Inc.", "name":
```

```

"ubi8", ... "summary": "Provides the latest release of Red Hat Universal Base Image
8.", "url":
"https://access.redhat.com/containers//registry.access.redhat.com/ubi8/images/8.2-
347",
...
},
"Names": [
 "registry.access.redhat.com/ubi8/ubi:latest",
 "registry.redhat.io/ubi8/ubi:latest"
],
...
]
}
]

```

#### 추가 리소스

- [podman v2.0 RESTful API](#)
- [스니크 미리보기: podman의 새로운 REST API](#)
- [Python 및 Bash를 사용하여 Podman RESTful API 탐색](#)
- [podman-system-service 도움말 페이지](#)