



Red Hat Enterprise Linux 9

コンテナの構築、実行、および管理

Red Hat Enterprise Linux 9 での Podman、Buildah、および Skopeo の使用

Red Hat Enterprise Linux 9 コンテナの構築、実行、および管理

Red Hat Enterprise Linux 9 での Podman、Buildah、および Skopeo の使用

法律上の通知

Copyright © 2024 Red Hat, Inc.

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.

概要

Red Hat Enterprise Linux 9 は、コンテナイメージを使用するコマンドラインツールを多数提供しています。Podman を使用して Pod およびコンテナイメージを管理できます。コンテナイメージをビルド、更新、および管理するには、Buildah を使用できます。リモートリポジトリーでイメージをコピーして検査するには、Skopeo を使用します。

目次

RED HAT ドキュメントへのフィードバック (英語のみ)	7
第1章 コンテナを使い始める	8
1.1. PODMAN、BUILDDAH、および SKOPEO の特徴	8
1.2. 一般的な PODMAN コマンド	9
1.3. DOCKER を使用せずにコンテナを実行	12
1.4. コンテナの RHEL アーキテクチャーの選択	12
1.5. コンテナツールの取得	12
1.6. ルートレスコンテナの設定	13
1.7. ルートレスコンテナへのアップグレード	15
1.8. ルートレスコンテナに関する特別な考慮事項	15
1.9. 高度な PODMAN 設定のためのモジュールの使用	16
1.10. 関連情報	17
第2章 コンテナイメージの種類	18
2.1. RHEL コンテナイメージの一般的な特徴	18
2.2. UBI イメージの特徴	18
2.3. UBI 標準イメージの概要	19
2.4. UBI INIT イメージの概要	20
2.5. UBI MINIMAL イメージの概要	20
2.6. UBI マイクロイメージの概要	21
第3章 コンテナレジストリーの使用	22
3.1. コンテナレジストリー	22
3.2. コンテナレジストリーの設定	23
3.3. コンテナイメージの検索	24
3.4. レジストリーからのイメージの取得 (プル)	25
3.5. 短縮名のエイリアスの設定	26
第4章 コンテナイメージの使用	28
4.1. 短縮名のエイリアスを使用したコンテナイメージのプル	28
4.2. イメージのリスト表示	29
4.3. ローカルイメージの検証	29
4.4. リモートイメージの検証	30
4.5. コンテナイメージのコピー	31
4.6. ローカルディレクトリーへのイメージレイヤーのコピー	31
4.7. イメージのタグ付け	32
4.8. イメージの保存および読み込み	33
4.9. UBI イメージの再配布	34
4.10. イメージの削除	35
第5章 コンテナの使用	37
5.1. PODMAN RUN コマンド	37
5.2. ホストからのコンテナでのコマンド実行	37
5.3. コンテナ内でのコマンドの実行	38
5.4. コンテナのリスト表示	39
5.5. コンテナの起動	40
5.6. ホストからのコンテナの検証	40
5.7. LOCALHOST のディレクトリーのコンテナへのマウント	41
5.8. コンテナのファイルシステムのマウント	42
5.9. 静的 IP を使用したデーモンとしてのサービスの実行	43
5.10. 実行中のコンテナ内でのコマンドの実行	43

5.11. 2つのコンテナ間でのファイルの共有	45
5.12. コンテナのエクスポートおよびインポート	47
5.13. コンテナの停止	49
5.14. コンテナの削除	50
5.15. コンテナのSELINUXポリシーの作成	51
5.16. PODMANでの事前実行フックの設定	51
5.17. コンテナ内のアプリケーションのデバッグ	52
第6章 コンテナランタイムの選択	53
6.1. RUNC コンテナランタイム	53
6.2. CRUN コンテナランタイム	53
6.3. RUNC および CRUN でのコンテナの実行	53
6.4. コンテナランタイムの一時的な変更	55
6.5. コンテナランタイムの永続的な変更	55
第7章 UBI コンテナへのソフトウェアの追加	57
7.1. UBI INIT イメージの使用	57
7.2. UBI マイクロイメージの使用	58
7.3. UBI コンテナへのソフトウェアの追加 (サブスクライブされたホスト)	59
7.4. 標準の UBI コンテナへのソフトウェアの追加	60
7.5. UBI MINIMAL コンテナへのソフトウェアの追加	61
7.6. UBI コンテナへのソフトウェアの追加 (サブスクライブしていないホスト)	62
7.7. UBI ベースのイメージの構築	62
7.8. APPLICATION STREAM ランタイムイメージの使用	64
7.9. UBI コンテナイメージソースコードの取得	64
第8章 コンテナイメージへの署名	67
8.1. GPG 署名を使用したコンテナイメージへの署名	67
8.2. GPG イメージ署名の検証	68
8.3. 秘密鍵を使用した SIGSTORE 署名によるコンテナイメージの署名	70
8.4. 公開鍵を使用した SIGSTORE イメージ署名の検証	72
8.5. FULCIO と REKOR を使用した SIGSTORE 署名によるコンテナイメージの署名	73
8.6. FULCIO と REKOR を使用した SIGSTORE 署名を持つコンテナイメージの検証	74
8.7. 秘密鍵と REKOR を使用した SIGSTORE 署名によるコンテナイメージの署名	76
第9章 コンテナネットワークの管理	79
9.1. コンテナネットワークのリストアップ	79
9.2. ネットワークの検査	79
9.3. ネットワークの作成	80
9.4. コンテナのネットワークへの接続	81
9.5. コンテナのネットワークからの切断	82
9.6. ネットワークの削除	82
9.7. 未使用の全ネットワークの削除	83
第10章 POD の使用	85
10.1. POD の作成	85
10.2. POD 情報の表示	86
10.3. POD の停止	88
10.4. POD の削除	88
第11章 コンテナ間の通信	90
11.1. ネットワークモードとレイヤー	90
11.2. SLIRP4NETNS と PASTA の違い	90
11.3. ネットワークモードの設定	91
11.4. コンテナのネットワーク設定の検査	91

11.5. コンテナとアプリケーション間の通信	92
11.6. コンテナとホスト間の通信	93
11.7. ポートマッピングによるコンテナ間の通信	93
11.8. DNS を利用したコンテナ間の通信	95
11.9. POD 内の 2 つのコンテナ間での通信	96
11.10. POD での通信	96
11.11. コンテナネットワークへの POD のアタッチ	97
第12章 コンテナネットワークモードの設定	99
12.1. 静的 IP でのコンテナ実行	99
12.2. SYSTEMD なしでの DHCP プラグイン実行	99
12.3. SYSTEMD を使用した DHCP プラグインの実行	100
12.4. MACVLAN プラグイン	101
12.5. ネットワークスタックの CNI から NETAVARK への切り替え	102
12.6. NETAVARK から CNI へのネットワークスタックの切り替え	103
第13章 PODMAN を使用した OPENSIFT へのコンテナの移植	105
13.1. PODMAN を使用した KUBERNETES YAML ファイルの生成	105
13.2. OPENSIFT 環境での KUBERNETES YAML ファイルの生成	107
13.3. PODMAN でのコンテナおよび POD の起動	107
13.4. OPENSIFT 環境でのコンテナおよび POD の起動	108
13.5. PODMAN を使用したコンテナと POD の手動実行	108
13.6. PODMAN を使用した YAML ファイルの生成	110
13.7. PODMAN を使用したコンテナと POD の自動実行	112
13.8. PODMAN を使用した POD の自動停止/削除	114
第14章 PODMAN を使用した SYSTEMD へのコンテナの移植	116
14.1. QUADLET を使用した SYSTEMD ユニットファイルの自動生成	116
14.2. SYSTEMD サービスの有効化	118
14.3. SYSTEMD を使用したコンテナの自動起動	118
14.4. PODMAN GENERATED SYSTEMD コマンドではなく QUADLET を使用する利点	120
14.5. PODMAN を使用した SYSTEMD ユニットファイルの生成	121
14.6. PODMAN を使用した SYSTEMD ユニットファイルの自動生成	123
14.7. SYSTEMD を使用した POD の自動起動	125
14.8. PODMAN を使用したコンテナの自動更新	129
14.9. SYSTEMD を使用したコンテナの自動更新	130
第15章 ANSIBLE PLAYBOOK を使用したコンテナの管理	132
15.1. バインドマウントを使用したルートレスコンテナの作成	132
15.2. PODMAN ボリュームを使用した ROOTFUL コンテナの作成	134
15.3. シークレットを使用した QUADLET アプリケーションの作成	135
第16章 RHEL WEB コンソールを使用したコンテナイメージの管理	139
16.1. WEB コンソールでのコンテナイメージの取得	139
16.2. WEB コンソールでのコンテナイメージのプルーニング	139
16.3. WEB コンソールでのコンテナイメージの削除	140
第17章 RHEL WEB コンソールを使用したコンテナの管理	141
17.1. WEB コンソールでのコンテナの作成	141
17.2. WEB コンソールでのコンテナの検査	143
17.3. WEB コンソールでのコンテナの状態の変更	143
17.4. WEB コンソールでのコンテナのコミット	144
17.5. WEB コンソールでのコンテナチェックポイントの作成	145
17.6. WEB コンソールでのコンテナチェックポイントの復元	146
17.7. WEB コンソールでのコンテナの削除	147

17.8. WEB コンソールでの POD の作成	147
17.9. WEB コンソールの POD 内にコンテナを作成する	148
17.10. WEB コンソールでの POD の状態の変更	150
17.11. WEB コンソールでの POD の削除	150
第18章 コンテナでの SKOPEO、BUILDDAH、および PODMAN の実行	152
18.1. コンテナでの SKOPEO の実行	152
18.2. 認証情報を使用したコンテナでの SKOPEO の実行	153
18.3. AUTHFILE を使用したコンテナでの SKOPEO の実行	154
18.4. ホストに対するコンテナイメージのコピー	155
18.5. コンテナでの BUILDDAH の実行	156
18.6. 特権および非特権 PODMAN コンテナ	157
18.7. 拡張された権限での PODMAN の実行	157
18.8. 少ない権限での PODMAN の実行	158
18.9. PODMAN コンテナ内でのコンテナのビルド	159
第19章 BUILDDAH でコンテナイメージの構築	161
19.1. BUILDDAH ツール	161
19.2. BUILDDAH のインストール	161
19.3. BUILDDAH でイメージの取得	162
19.4. BUILDDAH を使用した CONTAINERFILE からのイメージのビルド	163
19.5. BUILDDAH でゼロからイメージを新規作成	164
19.6. BUILDDAH でイメージの削除	166
第20章 BUILDDAH を使用したコンテナの操作	167
20.1. コンテナ内でのコマンドの実行	167
20.2. BUILDDAH でコンテナおよびイメージの検証	167
20.3. BUILDDAH MOUNT を使用したコンテナの変更	168
20.4. BUILDDAH COPY および BUILDDAH CONFIG を使用したコンテナの変更	170
20.5. プライベートレジストリーへのコンテナのプッシュ	171
20.6. DOCKER HUB へのコンテナのプッシュ	172
20.7. BUILDDAH でコンテナの削除	173
第21章 コンテナの監視	174
21.1. コンテナでヘルスチェックを使用する	174
21.2. コマンドラインを使用してヘルスチェックを実行する	175
21.3. CONTAINERFILE を使用してヘルスチェックを実行する	176
21.4. PODMAN システム情報の表示	178
21.5. PODMAN イベントタイプ	181
21.6. PODMAN イベントのモニタリング	183
21.7. PODMAN イベントを使用した監査	185
第22章 コンテナチェックポイントの作成および復元	187
22.1. ローカルでのコンテナチェックポイントの作成および復元	187
22.2. コンテナ復元を使用した起動時間の短縮	189
22.3. システム間のコンテナの移行	190
第23章 開発とトラブルシューティングに TOOLBX を使用する	193
23.1. TOOLBX コンテナの起動	193
23.2. 開発に TOOLBX を使用する	194
23.3. ホストシステムのトラブルシューティングに TOOLBX を使用する	194
23.4. TOOLBX コンテナの停止	195
第24章 HPC 環境での PODMAN の使用	197
24.1. PODMAN と MPI の使用	197

24.2. MPIRUN オプション	198
第25章 特殊なコンテナイメージの実行	200
25.1. ホストへの権限の付与	200
25.2. RUNLABEL が組み込まれたコンテナイメージ	200
25.3. RUNLABEL での RSYSLOG の実行	200
第26章 CONTAINER-TOOLS API の使用	203
26.1. ROOT モードで SYSTEMD を使用した PODMAN API の有効化	203
26.2. ルートレスモードで SYSTEMD を使用した PODMAN API の有効化	204
26.3. PODMAN API の手動実行	205

RED HAT ドキュメントへのフィードバック (英語のみ)

Red Hat ドキュメントに関するご意見や感想をお寄せください。また、改善点があればお知らせください。

Jira からのフィードバック送信 (アカウントが必要)

1. [Jira](#) の Web サイトにログインします。
2. 上部のナビゲーションバーで **Create** をクリックします。
3. **Summary** フィールドにわかりやすいタイトルを入力します。
4. **Description** フィールドに、ドキュメントの改善に関するご意見を記入してください。ドキュメントの該当部分へのリンクも追加してください。
5. ダイアログの下部にある **Create** をクリックします。

第1章 コンテナを使い始める

Linux コンテナは、イメージベースのデプロイメント方法の柔軟性と、軽量アプリケーションの分離を組み合わせた、主要なオープンソースアプリケーションをパッケージ化して、配信するテクノロジーとして登場しました。Red Hat Enterprise Linux は、以下のようなコア技術を使用して Linux コンテナを実装します。

- リソース管理用のコントロールグループ (cgroup)
- プロセス分離用の namespace
- SELinux (セキュリティー用)
- セキュアなマルチテナンシー

このような技術は、セキュリティーエクスプロイトの可能性を軽減し、エンタープライズ品質のコンテナを生成および実行する環境を提供します。

Red Hat OpenShift は、強力なコマンドラインと Web UI ツールを提供し、Pod と呼ばれる単位でコンテナを構築、管理、および実行します。Red Hat では、OpenShift 外で個々のコンテナおよびコンテナイメージをビルドして管理できます。本書では、RHEL システムで直接実行されるタスクを実行するためのツールについて説明します。

他のコンテナツールの実装とは異なり、ここで説明するツールはモノリシック Docker のコンテナエンジンと、**docker** コマンドを中心としたものではありません。代わりに、Red Hat はコンテナエンジンがなくても動作できる一連のコマンドラインツールを提供します。これには、以下が含まれます。

- **Podman** - Pod およびコンテナイメージの直接管理 (**run**、**stop**、**start**、**ps**、**attach**、**exec** など)
- **buildah** - コンテナイメージの構築、プッシュ、および署名
- **skopeo** - イメージのコピー、検証、削除、および署名
- **runc** - podman および buildah へのコンテナの実行機能と構築機能の提供
- **crun** - ルートレスコンテナの柔軟性、制御、セキュリティーを向上するために設定可能なオプションのランタイム。

これは、このツールが、Docker が生成して管理するのと同じ Linux コンテナや、その他の OCI 互換コンテナエンジンの管理に使用する Open Container Initiative (OCI) と互換性があるためです。ただし、シングルノードのユースケースでは、Red Hat Enterprise Linux で直接実行することが特に適しています。

マルチノードコンテナプラットフォームの場合は、[OpenShift](#) および [CRI-O Container Engine](#) の使用を参照してください。

1.1. PODMAN、BUILDDAH、および SKOPEO の特徴

Docker コマンド機能に代わる、Podman、Skopeo、Buildah ツールが開発されました。このシナリオの各ツールはより軽量になり、機能のサブセットに焦点を当てています。

Podman、Skopeo、Buildah ツールの主な利点は次のとおりです。

- ルートレスモードでの実行 - ルートレスコンテナは、特権を追加しなくても実行されるため、はるかに安全です。
- デーモンの必要なし - コンテナが実行されていない場合、Podman は実行されないため、これらのツールではアイドル時のリソース要件がはるかに少なくなります。Docker は逆に、デーモンが常に実行しています。
- ネイティブ **systemd** 統合 - Podman では **systemd** ユニットファイルを作成し、コンテナをシステムサービスとして実行できます。

Podman、Skopeo、および Buildah の特徴は次のとおりです。

- Podman、Buildah、および CRI-O のコンテナエンジンはすべて、Docker の保存場所 `/var/lib/docker` をデフォルトで使用する代わりに、同じバックエンドストアディレクトリ `/var/lib/containers` を使用します。
- Podman、Buildah、および CRI-O は、同じストレージディレクトリを共有しているため、互いのコンテナと対話することはできません。このツールはイメージを共有できます。
- プログラムで Podman と対話するには、ルートおよびルートレス環境の両方で動作する Podman v2.0 RESTful API を使用してください。詳細は、[コンテナツール API の使用](#) の章を参照してください。

関連情報

- [Say "Hello" to Buildah, Podman, and Skopeo](#)
- [Podman and Buildah for Docker users](#)
- [Buildah:OCI コンテナイメージを構築するツール](#)
- [Podman:OCI コンテナと Pod を管理するためのツール](#)
- [Skopeo:コンテナイメージのコピーと検証を行うツール](#)

1.2. 一般的な PODMAN コマンド

次の基本コマンドを使用して、**podman** ユーティリティーでイメージ、コンテナ、およびコンテナリソースを管理できます。すべての Podman コマンドの完全なリストを表示するには、**podman -h** を使用します。

attach

実行中のコンテナに割り当てます。

commit

変更したコンテナから新しいイメージを作成します。

container checkpoint

1つ以上の実行中のコンテナにチェックポイントを設定します。

container restore

チェックポイントから1つ以上のコンテナを復元します。

build

Containerfile の指示を使用してイメージをビルドします。

create

コンテナを作成しますが、起動はしません。

diff

コンテナのファイルシステムの変更を検証します。

exec

実行中のコンテナでプロセスを実行します。

export

コンテナのファイルシステムの内容を tar アーカイブとしてエクスポートします。

help, h

コマンドリスト、または特定のコマンドのヘルプを表示します。

Healthcheck

コンテナのヘルスチェックを実行します。

history

指定したイメージの履歴を表示します。

images

ローカルストレージ内のイメージをリスト表示します。

import

tarball をインポートして、ファイルシステムイメージを作成します。

info

システム情報を表示します。

inspect

コンテナまたはイメージの設定を表示します。

kill

1つ以上の実行中のコンテナに特定のシグナルを送信します。

kube generate

コンテナ、Pod、またはボリュームに基づいて Kubernetes YAML を生成します。

kube play

Kubernetes YAML に基づいてコンテナ、Pod、ボリュームを作成します。

load

アーカイブからイメージを読み込みます。

login

コンテナレジストリーにログインします。

logout

コンテナレジストリーからログアウトします。

logs

コンテナのログを取得します。

mount

作業中のコンテナの root ファイルシステムをマウントします。

pause

1つ以上のコンテナ内の全プロセスを一時停止します。

ps

コンテナをリスト表示します。

port

コンテナのポートマッピングまたは特定のマッピングをリスト表示します。

pull

レジストリーからイメージをプルします。

push

指定した宛先にイメージをプッシュします。

restart

1つ以上のコンテナを再起動します。

rm

ホストから1つ以上のコンテナを削除する。実行している場合は、**-f**を追加する

rmi

ローカルストレージから1つ以上のイメージを削除します。

run

新しいコンテナでコマンドを実行します。

save

イメージをアーカイブに保存します。

search

レジストリーでイメージを検索します。

start

1つ以上のコンテナを起動します。

stats

1つ以上のコンテナの CPU、メモリー、ネットワーク I/O、ブロック I/O、および PID の割合を表示します。

stop

1つ以上のコンテナを停止します。

tag

ローカルイメージに名前を追加します。

top

コンテナの実行中のプロセスを表示します。

umount, unmount

作業コンテナの root ファイルシステムをアンマウントする。

unpause

1つ以上のコンテナでプロセスの一時停止を解除します。

version

Podman のバージョン情報を表示します。

wait

1つ以上のコンテナをブロックします。

関連情報

- [Podman Basics Cheat Sheet](#)
- [5 Podman features to try now](#)

1.3. DOCKER を使用せずにコンテナを実行

Red Hat では、RHEL 9 から Docker コンテナエンジンと、docker コマンドが削除されました。

RHEL で Docker を使用する場合は、異なるアップストリームプロジェクトから Docker を取得できますが、RHEL 9 では対応していません。

- **podman-docker** パッケージをインストールできます。**docker** コマンドを実行するたびに、実際には **podman** コマンドが実行されます。
- Podman は Docker Socket API にも対応しているため、**podman-docker** パッケージは `/var/run/docker.sock` と `/var/run/podman/podman.sock` の間でリンクを設定します。そのため、Docker デーモンを必要とせずに、**docker-py** と **docker-compose** ツールを使用して Docker API コマンドをそのまま実行できます。Podman はこのような要求を処理します。
- **docker** コマンドなどの **podman** コマンドは、**Containerfile** または **Dockerfile** からコンテナイメージを構築できます。**Containerfile** および **Dockerfile** 内で使用できる利用可能なコマンドは同じです。
- **podman** が対応していない **docker** コマンドのオプションには、network、node、plugin (**podman** はプラグインをサポートしません)、rename (rm および create を使用して **podman** でコンテナの名前を変更します)、secret、service、stack、swarm (**podman** は Docker Swarm をサポートしません) が含まれます。container および image オプションは、**podman** で直接使用されるサブコマンドを実行するのに使用します。

関連情報

- [Podman and Buildah for Docker users](#)

1.4. コンテナの RHEL アーキテクチャーの選択

Red Hat は、以下のコンピューターアーキテクチャーに、コンテナイメージとコンテナ関連のソフトウェアを提供します。

- AMD64 および Intel 64 (ベースイメージおよびレイヤー構造イメージ。32 ビットアーキテクチャーはサポートされません)
- PowerPC 8 および 9 の 64 ビット (ベースイメージおよび大概のレイヤー構造イメージ)
- 64 ビット IBM Z (ベースイメージと、大概の階層構造イメージ)
- ARM 64 ビット (ベースイメージのみ)

初めは、すべてのアーキテクチャーですべての Red Hat イメージがサポートされたわけではありませんが、一覧に挙げられているすべてのアーキテクチャーでほぼすべてが利用可能になりました。

関連情報

- [Universal Base Images \(UBI\): イメージ、リポジトリ、およびパッケージ](#)

1.5. コンテナツールの取得

この手順では、Podman、Buildah、Skopeo、CRIU、Udica といった必要なライブラリーを含む **container-tools** メタパッケージをインストールする方法を説明します。



注記

安定したストリームは RHEL 9 では利用できません。Podman、Buildah、Skopeo などへの安定したアクセスを受けるには、RHEL EUS サブスクリプションを使用します。

手順

1. RHEL をインストールします。
2. RHEL の登録: ユーザー名とパスワードを入力します。ユーザー名とパスワードは、Red Hat カスタマーポータルログイン認証情報と同じです。

```
# subscription-manager register
```

```
Registering to: subscription.rhsm.redhat.com:443/subscription
```

```
Username: <username>
```

```
Password: <password>
```

3. RHEL にサブスクライブします。
 - RHEL に自動的にサブスクライブするには、以下を実行します。

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

- プール ID で RHEL をサブスクライブするには、次のコマンドを実行します。

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

4. **container-tools** メタパッケージをインストールします。

```
# dnf install container-tools
```

5. オプション: **podman-docker** パッケージをインストールします。

```
# dnf install podman-docker
```

podman-docker パッケージは、Docker コマンドラインインターフェイスと **docker-api** を、同等の Podman コマンドに置き換えます。

1.6. ルートレスコンテナの設定

システムで利用可能な機能をコンテナで完全利用できるように、Podman、Skopeo、Buildah などのコンテナツールをスーパーユーザー特権 (root ユーザー) が割り当てられたユーザーで実行するのが最善の方法です。ただし、Red Hat Enterprise Linux 8.1 以降で一般的に利用可能なルートレスコンテナと呼ばれる機能を使用すると、コンテナを一般ユーザーとして使用できます。

Docker などのコンテナエンジンでは、通常の (root 以外の) ユーザーとして Docker コマンドを実行できますが、これらの要求を実行する Docker デーモンは root として実行されます。これにより、一般ユーザーがコンテナ経由で要求を送信でき、システムに影響を与える可能性があります。システム管理者は、ルートレスコンテナユーザーを設定して、一般ユーザーがコンテナアクティビティに損害を与える可能性を回避しつつ、一般ユーザーが自分のアカウントで大半のコンテナ機能を安全に行えるようにします。

この手順では、Podman、Skopeo、および Buildah ツールを使用して、root 以外のユーザー (ルートレス) としてコンテナを操作するようにシステムを設定する方法を説明します。また、一般ユーザーの

アカウントは、コンテナの実行に必要なすべてのオペレーティングシステム機能に完全にアクセスできないために発生する可能性のある制限についても説明します。

前提条件

- root 以外のユーザーでコンテナツールを使用できるように、RHEL システムを設定する必要があります。

手順

1. RHEL をインストールします。
2. **podman** パッケージをインストールします。

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

3. ユーザーアカウントを新規作成します。

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

- ユーザーは、ルートレス Podman を使用できるように自動的に設定されます。
 - **useradd** コマンドは、**/etc/subuid** と **/etc/subgid** ファイルに、アクセス可能なユーザー ID とグループ ID の範囲を自動的に設定します。
 - **etc/subuid** や **/etc/subgid** を手動で変更した場合、新しい変更を適用させるために **podman system migrate** コマンドを実行する必要があります。
4. ユーザーに接続します。

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



注記

これらのコマンドでは正しい環境変数が設定されないため、**su** または **su -** コマンドは使用しないでください。

5. **registry.access.redhat.com/ubi9/ubi** コンテナイメージをプルします。

```
$ podman pull registry.access.redhat.com/ubi9/ubi
```

6. **myubi** という名前のコンテナを実行し、OS バージョンを表示します。

```
$ podman run --rm --name=myubi registry.access.redhat.com/ubi9/ubi \  
cat /etc/os-release  
NAME="Red Hat Enterprise Linux"  
VERSION="9 (Plow)"
```

関連情報

- [Podman を使用したルートレスコンテナ:The basics](#)

- `podman-system-migrate` の man ページ

1.7. ルートレスコンテナへのアップグレード

Red Hat Enterprise Linux 7 からルートレスコンテナにアップグレードするには、ユーザーおよびグループ ID を手動で設定する必要があります。

以下は、Red Hat Enterprise Linux 7 からルートレスコンテナにアップグレードするお t 機に考慮すべき事項です。

- 複数のルートレスコンテナユーザーを設定する場合は、ユーザーごとに一意の範囲を使用します。
- 既存のコンテナイメージとの互換性を最大限にするために、UID および GID に 65536 を使用します。ただし、この数は減らすことができます
- 1000 未満の UID または GID を使用したり、既存のユーザーアカウントから UID または GID を再利用したりしないでください (デフォルトでは 1000 から開始します)。

前提条件

- ユーザーアカウントが作成されている。

手順

- `usermod` コマンドを実行して、UID と GID をユーザーに割り当てます。

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

- `usermod --add-subuid` コマンドは、アクセス可能なユーザー ID の範囲をユーザーのアカウントに追加します。
- `usermod --add-subgids` コマンドは、アクセス可能なユーザー GID およびグループ ID の範囲をユーザーのアカウントに手動で追加します。

検証手順

- UID および GID が正しく設定されていることを確認します。

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

1.8. ルートレスコンテナに関する特別な考慮事項

root 以外のユーザーでコンテナを実行する場合は、考慮事項が複数あります。

- ホストコンテナストレージへのパスは、root ユーザー (`/var/lib/containers/storage`) と root 以外のユーザー (`$HOME/.local/share/containers/storage`) との間では異なります。
- ルートレスコンテナを実行するユーザーには、ホストシステムでユーザー ID およびグループ ID の範囲として実行する特別な権限が付与されます。ただし、ホストのオペレーティングシステムに対する root 権限はありません。

- `etc/subuid` や `/etc/subgid` を手動で変更した場合、新しい変更を適用させるために `podman system migrate` コマンドを実行する必要があります。
- ルートレスコンテナ環境を設定する必要がある場合は、ホームディレクトリーに設定ファイルを作成します (`$HOME/.config/containers`)。設定ファイルには、`storage.conf` (ストレージ設定用) および `containers.conf` (さまざまなコンテナ設定用) が含まれます。また、`registries.conf` ファイルを作成し、Podman を使用してイメージをプル、検索、または実行する時に利用可能なコンテナレジストリーを識別することもできます。
- root 権限なしで変更できないシステム機能もいくつかあります。たとえば、コンテナ内で `SYS_TIME` 機能を設定し、ネットワークタイムサービス (`ntpd`) を実行してシステムクロックを変更できません。root としてコンテナを実行し、ルートレスコンテナ環境を省略して root ユーザーの環境を使用する必要があります。以下に例を示します。

```
# podman run -d --cap-add SYS_TIME ntpd
```

この例では、`ntpd` がコンテナ内だけでなく、システム全体の時間を調整できることに注意してください。

- ルートレスコンテナは、1024 未満のポート番号にアクセスできません。たとえば、ルートレスコンテナの namespace 内では、コンテナの `httpd` サービスからポート 80 を公開するサービスを開始しますが、この namespace の外部からはアクセスできません。

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

ただし、そのポートをホストシステムに公開するには、コンテナには root ユーザーのコンテナ環境を使用するルート権限が必要です。

```
# podman run -d -p 80:80 httpd
```

- ワークステーションの管理者は、ユーザーが 1024 未満のポートでサービスを公開できるようにしますが、セキュリティへの影響を理解する必要があります。たとえば、一般ユーザーは、公式のポート 80 で Web サーバーを実行し、外部ユーザーに対して、管理者が設定したと見せかけることができます。これは、テスト用のワークステーションで問題ありませんが、ネットワークにアクセス可能な開発サーバーでは適切ではなく、実稼働サーバーでは実行しないでください。ユーザーがポート 80 にバインドできるようにするには、次のコマンドを実行します。

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

関連情報

- [ルートレス Podman の欠点](#)

1.9. 高度な PODMAN 設定のためのモジュールの使用

Podman モジュールを使用して、事前に定義された設定セットをロードできます。Podman モジュールは、Tom's Obvious Minimal Language (TOML) 形式の `containers.conf` ファイルです。

このモジュールは、次のディレクトリーまたはそのサブディレクトリーにあります。

- rootless ユーザーの場合: `$HOME/.config/containers/containers.conf.modules`

- root ユーザーの場合: **/etc/containers/containers.conf.modules**、または **/usr/share/containers/containers.conf.modules**

podman --module <your_module_name> コマンドを使用してオンデマンドでモジュールをロードし、システム設定ファイルおよびユーザー設定ファイルをオーバーライドできます。モジュールを操作する際には、次の点に留意してください。

- **--module** オプションを使用して、モジュールを複数回指定できます。
- **<your_module_name>** が絶対パスの場合、設定ファイルは直接ロードされます。
- 相対パスは、前述の3つのモジュールディレクトリーを基準にして解決されます。
- **\$HOME** 内のモジュールは、**/etc/** および **/usr/share/** ディレクトリー内のモジュールをオーバーライドします。

関連情報

- [アップストリームドキュメント](#)

1.10. 関連情報

- [A Practical Introduction to Container Terminology](#)

第2章 コンテナイメージの種類

コンテナイメージは、単一コンテナ実行の全要件、およびそのニーズおよび機能を記述するメタデータを含むバイナリーです。

コンテナイメージには、以下の2つのタイプがあります。

- Red Hat Enterprise Linux Base Images (RHEL ベースイメージ)
- Red Hat Universal Base Images (UBI イメージ)

どちらのタイプのコンテナイメージも Red Hat Enterprise Linux の一部から構築されます。これらのコンテナを使用することで、ユーザーは、信頼性、セキュリティ、パフォーマンス、ライフサイクルを最大限に活用できます。

2種類のコンテナイメージの主な違いは、UBI イメージではコンテナイメージを他のタイプと共有できる点です。UBI を使用してコンテナ化アプリケーションをビルドして、任意のレジストリサーバーにプッシュし、他のサーバーと簡単に共有し、Red Hat 以外のプラットフォームにもデプロイできます。UBI イメージは、コンテナで開発されるクラウドネイティブおよび Web アプリケーションユースケースの基礎として設計されています。

2.1. RHEL コンテナイメージの一般的な特徴

以下の特徴は、RHEL ベースイメージと UBI イメージの両方に当てはまります。

一般的には、RHEL コンテナイメージの特徴は以下のとおりです。

- **サポート対象:**Red Hat では、コンテナ化されたアプリケーションでの使用をサポートしています。Red Hat Enterprise Linux で、安全で、テストされ、認定されたものと同じソフトウェアパッケージが含まれています。
- **カタログ化:**[Red Hat Container Catalog](#) の一覧に追加されます。ここでは、各イメージの説明、技術詳細、および正常指数を確認できます。
- **更新:**明確に定義された更新スケジュールで提供されます。最新のソフトウェアを取得するには、[Red Hat Container Image の更新記事](#) を参照してください。
- **追跡:**Red Hat 製品エラータにより追跡され、各更新に追加された変更を理解するのに役立ちます。
- **再利用可能:**コンテナイメージは、一度実稼働環境にダウンロードしてキャッシュする必要があります。各コンテナイメージは、基盤として含まれるすべてのコンテナで再利用できます。

2.2. UBI イメージの特徴

UBI イメージを使用すると、コンテナイメージを他と共有できます。4つの UBI イメージ (Micro、Minimal、Standard、および init) が提供されます。アプリケーションのビルド用に、事前にビルドされた言語のランタイムイメージと DNF リポジトリが提供されます。

UBI イメージの特徴は以下のとおりです。

- **RHEL コンテンツのサブセットから構築:**Red Hat Universal Base イメージは、通常の Red Hat Enterprise Linux コンテンツのサブセットから構築されます。

- **再配布可能:**UBI イメージは、Red Hat のお客様、パートナー、ISV など向けに標準化できます。UBI イメージを使用すると、自由に共有およびデプロイが可能な公式の Red Hat ソフトウェアにコンテナイメージを構築できます。
- **4つのベースイメージをセットで提供する:** Micro、Minimal、Standard、init
- **ビルド済みの言語ランタイムコンテナイメージのセットを提供します。** [Application Streams](#) をベースとするランタイムイメージは、アプリケーションの基盤を提供し、python、perl、php、dotnet、nodejs、ruby などの標準かつサポート対象のランタイムから利点を受けることができます。
- **関連のある DNF リポジトリを提供する:**DNF リポジトリには、RPM パッケージと更新が含まれており、アプリケーションの依存関係を追加して、UBI コンテナイメージを再ビルドできます。
 - **ubi-9-baseos** リポジトリは、コンテナに追加できる RHEL パッケージの再配布可能なサブセットを保持します。
 - **ubi-9-appstream** リポジトリは、特定のランタイムを必要とするアプリケーションで使用する環境を標準化するために、UBI イメージに追加できるアプリケーションストリームパッケージを保持しています。
 - **UBI RPM の追加:**事前設定された UBI リポジトリから UBI イメージに RPM パッケージを追加できます。切断した環境でこのような機能を使用するには、その機能を使用する UBI コンテンツ配信ネットワーク (<https://cdn-ubi.redhat.com>) を許可リストに追加する必要があります。詳細は [Red Hat Container Images are trying to connect to https://cdn-ubi.redhat.com](#) を参照してください。
- **ライセンス:**[Red Hat Universal Base Image End User License Agreement](#) に従い、UBI イメージを自由に使用および再配布できます。



注記

レイヤー化されたイメージはすべて UBI イメージに基づいています。どの UBI イメージがイメージベースであるかを確認するには、[Red Hat Container Catalog](#) で Containerfile を表示し、UBI イメージに必要なすべてのコンテンツが含まれていることを確認します。

関連情報

- [Red Hat Universal Base Image の概要](#)
- [Universal Base Images \(UBI\):イメージ、リポジトリ、およびパッケージ](#)
- [Red Hat Universal Base Image に必要な情報](#)
- [よくある質問 - Universal Base Image](#)

2.3. UBI 標準イメージの概要

標準イメージ (名称 **ubi**) は、RHEL で実行されるアプリケーション用に設計されています。UBI 標準イメージの主な機能には、以下が含まれます。

- **init システム:****systemd** サービスの管理に必要な **systemd** 初期化システムのすべての機能は、標準のベースイメージで利用できます。この init システムを使用すると、Web サーバー (**httpd**) や FTP サーバー (**vsftpd**) などのサービスを自動的に起動するように事前設定された RPM パッ

ケージをインストールできます。

- **dnf**:ソフトウェアの追加や更新が可能な、無料の dnf リポジトリにアクセスできます。dnf コマンドの標準セット (**dnf**、**dnf-config-manager**、**dnfdownloader** など) を使用できます。
- **ユーティリティー**:ユーティリティーには **tar**、**dmidecode**、**gzip**、**getfacl** や他の ACL コマンド、**dmsetup**、ここに記載されていない他のユーティリティーとのデバイスマッパーコマンドが含まれます。

2.4. UBI INIT イメージの概要

UBI init イメージ (名称 **ubi8-init**) には、**systemd** 初期化システムが含まれているため、Web サーバーやファイルサーバーなどの **systemd** サービスを実行するイメージを構築するのに役立ちます。Init イメージの内容は、標準イメージで得られるものよりも少なくなります。minimal イメージよりも多くなります。

注記

ubi9-init イメージは **ubi9** イメージの上に構築されるため、その内容はほとんど同じです。ただし、重要な相違点がいくつかあります。

- **ubi9-init**:
 - CMD は **/sbin/init** に設定され、デフォルトで **systemd** Init サービスを開始します。
 - **ps** およびプロセス関連のコマンド (**procps-ng** パッケージ) が含まれます。
 - また、**SIGRTMIN+3** を **StopSignal** として設定しています。これは、**ubi9-init** の **systemd** が通常の終了信号 (**SIGTERM** および **SIGKILL**) を無視するためですが、**SIGRTMIN+3** を受け取った場合は終了します。
- **ubi9**:
 - CMD は **/bin/bash** に設定されます。
 - **ps** およびプロセス関連のコマンド (**procps-ng** パッケージ) は含まれません。
 - 通常の終了シグナルを無視しません (**SIGTERM** および **SIGKILL**)。

2.5. UBI MINIMAL イメージの概要

ubi-minimal という名前の UBI minimal イメージは、縮小された事前にインストールされたコンテンツセットおよびパッケージマネージャー (**microdnf**) を提供します。これにより、イメージに含まれる依存関係を縮小しながら、**Containerfile** を使用できます。

UBI minimal イメージの主な機能には、以下が含まれます。

- **サイズが小さい**:minimal イメージは、ディスク上では約 92M、圧縮時は 32M です。これにより、サイズが、標準イメージの半分に満たなくなります。
- **ソフトウェアインストール (microdnf)**:ソフトウェアリポジトリおよび RPM ソフトウェアパッケージを使用する完全に開発された **dnf** 機能を追加する代わりに、minimal イメージには **microdnf** ユーティリティーが含まれます。**microdnf** は **dnf** の縮小バージョンであるため、リ

ポジトリの有効化/無効化、パッケージの削除と更新、パッケージのインストール後にキャッシュを削除できます。

- **RHEL パッケージをベースとする**:minimal イメージには、通常の RHEL ソフトウェアの RPM パッケージから機能がいくつか削除されたものです。minimal イメージには、**systemd** や System V init、Python ランタイム環境、および一部のシェルユーティリティーなど、初期化およびサービス管理システムが含まれません。オーバーヘッドの量を可能な限り最小限に抑えながら、RHEL リポジトリを使用してイメージを構築できます。
- **microdnfのモジュールがサポートされている**:**microdnf** コマンドで使用されるモジュールにより、利用可能な場合は、同じソフトウェアの複数のバージョンをインストールできます。**microdnf module enable**、**microdnf module disable**、および **microdnf module reset** を使用して、モジュールストリームをそれぞれ有効化、無効化、およびリセットできます。
 - たとえば、UBI minimal コンテナ内で **nodejs:14** モジュールストリームを有効にするには、次のコマンドを実行します。

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

Running transaction test...
```

Red Hat は UBI の最新バージョンのみをサポートし、ドットリリースの過去バージョンはサポートしません。特定のドットリリースを使用し続ける必要がある場合は、[延長更新サポート](#) を参照してください。

2.6. UBI マイクロイメージの概要

ubi-micro は、取得可能な最小の UBI イメージで、パッケージマネージャーと、通常はコンテナイメージに含まれるすべての依存関係を除外することで得られます。これにより、他のアプリケーションに UBI Standard、Minimal、または Init を使用する場合でも、**ubi-micro** イメージをベースとするコンテナイメージに対する攻撃対象領域を最小限に抑えられるので、minimal アプリケーションに適しています。Linux ディストリビューションパッケージのないコンテナイメージは Distroless コンテナイメージと呼ばれます。

第3章 コンテナレジストリーの使用

コンテナイメージレジストリーは、コンテナイメージとコンテナベースのアプリケーションアーティファクトを保存するためのリポジトリまたはリポジトリのコレクションです。`/etc/containers/registries.conf` ファイルは、Podman、Buildah、Skopeo などのさまざまなコンテナツールで使用できるコンテナイメージレジストリーを含むシステム全体の設定ファイルです。

コンテナツールに指定されたコンテナイメージが完全修飾されていない場合に、コンテナツールは `registries.conf` ファイルを参照します。`registries.conf` ファイル内で短縮名のエイリアスを指定すると、完全修飾されていないイメージの取得元を管理者が完全に制御できるようになります。たとえば、`podman pull example.com/example_image` コマンドは、`registries.conf` ファイルで指定されているように、`example.com` レジストリーからローカルシステムにコンテナイメージをプルします。

3.1. コンテナレジストリー

コンテナレジストリーは、コンテナイメージとコンテナベースのアプリケーションアーティファクトを保存するためのリポジトリまたはリポジトリのコレクションです。Red Hat が提供するレジストリーは以下のとおりです。

- registry.redhat.io (認証が必要)
- registry.access.redhat.com (認証なし)
- registry.connect.redhat.com (Red Hat Partner Connect プログラムイメージ)

リモートレジストリー (Red Hat の独自のコンテナレジストリーなど) からコンテナイメージを取得して、ローカルシステムに追加するには、`podman pull` コマンドを使用します。

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

ここで、`<registry>[:<port>]/[<namespace>]/<name>:<tag>` はコンテナイメージの名前です。

たとえば、`registry.redhat.io/ubi9/ubi` コンテナイメージは以下によって識別されます。

- レジストリーサーバー (`registry.redhat.io`)
- namespace (`ubi9`)
- イメージ名 (`ubi`)

同じイメージにバージョンが複数ある場合は、イメージ名を明示的に指定するタグを追加します。デフォルトでは、Podman は `:latest` タグを使用します (例: `ubi9/ubi:latest`)。

一部のレジストリーでは、`<namespace>` も使用して、異なるユーザーまたは組織によって所有される同じ `<name>` のイメージを区別します。以下に例を示します。

名前空間	<code><namespace>/<name></code> の例
organization	<code>redhat/kubernetes</code> 、 <code>google/kubernetes</code>
login (ユーザー名)	<code>alice/application</code> 、 <code>bob/application</code>

名前空間	(<namespace>/<name>) の例
role	devel/database、test/database、prod/databa se

registry.redhat.io への移行の詳細は、[Red Hat Container Registry Authentication](#) を参照してください。registry.redhat.io からコンテナを取得する前に、RHEL サブスクリプション認証情報を使用して認証する必要があります。

3.2. コンテナレジストリーの設定

podman info --format コマンドを使用して、コンテナレジストリーを表示できます。

```
$ podman info -f json | jq '.registries["search"]'
[
  "registry.access.redhat.com",
  "registry.redhat.io",
  "docker.io"
]
```



注記

podman info コマンドは、Podman 4.0.0 以降で使用できます。

registries.conf 設定ファイルでコンテナレジストリーのリストを編集できます。root ユーザーとして、**/etc/containers/registries.conf** ファイルを編集し、デフォルトのシステム全体の検索設定を変更します。

ユーザーとして、**\$HOME/.config/containers/registries.conf** ファイルを作成し、システム全体の設定を上書きします。

```
unqualified-search-registries = ["registry.access.redhat.com", "registry.redhat.io", "docker.io"]
short-name-mode = "enforcing"
```

デフォルトでは、**podman pull** および **podman search** コマンドは、**unqualified-search-registries** のリストに記載のレジストリーから、その順序でコンテナイメージを検索します。

ローカルコンテナレジストリーの設定

TLS 検証なしでローカルコンテナレジストリーを設定できます。TLS 検証を無効にする方法は 2 つあります。まず、Podman で **--tls-verify=false** オプションを使用できます。次に、**registries.conf** ファイルに **insecure=true** を設定できます。

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

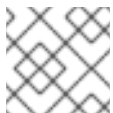
レジストリー、名前空間、またはイメージのブロック

ローカルシステムがアクセスできないレジストリーを定義できます。**blocked=true** を設定して、特定のレジストリーをブロックできます。

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

接頭辞を **prefix="registry.example.org/namespace"** に設定して、名前空間をブロックすることもできます。たとえば、**podman pull registry.example.org/example/image:latest** コマンドを使用するイメージのプルは、指定した接頭辞が一致するためブロックされます。

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



注記

prefix はオプションで、デフォルト値は **location** の値と同じです。

prefix="registry.example.org/namespace/image" を設定して、特定のイメージをブロックできません。

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

レジストリーのミラーリング

元のレジストリーにアクセスできない場合は、レジストリーミラーを設定できます。たとえば、機密レベルの高い環境で作業するため、インターネットに接続することはできません。指定された順序でアクセスされる複数のミラーを指定できます。たとえば、**podman pull registry.example.com/myimage:latest** コマンドを実行すると、まず **mirror-1.com** が試行され、次に **mirror-2.com** が試行されます。

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

関連情報

- [How to manage Linux container registries](#)
- **podman-pull** の man ページ
- **podman-info** の man ページ

3.3. コンテナイメージの検索

podman search コマンドを使用すると、イメージ用に選択したコンテナレジストリーを検索できます。また、[Red Hat Container Catalog](#) でイメージを検索することもできます。Red Hat コンテナレジストリーには、イメージの説明、コンテンツ、ヘルスインドエックスなど情報が含まれます。



注記

podman search コマンドは、イメージの存在を判断する信頼できる方法ではありません。v1 および v2 Docker ディストリビューション API の **podman search** 動作は、各レジストリーの実装に固有のもので、一部のレジストリーは、検索をまったくサポートしない場合があります。検索用語を使用しない検索は、v2 API を実装するレジストリーでのみ機能します。**docker search** コマンドにも、同じことが言えます。

quay.io レジストリーで **postgresql-10** イメージを検索するには、手順に従います。

前提条件

- **container-tools** メタパッケージがインストールされている。
- レジストリーが設定されている。

手順

1. レジストリーに対して認証します。

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

2. イメージを検索します。

- 特定のレジストリーで特定のイメージを検索するには、以下を入力します。

```
# podman search quay.io/postgresql-10
INDEX      NAME                                DESCRIPTION          STARS  OFFICIAL
AUTOMATED
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
```

- または、特定のレジストリーで提供されるすべてのイメージを表示するには、以下を入力します。

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

- すべてのレジストリー全体でイメージ名を検索するには、以下を入力します。

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

完全な説明を表示するには、**--no-trunc** オプションをコマンドに渡します。

関連情報

- **podman-search** の man ページ

3.4. レジストリーからのイメージの取得 (プル)

podman pull コマンドを使用して、ローカルシステムにイメージを取得します。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. registry.redhat.io レジストリーにログインします。

```
$ podman login registry.redhat.io
Username: <username>
Password: <password>
Login Succeeded!
```

2. registry.redhat.io/ubi9/ubi コンテナイメージをプルします。

```
$ podman pull registry.redhat.io/ubi9/ubi
```

検証手順

- ローカルシステムにプルしたすべてのイメージをリスト表示します。

```
$ podman images
REPOSITORY          TAG      IMAGE ID      CREATED      SIZE
registry.redhat.io/ubi9/ubi  latest  3269c37eae33  7 weeks ago  208 MB
```

関連情報

- **podman-pull** の man ページ

3.5. 短縮名のエイリアスの設定

Red Hat は、常に完全修飾名でイメージをプルすることを推奨します。ただし、短縮名でイメージをプルすることが一般的です。たとえば、**registry.access.redhat.com/ubi9:latest** の代わりに **ubi9** を使用できます。

registries.conf ファイルにより、短縮名のエイリアスを指定でき、管理者はイメージをプルする場所を完全に制御できます。エイリアスは、**"name" = "value"** の形式で **[aliases]** テーブルで指定されます。エイリアスのリストは、**/etc/containers/registries.conf.d** ディレクトリーで確認できます。Red Hat は、このディレクトリーでエイリアスのセットを提供します。たとえば、**podman pull ubi9** は、**registry.access.redhat.com/ubi9:latest** である適切なイメージに直接解決します。

以下に例を示します。

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

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

short-names モードは以下のようになります。

- **enforcing**: イメージのプル中に一致するエイリアスが見つからない場合、Podman はユーザー

が非修飾レジストリーのいずれかを選択するよう求めます。選択したイメージを正常に取得すると、Podman は、**\$HOME/.cache/containers/short-name-aliases.conf** ファイル (ルートレスユーザー) または **/var/cache/containers/short-name-aliases.conf** (root ユーザー) に新しい短縮名のエイリアスを自動的に記録します。ユーザーを要求できない場合 (stdin や stdout など) が TTY ではない場合は、Podman は失敗します。**short-name-aliases.conf** ファイルは、両方が同じエイリアスを指定する場合、**registries.conf** ファイルよりも優先されることに注意してください。

- **permissive:enforcing** モードと似ていますが、ユーザーにプロンプトが表示されないと Podman は失敗しません。代わりに、Podman は指定された順序で修飾されていないすべてのレジストリーを検索します。エイリアスは記録されないことに注意してください。
- **disabled**: すべての非修飾検索レジストリーが指定の順序で試行され、エイリアスは記録されません。



注記

Red Hat では、レジストリー、名前空間、イメージ名、およびタグが含まれる完全修飾イメージ名を使用することを推奨します。短縮名を使用する場合は、なりすましの固有リスクが常にあります。不明なユーザーや匿名ユーザーが任意の名前でアカウントを作成できないように、信頼できるレジストリー (つまりレジストリー) を追加します。たとえば、ユーザーは **example.registry.com registry** レジストリーから **example** コンテナイメージをプルします。**example.registry.com** が検索リストの最初でない場合には、攻撃者は検索リストの前のほうに別の **example** イメージを配置することができてしまいます。目的のコンテンツではなく、攻撃者が配置したイメージを間違えてプルして実行する可能性があります。

関連情報

- [Container image short names in Podman](#)

第4章 コンテナイメージの使用

Podman ツールは、コンテナイメージと連携するように設計されています。このツールを使用して、イメージのプル、イメージ署名の検査、タグ付け、保存、読み込み、再配布、および定義を行うことができます。

4.1. 短縮名のエイリアスを使用したコンテナイメージのプル

セキュアな短縮名を使用して、ローカルシステムにイメージを取得できます。以下の手順では、**fedora** または **nginx** のコンテナイメージをプルする方法を説明します。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

- コンテナイメージをプルします。
 - **fedora** イメージをプルします。

```
$ 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
...
```

エイリアスが検出され、**registry.fedoraproject.org/fedora** イメージが安全にプルされます。**unqualified-search-registries** のリストは、**fedora** イメージ名を解決するためには使用されません。

- **nginx** イメージをプルします。

```
$ 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
...
```

一致するエイリアスが見つからない場合は、**unqualified-search-registries** リストのいずれかを選択するように求められます。選択したイメージが正常にプルされると、新しい短縮名のエイリアスがローカルに記録されます。そうでない場合はエラーが生じます。

検証

- ローカルシステムにプルしたすべてのイメージをリスト表示します。

```
$ 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

関連情報

- [Container image short names in Podman](#)

4.2. イメージのリスト表示

podman images コマンドを使用して、ローカルストレージのイメージをリスト表示します。

前提条件

- **container-tools** メタパッケージがインストールされている。
- プルしたイメージがローカルシステムで利用できる。

手順

- ローカルストレージ内のイメージをすべてリスト表示します。

```
$ podman images
REPOSITORY          TAG      IMAGE ID      CREATED      SIZE
registry.access.redhat.com/ubi9/ubi latest 3269c37eae33 6 weeks ago 208 MB
```

関連情報

- **podman-images** の man ページ

4.3. ローカルイメージの検証

ローカルシステムにイメージをプルして実行したら、**podman inspect** コマンドを使用してイメージを調査できます。たとえば、イメージの内容を理解して、イメージ内のソフトウェアを確認します。**podman inspect** コマンドは、名前または ID で識別されるコンテナおよびイメージに関する情報を表示します。

前提条件

- **container-tools** メタパッケージがインストールされている。
- プルしたイメージがローカルシステムで利用できる。

手順

- **registry.redhat.io/ubi9/ubi** イメージを確認します。

```
$ podman inspect registry.redhat.io/ubi9/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": "ubi9-container",
"com.redhat.license_terms": "https://www.redhat.com/...",
"description": "The Universal Base Image is ..."
}
...

```

"**Cmd**" キーは、コンテナ内で実行するデフォルトのコマンドを指定します。このコマンドは、**podman run** コマンドに引数として指定すると、上書きできます。この ubi9/ubi コンテナは、**podman run** で起動時に他の引数を指定していない場合には、bash シェルを実行します。"**Entrypoint**" キーが設定された場合は、"**Cmd**" 値の代わりにその値が使用されます (また、Entrypoint コマンドの引数として "**Cmd**" の値が使用されます)。

関連情報

- **podman-inspect** の man ページ

4.4. リモートイメージの検証

skopeo inspect コマンドを使用して、システムにイメージをプルする前に、リモートコンテナレジストリーからイメージに関する情報を表示します。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

- **container-tools** メタパッケージがインストールされている。
- **registry.redhat.io/ubi9/ubi-init** イメージを確認します。

```

# skopeo inspect docker://registry.redhat.io/ubi9/ubi-init
{
  "Name": "registry.redhat.io/ubi9/ubi-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": "ubi9-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"
  }
}

```

```

| }
| }
| ...

```

関連情報

- **skopeo-inspect** の man ページ

4.5. コンテナイメージのコピー

skopeo copy コマンドを使用して、コンテナイメージをあるレジストリーから別のレジストリーにコピーできます。たとえば、外部レジストリーのイメージを使用して内部リポジトリーに反映させたり、2つの異なる場所のイメージレジストリーを同期したりできます。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

- **skopeo** コンテナイメージを **docker://quay.io** から **docker://registry.example.com** にコピーします。

```

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

```

関連情報

- **skopeo-copy** の man ページ

4.6. ローカルディレクトリーへのイメージレイヤーのコピー

skopeo copy コマンドを使用して、コンテナイメージのレイヤーをローカルディレクトリーにコピーできます。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. **/var/lib/images/nginx** ディレクトリーを作成します。

```

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

```

2. **docker://docker.io/nginx:latest** イメージのレイヤーを、新たに作成したディレクトリーにコピーします。

```

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

```

検証

- `/var/lib/images/nginx` ディレクトリーの内容を表示します。

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

関連情報

- `skopeo-copy` の man ページ

4.7. イメージのタグ付け

`podman tag` コマンドを使用して、ローカルイメージに別の名前を追加します。この名前は、`<registryhost>/<username>/<name>:<tag>` のように複数の部分で構成されます。

前提条件

- `container-tools` メタパッケージがインストールされている。
- プルしたイメージがローカルシステムで利用できる。

手順

1. すべてのイメージをリスト表示します。

```
$ podman images
REPOSITORY          TAG      IMAGE ID      CREATED      SIZE
registry.redhat.io/ubi9/ubi  latest  3269c37eae33  7 weeks ago  208 MB
```

2. 以下のいずれかを使用して、`myubi` 名を `registry.redhat.io/ubi9/ubi` イメージに割り当てます。

- イメージ名:

```
$ podman tag registry.redhat.io/ubi9/ubi myubi
```

- イメージ ID:

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

どちらのコマンドも、同じ結果になります。

3. すべてのイメージをリスト表示します。

```
$ podman images
REPOSITORY          TAG      IMAGE ID      CREATED      SIZE
registry.redhat.io/ubi9/ubi  latest  3269c37eae33  2 months ago  208 MB
localhost/myubi        latest  3269c37eae33  2 months ago  208 MB
```

デフォルトのタグがいずれのイメージでも `latest` であることに注意してください。すべてのイメージ名が単一のイメージ ID `3269c37eae33` に割り当てられていることを確認できます。

4. 以下のいずれかを使用して **registry.redhat.io/ubi9/ubi** イメージに **9** タグを追加します。

- イメージ名:

```
$ podman tag registry.redhat.io/ubi9/ubi myubi:9
```

- イメージ ID:

```
$ podman tag 3269c37eae33 myubi:9
```

どちらのコマンドも、同じ結果になります。

5. すべてのイメージをリスト表示します。

```
$ podman images
REPOSITORY          TAG      IMAGE ID      CREATED      SIZE
registry.redhat.io/ubi9/ubi  latest  3269c37eae33  2 months ago  208 MB
localhost/myubi     latest  3269c37eae33  2 months ago  208 MB
localhost/myubi     9       3269c37eae33  2 months ago  208 MB
```

デフォルトのタグがいずれのイメージでも **latest** であることに注意してください。すべてのイメージ名が単一のイメージ ID 3269c37eae33 に割り当てられていることを確認できます。

registry.redhat.io/ubi9/ubi イメージにタグ付けした後に、コンテナを実行するオプションが3つあります。

- ID (**3269c37eae33**)
- 名前 (**localhost/myubi:latest**)
- 名前 (**localhost/myubi:9**)

関連情報

- **podman-tag** の man ページ

4.8. イメージの保存および読み込み

podman save コマンドを使用して、イメージをコンテナアーカイブに保存します。別のコンテナ環境に後で復元したり、別のユーザーに送信することもできます。**--format** オプションを使用して、アーカイブ形式を指定できます。サポート対象の形式は以下のとおりです。

- **docker-archive**
- **oci-archive**
- **oci-dir** (oci マニフェストタイプのあるディレクトリー)
- **docker-dir** (v2s2 マニフェストタイプを含むディレクトリー)

デフォルトの形式は **docker-dir** です。

podman load コマンドを使用して、コンテナイメージアーカイブからコンテナストレージにイメージを読み込みます。

前提条件

- **container-tools** メタパッケージがインストールされている。
- プルしたイメージがローカルシステムで利用できる。

手順

1. **registry.redhat.io/rhel9/rsyslog** イメージを tarball として保存します。

- デフォルトの **docker-dir** 形式で以下を行います。

```
$ podman save -o myrsyslog.tar registry.redhat.io/rhel9/rsyslog:latest
```

- **oci-archive** 形式で、**--format** オプションを使用します。

```
$ podman save -o myrsyslog-oci.tar --format=oci-archive registry.redhat.io/rhel9/rsyslog
```

myrsyslog.tar および **myrsyslog-oci.tar** アーカイブは現在のディレクトリーに保存されます。次の手順では、**myrsyslog.tar** tarball で実行されます。

2. **myrsyslog.tar** のファイルタイプを確認します。

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

3. **myrsyslog.tar** から **registry.redhat.io/rhel9/rsyslog:latest** イメージを読み込むには、以下を実行します。

```
$ podman load -i myrsyslog.tar
...
Loaded image(s): registry.redhat.io/rhel9/rsyslog:latest
```

関連情報

- **podman-save** の man ページ

4.9. UBI イメージの再配布

podman push コマンドを使用して、UBI イメージを独自のレジストリーやサードパーティーのレジストリーにプッシュするか、他のと共有します。UBI dnf リポジトリーから、必要に応じてイメージをアップグレードまたは追加できます。

前提条件

- **container-tools** メタパッケージがインストールされている。
- プルしたイメージがローカルシステムで利用できる。

手順

1. オプション: **ubi** イメージに名前を追加します。

```
# podman tag registry.redhat.io/ubi9/ubi registry.example.com:5000/ubi9/ubi
```

2. **registry.example.com:5000/ubi9/ubi** イメージをローカルストレージからレジストリーにプッシュします。

```
# podman push registry.example.com:5000/ubi9/ubi
```

重要

このイメージを使用する方法には制限がいくつかありますが、その方法を参照する方法にもいくつかの制限があります。たとえば、Red Hat Container Certification または Red Hat OpenShift Operator Certification の [Red Hat Partner Connect Program](#) で認定されていなければ、Red Hat の認定イメージまたは Red Hat のサポートイメージとはみなされません。

4.10. イメージの削除

podman rmi コマンドを使用して、ローカルに保存されたコンテナイメージを削除します。ID または名前を使用して、イメージを削除できます。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. ローカルシステムにある全イメージのリストを表示します。

```
$ 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. すべてのコンテナをリスト表示します。

```
$ 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 イメージを削除するには、**podman stop** コマンドを使用して、このイメージから実行中のコンテナをすべて停止する必要があります。コンテナを ID または名前を使用して停止できます。

3. **mysyslog** コンテナを停止します。

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

4. **registry.redhat.io/rhel8/rsyslog** イメージを削除します。

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

- 複数のイメージを削除するには、以下のコマンドを実行します。

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

- システムからすべてのイメージを削除するには、以下のコマンドを実行します。

```
$ podman rmi -a
```

- 複数の名前 (タグ) が関連付けられているイメージを削除するには、**-f** オプションを追加して削除します。

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

関連情報

- `podman-rmi` の man ページ

第5章 コンテナの使用

コンテナは、デプロイメントされたコンテナイメージにあるファイルから作成された、実行中プロセスまたは停止プロセスを表します。Podman ツールを使用してコンテナと連携できます。

5.1. PODMAN RUN コマンド

podman run コマンドは、コンテナイメージをもとに新しいコンテナでプロセスを実行します。コンテナイメージがすでにロードされていない場合は、**podman run** は、そのイメージからコンテナを起動する前に、**podman pull image** を実行するのと同じ方法で、リポジトリからイメージおよびイメージの全依存関係を取得します。コンテナプロセスには、独自のファイルシステム、独自のネットワーク、独自のプロセスツリーがあります。

podman run コマンドの形式は次のとおりです。

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

基本オプションは次のとおりです。

- **--detach (-d)**:コンテナをバックグラウンドで実行し、新しいコンテナ ID を出力します。
- **--attach (-a)**:フォアグラウンドモードでコンテナを実行します。
- **--name (-n)**:コンテナに名前を割り当てます。**--name** で名前がコンテナに割り当てられていない場合には、ランダムな文字列で名前が生成されます。これはバックグラウンドコンテナとフォアグラウンドコンテナの両方で機能します。
- **--rm**:終了時にコンテナを自動的に削除します。コンテナが正常に作成または起動できない場合には、削除されない点に注意してください。
- **--tty (-t)**:コンテナの標準入力に、疑似端末を割り当て、接続します。
- **--interactive (-i)**:対話式プロセスの場合には、**-i** と **-t** を併用して、コンテナプロセスに端末を割り当てます。**-i -t** は頻繁に **-it** と記述されます。

5.2. ホストからのコンテナでのコマンド実行

podman run コマンドを使用して、コンテナのオペレーティングシステムの種類を表示します。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. **cat /etc/os-release** コマンドを使用して、**registry.access.redhat.com/ubi9/ubi** コンテナイメージをベースとするコンテナのオペレーティングシステムの種類を表示します。

```
$ podman run --rm registry.access.redhat.com/ubi9/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 9"
...
```

- オプション: すべてのコンテナをリスト表示します。

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

--rm オプションがあるのでコンテナは表示されません。コンテナが削除されました。

関連情報

- `podman-run` の man ページ

5.3. コンテナ内でのコマンドの実行

`podman run` コマンドを使用して、コンテナを対話的に実行します。

前提条件

- `container-tools` メタパッケージがインストールされている。

手順

1. `registry.redhat.io/ubi9/ubi` イメージに基づいて、`myubi` という名前のコンテナを実行します。

```
$ podman run --name=myubi -it registry.access.redhat.com/ubi9/ubi /bin/bash
[root@6ccffd0f6421 /]#
```

- `-i` オプションは対話式のセッションを作成します。`-t` オプションを指定しないと、シェルは開いたままにも拘らず、シェルには何も入力できません。
 - `-t` オプションは、端末セッションを開きます。`-i` オプションを指定しないと、シェルが開き、終了します。
2. システムユーティリティのセットが含まれる `procps-ng` パッケージをインストールします (例: `ps`、`top`、`uptime` など)。

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

3. `ps -ef` コマンドを使用して、現在のプロセスをリスト表示します。

```
# 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` を実行してコンテナを終了し、ホストに戻ります。

```
# exit
```

- オプション: すべてのコンテナをリスト表示します。

```
$ podman ps
CONTAINER ID IMAGE COMMAND CREATED STATUS
PORTS NAMES
1984555a2c27 registry.redhat.io/ubi9/ubi:latest /bin/bash 21 minutes ago Exited (0) 21
minutes ago myubi
```

コンテナが終了ステータスであることを確認できます。

関連情報

- `podman-run` の man ページ

5.4. コンテナのリスト表示

`podman ps` コマンドを使用して、システムで実行中のコンテナのリストを表示します。

前提条件

- `container-tools` メタパッケージがインストールされている。

手順

1. `registry.redhat.io/rhel9/rsyslog` イメージをベースとするコンテナを実行します。

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

2. すべてのコンテナをリスト表示します。

- 実行中のコンテナのリストを表示するには、以下のコマンドを実行します。

```
$ podman ps
CONTAINER ID IMAGE COMMAND CREATED STATUS
PORTS NAMES
74b1da000a11 rhel9/rsyslog /bin/rsyslog.sh 2 minutes ago Up About a minute
musing_brown
```

- (実行中または停止中の) すべてのコンテナを一覧表示するには、以下を実行します。

```
$ podman ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS
NAMES IS INFRA
d65aecc325a4 ubi9/ubi /bin/bash 3 secs ago Exited (0) 5 secs ago peaceful_hopper
false
74b1da000a11 rhel9/rsyslog rsyslog.sh 2 mins ago Up About a minute musing_brown
false
```

実行されていないものの削除されていない (`--rm` オプション) コンテナが存在する場合には、コンテナがあるので、再起動できます。

関連情報

- **podman-ps** の man ページ

5.5. コンテナの起動

コンテナを実行してから停止し、削除しない場合には、ローカルシステムに保存されて再び実行する準備ができます。**podman start** コマンドを使用して、コンテナを再実行できます。コンテナ ID または名前を使用して、コンテナを指定できます。

前提条件

- **container-tools** メタパッケージがインストールされている。
- 1つ以上のコンテナが停止されている。

手順

1. **myubi** コンテナを起動します。

- 非対話モードで以下を行います。

```
$ podman start myubi
```

または、**podman start 1984555a2c27** を使用することもできます。

- インタラクティブモードで **-a (--attach)** および **-i (--interactive)** オプションを使用して、コンテナの bash シェルと連携します。

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

または、**podman start -a -i 1984555a2c27** を使用することができます。

2. **exit** を実行してコンテナを終了し、ホストに戻ります。

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

関連情報

- **podman-start** の man ページ

5.6. ホストからのコンテナの検証

podman inspect コマンドを使用して、既存のコンテナのメタデータを JSON 形式で検証します。コンテナ ID または名前を使用して、コンテナを指定できます。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

- ID 64ad95327c74 で定義されるコンテナを検査します。
 - すべてのメタデータを取得するには、以下のコマンドを実行します。

■

```
$ 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 ファイルから特定の項目 (例: **StartedAt** タイムスタンプ) を取得するには、以下を実行します。

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

その情報は階層構造で保存されます。コンテナの **StartedAt** タイムスタンプ (**StartedAt** は **State** の配下にある) を確認するには、**--format** オプションとコンテナ ID または名前を使用します。

検証する他の項目の例には、以下が含まれます。

- **.path**: コンテナとともに実行するコマンドを表示します。
- **.Args**: コマンドに指定する引数
- **.Config.ExposedPorts**: コンテナから公開する TCP または UDP ポート
- **.state.Pid**: コンテナのプロセス ID を表示します。
- **.HostConfig.PortBindings**: コンテナからホストへのポートマッピング

関連情報

- **podman-inspect** の man ページ

5.7. LOCALHOST のディレクトリーのコンテナへのマウント

コンテナでホストの **/dev/log** デバイスをマウントして、コンテナ内のログメッセージをホストシステムに公開できます。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. **log_test** という名前のコンテナを実行し、コンテナにホストの **/dev/log** デバイスをマウントします。

```
# podman run --name="log_test" -v /dev/log:/dev/log --rm \
registry.redhat.io/ubi9/ubi logger "Testing logging to the host"
```

2. **journalctl** ユーティリティーを使用してログを表示します。

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

--rm オプションは、終了時にコンテナを削除します。

関連情報

- **podman-run** の man ページ

5.8. コンテナのファイルシステムのマウント

podman mount コマンドを使用して、ホストからアクセス可能な場所に、作業コンテナの root ファイルシステムをマウントします。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. **mysyslog** という名前のコンテナを実行します。

```
# podman run -d --name=mysyslog registry.redhat.io/rhel9/rsyslog
```

2. オプション: すべてのコンテナをリスト表示します。

```
# podman ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS
PORTS NAMES
c56ef6a256f8 registry.redhat.io/rhel9/rsyslog:latest /bin/rsyslog.sh 20 minutes ago Up 20
minutes ago mysyslog
```

3. **mysyslog** コンテナをマウントします。

```
# podman mount mysyslog
/var/lib/containers/storage/overlay/990b5c6ddcdeed4bde7b245885ce4544c553d108310e2b797
d7be46750894719/merged
```

4. **ls** コマンドを使用して、マウントポイントのコンテンツを表示します。

```
# 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 バージョンを表示します。

-

```
# cat
/var/lib/containers/storage/overlay/990b5c6ddcdeed4bde7b245885ce4544c553d108310
e2b797d7be46750894719/merged/etc/os-release
NAME="Red Hat Enterprise Linux"
VERSION="9 (Ootpa)"
ID="rhel"
ID_LIKE="fedora"
...
```

関連情報

- `podman-mount` の man ページ

5.9. 静的 IP を使用したデーモンとしてのサービスの実行

以下の例は、`rsyslog` サービスをバックグラウンドでデーモンプロセスとして実行します。`--ip` オプションは、コンテナのネットワークインターフェイスを特定の IP アドレスに設定します (例: 10.88.0.44)。その後、`podman inspect` コマンドを実行して、IP アドレスを適切に設定できます。

前提条件

- `container-tools` メタパッケージがインストールされている。

手順

1. コンテナのネットワークインターフェイスを IP アドレス 10.88.0.44 に設定します。

```
# podman run -d --ip=10.88.0.44 registry.access.redhat.com/rhel9/rsyslog
efde5f0a8c723f70dd5cb5dc3d5039df3b962fae65575b08662e0d5b5f9fbe85
```

2. IP アドレスが正しく設定されていることを確認します。

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

関連情報

- `podman-inspect` の man ページ
- `podman-run` の man ページ

5.10. 実行中のコンテナ内でのコマンドの実行

`podman exec` コマンドを使用して、実行中のコンテナでコマンドを実行し、そのコンテナを調べます。コンテナアクティビティを中断せずに実行中のコンテナを調査できるので、`podman run` コマンドの代わりに `podman exec` コマンドを使用します。

前提条件

- `container-tools` メタパッケージがインストールされている。
- コンテナが実行されている。

手順

1. **myrsyslog** コンテナ内で **rpm -qa** コマンドを実行し、インストールされているパッケージのリストを表示します。

```
$ 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** コンテナで **/bin/bash** コマンドを実行します。

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

3. システムユーティリティーのセットが含まれる **procps-ng** パッケージをインストールします (例: **ps**、**top**、**uptime** など)。

```
# dnf install procps-ng
```

4. コンテナを検査します。

- システムにある全プロセスをリスト表示するには、以下のコマンドを実行します。

```
# 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
```

- ファイルシステムのディスク領域の使用量を表示するには、次のコマンドを実行します。

```
# 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
...
```

- システム情報を表示するには、以下のコマンドを実行します。

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

- MB 単位でメモリーの空き容量を表示するには、次のコマンドを実行します。

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


関連情報

- `podman-exec` の man ページ

5.11. 2つのコンテナ間でのファイルの共有

コンテナが削除されても、ボリュームを使用してコンテナ内のデータを永続化できます。ボリュームは、複数のコンテナ間でのデータ共有に使用できます。ボリュームとは、ホストマシンに保存されているフォルダーです。ボリュームはコンテナとホスト間で共有できます。

主な利点は以下のとおりです。

- ボリュームはコンテナ間で共有できます。
- ボリュームは、他と比べるとバックアップまたは移行が簡単です。
- ボリュームを使用するとコンテナのサイズが増えません。

前提条件

- `container-tools` メタパッケージがインストールされている。

手順

1. ボリュームを作成します。

```
$ podman volume create hostvolume
```

2. ボリュームに関する情報を表示します。

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

volumes ディレクトリーにボリュームが作成されることに注意してください。\$ `mntPoint=$(podman volume inspect hostvolume --format {{.Mountpoint}})` で、変数へのマウントポイントパスを保存して操作を簡素化できます。

`sudo podman volume create hostvolume` を実行すると、マウントポイントが `/var/lib/containers/storage/volumes/hostvolume/_data` に変わります。

3. `mntPoint` 変数に保管されたパスを使用して、ディレクトリー内にテキストファイルを作成します。

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

4. **mntPoint** 変数で定義されたディレクトリー内の全ファイルをリスト表示します。

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

5. **myubi1** という名前のコンテナを実行し、ホストのボリューム名 **hostvolume** で定義したディレクトリーをコンテナの **/containervolume1** ディレクトリーにマッピングします。

```
$ podman run -it --name myubi1 -v hostvolume:/containervolume1  
registry.access.redhat.com/ubi9/ubi /bin/bash
```

mntPoint 変数 (**-v \$mntPoint:/containervolume1**) で定義したボリュームパスを使用する場合には、**podman volume prune** コマンドを実行すると未使用のボリュームが削除され、データが失われる場合がある点に注意してください。常に **-v hostvolume_name:/containervolume_name** を使用します。

6. コンテナ上にある共有ボリューム内のファイルをリスト表示します。

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

ホスト上で作成した **host.txt** ファイルが表示されます。

7. **/containervolume1** ディレクトリーにテキストファイルを作成します。

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

8. **CTRL+p** および **CTRL+q** を使用してコンテナからデタッチします。
9. ホスト上にある共有ボリューム内のファイルをリスト表示します。以下の2つのファイルが表示されるはずです。

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

この時点で、コンテナとホスト間でファイルを共有しています。2つのコンテナ間でファイルを共有するには、**myubi2** という名前の別のコンテナを実行します。

10. **myubi2** という名前のコンテナを実行し、ホストのボリューム名 **hostvolume** で定義したディレクトリーをコンテナの **/containervolume2** ディレクトリーにマッピングします。

```
$ podman run -it --name myubi2 -v hostvolume:/containervolume2  
registry.access.redhat.com/ubi9/ubi /bin/bash
```

11. コンテナ上にある共有ボリューム内のファイルをリスト表示します。

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

ホストで作成した **host.txt** ファイルと、**myubi1** コンテナ内に作成した **container1.txt** ファイルが表示されます。

12. **/containervolume2** ディレクトリーにテキストファイルを作成します。

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

13. **CTRL+p** および **CTRL+q** を使用してコンテナからデタッチします。
14. ホスト上にある共有ボリューム内のファイルをリスト表示します。以下の3つのファイルが表示されるはずです。

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

関連情報

- **podman-volume** の man ページ

5.12. コンテナのエクスポートおよびインポート

podman export コマンドを使用して、実行中のコンテナのファイルシステムをローカルマシンの tarball にエクスポートできます。たとえば、頻繁に使用しない大規模なコンテナがある場合、スナップショットを保存して後で復元できるようにする場合には、**podman export** コマンドを使用して、実行中のコンテナの現在のスナップショットを tarball にエクスポートできます。

podman import コマンドを使用して tarball をインポートし、ファイルシステムイメージとして保存できます。これにより、このファイルシステムイメージを実行するか、他のイメージのレイヤーとして使用できます。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. **registry.access.redhat.com/ubi9/ubi** イメージに基づいて、**myubi** コンテナを実行します。

```
$ podman run -dt --name=myubi registry.access.redhat.com/9/ubi
```

2. オプション: すべてのコンテナをリスト表示します。

```
$ podman ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS
PORTS NAMES
a6a6d4896142 registry.access.redhat.com/9:latest /bin/bash 7 seconds ago Up 7
seconds ago myubi
```

3. **myubi** コンテナに割り当てます。

```
$ podman attach myubi
```

4. **testfile** という名前のファイルを作成します。

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

5. **CTRL+p** および **CTRL+q** を使用してコンテナからデタッチします。

- ローカルマシンで、**myubi** のファイルシステムを **myubi-container.tar** としてエクスポートします。

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

- オプション: 現在のディレクトリーの内容をリスト表示します。

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

- オプション: **myubi-container** ディレクトリーを作成し、**myubi-container.tar** アーカイブからすべてのファイルをデプロイメントします。ツリー形式で **myubi-directory** の内容をリスト表示します。

```
$ 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.tar にコンテナのファイルシステムが含まれていることを確認できます。

- myubi.tar** をインポートして、ファイルシステムイメージとして保存します。

```
$ 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
```

- すべてのイメージをリスト表示します。

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

11. **testfile** ファイルの内容を表示します。

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

関連情報

- **podman-export** man ページ
- **podman-import** man ページ

5.13. コンテナの停止

実行中のコンテナを停止するには、**podman stop** コマンドを使用します。コンテナ ID または名前を使用して、コンテナを指定できます。

前提条件

- **container-tools** メタパッケージがインストールされている。
- 1つ以上のコンテナが実行中である。

手順

- **myubi** コンテナを停止します。
 - コンテナ名を使用する場合

```
$ podman stop myubi
```

- コンテナ ID を使用する場合

```
$ podman stop 1984555a2c27
```

端末セッションに接続されている、実行中のコンテナを停止するには、コンテナで **exit** コマンドを入力してください。

podman stop コマンドは、SIGTERM シグナルを送信し、実行中のコンテナを終了します。定義された期間 (デフォルトでは 10 秒) 後にコンテナが停止しない場合は、Podman は SIGKILL シグナルを送信します。

また、**podman kill** コマンドを使用して、コンテナを強制終了 (SIGKILL) するか、別のシグナルをコンテナに送信することもできます。以下は、SIGHUP シグナルをコンテナに送信する例です (アプリケーションでサポートされていると、SIGHUP により、アプリケーションが設定ファイルを再読み取りします)。

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

関連情報

- **podman-stop** の man ページ
- **podman-kill** の man ページ

5.14. コンテナの削除

podman rm コマンドを使用して、コンテナを削除します。コンテナ ID または名前コンテナを指定できます。

前提条件

- **container-tools** メタパッケージがインストールされている。
- 1つ以上のコンテナが停止されている。

手順

1. (実行中または停止中の) すべてのコンテナを一覧表示するには、以下を実行します。

```
$ podman ps -a
CONTAINER ID IMAGE          COMMAND          CREATED   STATUS    PORTS NAMES
IS INFRA
d65aecc325a4 ubi9/ubi       /bin/bash       3 secs ago Exited (0) 5 secs ago peaceful_hopper false
74b1da000a11 rhel9/rsyslog rsyslog.sh      2 mins ago Up About a minute musing_brown
false
```

2. コンテナを削除します。

- **peaceful_hopper** を削除するには以下を実行します。

```
$ podman rm peaceful_hopper
```

peaceful_hopper コンテナが終了ステータスであったことに注意してください。コンテナが停止されているので、即座に削除できます。

- **musing_brown** コンテナを削除するには、まずコンテナを停止してから削除します。

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

注記

- 複数のコンテナを削除するには、以下を実行します。

```
$ podman rm clever_yonath furious_shockley
```

- ローカルシステムからすべてのコンテナを削除するには、以下のコマンドを実行します。

```
$ podman rm -a
```

関連情報

- `podman-rm` の man ページ

5.15. コンテナの SELINUX ポリシーの作成

コンテナの SELinux ポリシーを生成するには、UDICA ツールを使用します。詳細は [udica の SELinux ポリシージェネレーターの概要](#) を参照してください。

5.16. PODMAN での事前実行フックの設定

プラグインスクリプトを作成して、コンテナ操作の詳細な制御を定義できます。特に、コンテナイメージのプル、実行、リストなどの不正なアクションをブロックできます。



注記

ファイル `/etc/containers/podman_preexec_hooks.txt` は管理者が作成する必要があり、空であってもかまいません。`/etc/containers/podman_preexec_hooks.txt` が存在しない場合、プラグインスクリプトは実行されません。

次のルールがプラグインスクリプトに適用されます。

- root 所有である必要があり、書き込み可能ではありません。
- `/usr/libexec/podman/pre-exec-hooks` および `/etc/containers/pre-exec-hooks` ディレクトリに配置する必要があります。
- 英数字順に順次実行します。
- すべてのプラグインスクリプトがゼロ値を返す場合、`podman` コマンドが実行されます。
- いずれかのプラグインスクリプトがゼロ以外の値を返した場合、それは失敗を示します。`podman` コマンドは終了し、最初に失敗したスクリプトのゼロ以外の値を返します。
- Red Hat では、スクリプトを正しい順序で実行するために、次の命名規則を使用することを推奨します。**DDD_name.lang**。それぞれ、次のとおりです。
 - **DDD** は、スクリプトの実行順序を示す 10 進数です。必要に応じて、先頭に 1 つまたは 2 つのゼロを使用します。
 - **name** はプラグインスクリプトの名前です。
 - **lang** (オプション) は、指定されたプログラミング言語のファイル拡張子です。たとえば、プラグインスクリプトの名前は次のようになります。**001-check-groups.sh**



注記

プラグインスクリプトは作成時点で有効です。プラグインスクリプトの前に作成されたコンテナは影響を受けません。

前提条件

- `container-tools` メタパッケージがインストールされている。

手順

- **001-check-groups.sh** という名前のスクリプトプラグインを作成します。以下に例を示します。

```
#!/bin/bash
if id -nG "$USER" 2> /dev/null | grep -qw "$GROUP" 2> /dev/null ; then
    exit 0
else
    exit 1
fi
```

- スクリプトは、ユーザーが指定されたグループに属しているかどうかを確認します。
- **USER** と **GROUP** は、Podman によって設定される環境変数です。
- **001-check-groups.sh** スクリプトによって提供される終了コードは、**podman** バイナリーに提供されます。
- **podman** コマンドは終了し、最初に失敗したスクリプトのゼロ以外の値を返します。

検証

- **001-check-groups.sh** スクリプトが正しく動作するかどうかを確認します。

```
$ podman run image
...
```

ユーザーが正しいグループに属していない場合は、次のエラーが表示されます。

```
external preexec hook /etc/containers/pre-exec-hooks/001-check-groups.sh failed
```

5.17. コンテナ内のアプリケーションのデバッグ

トラブルシューティングのさまざまな側面に合わせてカスタマイズされたさまざまなコマンドラインツールを使用できます。詳細は、[コンテナ内のアプリケーションのデバッグ](#)を参照してください。

第6章 コンテナランタイムの選択

runc と crun はコンテナランタイムであり、どちらも OCI ランタイム仕様を実装しているため、同じ意味で使用できます。crun コンテナランタイムには、runc よりも高速で必要なメモリーが少ないため、いくつかの利点があります。そのため、crun コンテナランタイムの使用が推奨されるコンテナランタイムです。

6.1. RUNC コンテナランタイム

runc コンテナランタイムは、Open Container Initiative (OCI) コンテナランタイム仕様の軽量で移植可能な実装です。runc ランタイムは多くの低レベルコードを Docker と共有しますが、Docker プラットフォームのコンポーネントには依存しません。runc は Linux 名前空間とライブ移行をサポートしており、移植可能なパフォーマンスプロファイルがあります。

また、SELinux、コントロールグループ (cgroups)、seccomp などの Linux セキュリティー機能に完全に対応します。runc でイメージをビルドして実行したり、runc で OCI 互換イメージを実行したりできます。

6.2. CRUN コンテナランタイム

crun は、C 言語で書かれた高速および低メモリーフットプリントの OCI コンテナランタイムで、crun バイナリーは runc バイナリーの最大 1/50 のサイズで、2 倍の速度です。crun を使用して、コンテナの実行時に最小限のプロセス数を設定することもできます。crun ランタイムは OCI フックもサポートしています。

crun の追加機能には、以下が含まれます。

- ルートレスコンテナのグループによるファイルの共有
- OCI フックの標準出力および標準エラーの制御
- cgroup v2 で古いバージョンの **systemd** を実行する
- 他のプログラムによって使用される C ライブラリー
- 拡張性
- 移植性

関連情報

- [An introduction to crun, a fast and low-memory footprint container runtime](#)

6.3. RUNC および CRUN でのコンテナの実行

runc または crun では、コンテナはバンドルを使用して設定されます。コンテナのバンドルは、**config.json** という名前の仕様ファイルと、root ファイルシステムを含むディレクトリーです。root ファイルシステムには、コンテナの内容が含まれます。



注記

<runtime> は crun または runc です。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. **registry.access.redhat.com/ubi9/ubi** コンテナイメージをプルします。

```
# podman pull registry.access.redhat.com/ubi9/ubi
```

2. **registry.access.redhat.com/ubi9/ubi** イメージを **rhel.tar** アーカイブにエクスポートします。

```
# podman export $(podman create registry.access.redhat.com/ubi9/ubi) > rhel.tar
```

3. **bundle/rootfs** ディレクトリーを作成します。

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

4. **rhel.tar** アーカイブを **bundle/rootfs** ディレクトリーにデプロイメントします。

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

5. バンドル用に **config.json** という名前の新規仕様ファイルを作成します。

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

- **-b** オプションは、バンドルのディレクトリーを指定します。デフォルト値は現在のディレクトリーです。

6. オプション: 設定を変更します。

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

7. バンドル用に **myubi** という名前のコンテナのインスタンスを作成します。

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

8. **myubi** コンテナを起動します。

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



注記

コンテナインスタンスの名前は、ホストで一意的である必要があります。コンテナの新規インスタンスを起動するには、**# <runtime> start <container_name>** を実行します。

検証

- **<runtime>** によって起動したコンテナをリスト表示します。

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

関連情報

- [crun の man ページ](#)
- [runc の man ページ](#)
- [An introduction to crun, a fast and low-memory footprint container runtime](#)

6.4. コンテナランタイムの一時的な変更

`podman run` コマンドに `--runtime` オプションを指定して使用して、コンテナのランタイムを変更できます。



注記

<runtime> は `crun` または `runc` です。

前提条件

- `container-tools` メタパッケージがインストールされている。

手順

1. `registry.access.redhat.com/ubi9/ubi` コンテナイメージをプルします。

```
$ podman pull registry.access.redhat.com/ubi9/ubi
```

2. `--runtime` オプションを使用してコンテナランタイムを変更します。

```
$ podman run --name=myubi -dt --runtime=<runtime> ubi9
e4654eb4df12ac031f1d0f2657dc4ae6ff8eb0085bf114623b66cc664072e69b
```

3. オプション:すべてのイメージをリスト表示します。

```
$ podman ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS
PORTS NAMES
e4654eb4df12 registry.access.redhat.com/ubi9:latest bash 4 seconds ago Up 4 seconds
ago myubi
```

検証

- OCI ランタイムが `myubi` コンテナで <runtime> に設定されていることを確認します。

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

関連情報

- [An introduction to crun, a fast and low-memory footprint container runtime](#)

6.5. コンテナランタイムの永続的な変更

コンテナランタイムおよびそのオプションを、`/etc/containers/containers.conf` 設定ファイル (root ユーザーとして) または `$HOME/.config/containers/containers.conf` 設定ファイル (非 root ユーザーとして) で設定できます。



注記

`<runtime>` は `crun` または `runc` ランタイムです。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. `/etc/containers/containers.conf` ファイルでランタイムを変更します。

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

2. `myubi` という名前のコンテナを実行します。

```
# podman run --name=myubi -dt ubi9 bash
Resolved "ubi9" as an alias (/etc/containers/registries.conf.d/001-rhel-shortnames.conf)
Trying to pull registry.access.redhat.com/ubi9:latest...
...
Storing signatures
```

検証

- OCI ランタイムが `myubi` コンテナで `<runtime>` に設定されていることを確認します。

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

関連情報

- [An introduction to crun, a fast and low-memory footprint container runtime](#)
- **containers.conf** の man ページ

第7章 UBI コンテナへのソフトウェアの追加

UBI (Red Hat Universal Base Images) は、RHEL コンテンツのサブセットから構築されます。UBI は、UBI でインストールするために自由に利用可能な RHEL パッケージのサブセットも提供します。実行中のコンテナにソフトウェアを追加または更新するには、RPM パッケージおよび更新を含む `dnf` リポジトリを使用します。UBI は、Python、Perl、Node.js、Ruby などの事前にビルドされた言語ランタイムコンテナイメージを提供します。

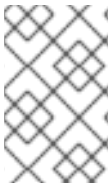
UBI コンテナを実行するパッケージを UBI リポジトリから追加するには、以下を行います。

- UBI init および UBI 標準イメージでは、**dnf** コマンドを使用します。
- UBI minimal イメージでは、**microdnf** コマンドを使用します。



注記

実行中のコンテナでソフトウェアパッケージを直接インストールして作業すると、パッケージを一時的に追加します。この変更はコンテナイメージに保存されません。パッケージの変更を永続化するには、[Buildah を使用した Containerfile からのイメージのビルド](#) セクションを参照してください。



注記

UBI コンテナにソフトウェアを追加する場合は、サブスクライブしている RHEL ホスト、またはサブスクライブしていない (または非 RHEL) システムで UBI を更新する手順が異なります。

7.1. UBI INIT イメージの使用

Containerfile を使用してコンテナをビルドできます。この **Containerfile** は、コンテナがホストシステムで実行されている場合に Web サーバー (**httpd**) をインストールして、**systemd** サービス (**/sbin/init**) により Web サーバーが自動的に起動されるように設定します。**podman build** コマンドは、1つ以上の **Containerfiles** および指定されたビルドコンテキストディレクトリー内の命令を使用してイメージをビルドします。コンテキストディレクトリーは、アーカイブ、Git リポジトリ、または **Containerfile** の URL として指定できます。コンテキストディレクトリーが指定されていない場合、現在の作業ディレクトリーはビルドコンテキストと見なされ、**Containerfile** が含まれている必要があります。**--file** オプションで **Containerfile** を指定することもできます。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. 新しいディレクトリーに以下の内容を含む **Containerfile** を作成します。

```
FROM registry.access.redhat.com/ubi9/ubi-init
RUN dnf -y install httpd; dnf 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 は、**httpd** パッケージをインストールし、ブート時に **httpd** サービスが開始できるようにし、テストファイル (**index.html**) を作成し、Web サーバーをホスト (ポート 80) に公開し、コンテナの起動時、**systemd** init サービス (**/sbin/init**) を開始します。

2. コンテナをビルドします。

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

3. オプション: **systemd** を使用してコンテナを実行する場合、システムで SELinux が有効になっている場合は、**container_manage_cgroup** ブール変数を設定する必要があります。

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

4. **mysysd_run** という名前のコンテナを実行します。

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

mysysd イメージが **mysysd_run** コンテナをデーモンプロセスとして実行し、コンテナのポート 80 がホストシステムのポート 80 に公開されます。



注記

ルートレスモードでは、1024 以上のホストのポート番号を選択する必要があります。以下に例を示します。

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

1024 未満のポート番号を使用するには、**net.ipv4.ip_unprivileged_port_start** 変数を変更する必要があります。

```
# sysctl net.ipv4.ip_unprivileged_port_start=80
```

5. コンテナが実行されていることを確認します。

```
# 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. Web サーバーをテストします。

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

関連情報

- [ルートレス Podman の欠点](#)

7.2. UBI マイクロイメージの使用

Buildah ツールを使用して **ubi-micro** コンテナイメージをビルドできます。

前提条件

- **container-tools** メタパッケージがインストールされている。

前提条件

- **containers-tool** メタパッケージが提供する **podman** ツールがインストールされている。

手順

1. **registry.access.redhat.com/ubi8/ubi-micro** イメージをプルしてビルドします。

```
# microcontainer=$(buildah from registry.access.redhat.com/ubi9/ubi-micro)
```

2. 作業中のコンテナの root ファイルシステムをマウントします。

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

3. **httpd** サービスを **micromount** ディレクトリーにインストールします。

```
# dnf install \
  --installroot $micromount \
  --releasever=/ \
  --setopt install_weak_deps=false \
  --setopt=reposdir=/etc/yum.repos.d/ \
  --nodocs -y \
  httpd
# dnf clean all \
  --installroot $micromount
```

4. 作業コンテナで root ファイルシステムをアンマウントします。

```
# buildah umount $microcontainer
```

5. 作業コンテナから **ubi-micro-httpd** イメージを作成します。

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

検証手順

1. **ubi-micro-httpd** イメージの詳細を表示します。

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

7.3. UBI コンテナへのソフトウェアの追加 (サブスクライブされたホスト)

登録およびサブスクライブされた RHEL ホストで UBI コンテナを実行している場合は、RHEL Base リポジトリーおよび AppStream リポジトリーが、標準の UBI コンテナとすべての UBI リポジトリーで有効になります。

関連情報

- [Universal Base Images \(UBI\):Images, repositories, packages, and source code](#) を参照してください。

7.4. 標準の UBI コンテナへのソフトウェアの追加

標準の UBI コンテナ内にソフトウェアを追加するには、UBI 以外の dnf リポジトリを無効にして、ビルドするコンテナが再配布されるようにします。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. **registry.access.redhat.com/ubi9/ubi** イメージをプルして実行します。

```
$ podman run -it --name myubi registry.access.redhat.com/ubi9/ubi
```

2. **myubi** コンテナにパッケージを追加します。

- UBI リポジトリにあるパッケージを追加するには、UBI リポジトリ以外の dnf リポジトリをすべて無効にします。たとえば、**bzip2** パッケージを追加するには、次のコマンドを実行します。

```
# dnf install --disablerepo=* --enablerepo=ubi-8-appstream-rpms --enablerepo=ubi-8-baseos-rpms bzip2
```

- UBI リポジトリにないパッケージを追加するには、リポジトリを無効にしないでください。たとえば、**zsh** パッケージを追加するには、次のコマンドを実行します。

```
# dnf install zsh
```

- 別のホストリポジトリにあるパッケージを追加するには、必要なリポジトリを明示的に有効にします。たとえば、**codeready-builder-for-rhel-8-x86_64-rpms** リポジトリから **python38-devel** パッケージをインストールするには、以下のコマンドを実行します。

```
# dnf install --enablerepo=codeready-builder-for-rhel-8-x86_64-rpms python38-devel
```

検証手順

1. コンテナ内で有効なリポジトリのリストを表示します。

```
# dnf repolist
```

2. 必要なリポジトリがリスト表示されていることを確認します。
3. インストール済みパッケージのリストを表示します。

```
# rpm -qa
```

4. 必要なパッケージがリスト表示されていることを確認します。



注記

Red Hat UBI リポジトリに含まれていない Red Hat パッケージをインストールすると、サブスクライブしている RHEL システム以外でコンテナを配布する機能が限定される可能性があります。

7.5. UBI MINIMAL コンテナへのソフトウェアの追加

UBI dnf リポジトリは、デフォルトで UBI minimal イメージで有効になります。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. **registry.access.redhat.com/ubi9/ubi-minimal** イメージをプルして実行します。

```
$ podman run -it --name myubimin registry.access.redhat.com/ubi9/ubi-minimal
```

2. **myubimin** コンテナにパッケージを追加します。

- UBI リポジトリにあるパッケージを追加するには、リポジトリを無効にしないでください。たとえば、**bzip2** パッケージを追加するには、次のコマンドを実行します。

```
# microdnf install bzip2
```

- 別のホストリポジトリにあるパッケージを追加するには、必要なりポジトリを明示的に有効にします。たとえば、**codeready-builder-for-rhel-8-x86_64-rpms** リポジトリから **python38-devel** パッケージをインストールするには、以下のコマンドを実行します。

```
# microdnf install --enablerepo=codeready-builder-for-rhel-8-x86_64-rpms python38-devel
```

検証手順

1. コンテナ内で有効なりポジトリのリストを表示します。

```
# microdnf repolist
```

2. 必要なりポジトリがリスト表示されていることを確認します。
3. インストール済みパッケージのリストを表示します。

```
# rpm -qa
```

4. 必要なパッケージがリスト表示されていることを確認します。



注記

Red Hat UBI リポジトリに含まれていない Red Hat パッケージをインストールすると、サブスクライブしている RHEL システム以外でコンテナを配布する機能が限定される可能性があります。

7.6. UBI コンテナへのソフトウェアの追加 (サブスクライブしていないホスト)

サブスクライブしていない RHEL システムにソフトウェアパッケージを追加するときに、リポジトリを無効にする必要はありません。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

- UBI 標準または UBI init イメージに基づいて、実行中のコンテナにパッケージを追加します。リポジトリを無効にしないでください。**podman run** コマンドを使用してコンテナを実行し、コンテナ内で **dnf install** コマンドを使用します。
 - たとえば、**bzip2** パッケージを UBI 標準ベースのコンテナに追加するには、次のコマンドを実行します。

```
$ podman run -it --name myubi registry.access.redhat.com/ubi9/ubi
# dnf install bzip2
```

- たとえば、**bzip2** パッケージを UBI init ベースのコンテナに追加するには、次のコマンドを実行します。

```
$ podman run -it --name myubimin registry.access.redhat.com/ubi9/ubi-minimal
# microdnf install bzip2
```

検証手順

1. 有効なリポジトリをリスト表示します。
 - UBI 標準イメージまたは UBI init イメージに基づいて、コンテナ内で有効なリポジトリをすべてリスト表示するには、次のコマンドを実行します。

```
# dnf repolist
```

- UBI minimal コンテナに基づいて、コンテナ内で有効なリポジトリを一覧表示するには、以下を実行します。

```
# microdnf repolist
```

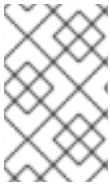
2. 必要なリポジトリがリスト表示されていることを確認します。
3. インストール済みパッケージのリストを表示します。

```
# rpm -qa
```

4. 必要なパッケージがリスト表示されていることを確認します。

7.7. UBI ベースのイメージの構築

Buildah コーティリティーを使用して、**Containerfile** から UBI ベースの Web サーバーコンテナを作成できます。非 UBI dnf リポジトリをすべて無効にして、イメージに再配布できる Red Hat ソフトウェアのみが含まれていることを確認する必要があります。



注記

UBI の最小イメージの場合は、**dnf** の代わりに **microdnf** を使用します。RUN **microdnf update -y && rm -rf /var/cache/yum** および **RUN microdnf install httpd -y && microdnf clean all** コマンド。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. **Containerfile** を作成します。

```
FROM registry.access.redhat.com/ubi9/ubi
USER root
LABEL maintainer="John Doe"
# Update image
RUN dnf update --disablerepo=* --enablerepo=ubi-8-appstream-rpms --enablerepo=ubi-8-
baseos-rpms -y && rm -rf /var/cache/yum
RUN dnf install --disablerepo=* --enablerepo=ubi-8-appstream-rpms --enablerepo=ubi-8-
baseos-rpms 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. コンテナイメージをビルドします。

```
# buildah bud -t johndoe/webserver .
STEP 1: FROM registry.access.redhat.com/ubi9/ubi:latest
STEP 2: USER root
STEP 3: LABEL maintainer="John Doe"
STEP 4: RUN dnf update --disablerepo=* --enablerepo=ubi-8-appstream-rpms --
enablerepo=ubi-8-baseos-rpms -y
...
Writing manifest to image destination
Storing signatures
--> f9874f27050
f9874f270500c255b950e751e53d37c6f8f6dba13425d42f30c2a8ef26b769f2
```

検証手順

1. Web サーバーを実行します。

```
# podman run -d --name=myweb -p 80:80 johndoe/webserver
bbe98c71d18720d966e4567949888dc4fb86eec7d304e785d5177168a5965f64
```

2. Web サーバーをテストします。

```
# curl http://localhost/index.html
The Web Server is Running
```

7.8. APPLICATION STREAM ランタイムイメージの使用

[Application Streams](#) に基づくランタイムイメージは、コンテナビルドのベースとして使用できる一連のコンテナイメージを提供します。

サポートされるランタイムイメージは Python、Ruby、s2-core、s2i-base、.NET Core、PHP です。ランタイムイメージは [Red Hat Container Catalog](#) で利用できます。



注記

このような UBI イメージコンテナには、従来のイメージと同じ基本ソフトウェアが含まれているため、詳細は [Using Red Hat Software Collections Container Images](#) を参照してください。

関連情報

- [Red Hat Container Catalog](#)
- [Red Hat Container Image Updates](#)

7.9. UBI コンテナイメージソースコードの取得

すべての Red Hat UBI ベースのイメージのソースコードは、ダウンロード可能なコンテナイメージの形で入手できます。コンテナとしてパッケージ化されているにもかかわらず、ソースコンテナイメージを実行できません。システムに Red Hat ソースコンテナイメージをインストールするには、**podman pull** コマンドではなく、**skopeo** コマンドを使用します。

ソースコンテナイメージは、それらが表すバイナリーコンテナに基づいて名前が付けられます。たとえば、ある標準的な RHEL UBI 9 コンテナでは、**registry.access.redhat.com/ubi9:8.1-397** に **-source** を追加してソースコンテナイメージを取得します (**registry.access.redhat.com/ubi9:8.1-397-source**)。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. **skopeo copy** コマンドを使用して、ソースコンテナイメージをローカルディレクトリーにコピーします。

```
$ skopeo copy \
docker://registry.access.redhat.com/ubi9:8.1-397-source \
dir:$HOME/TEST
...
Copying blob 477bc8106765 done
Copying blob c438818481d3 done
```

```
...
Writing manifest to image destination
Storing signatures
```

2. **skopeo inspect** コマンドを使用して、ソースコンテナイメージを検査します。

```
$ 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. すべてのコンテンツをデプロイメントします。

```
$ cd $HOME/TEST
$ for f in $(ls); do tar xvf $f; done
```

4. 結果を確認します。

```
$ find blobs/ rpm_dir/
blobs/
blobs/sha256
blobs/sha256/10914f1fff060ce31388f5ab963871870535aaaa551629f5ad182384d60fdf82
rpm_dir/
rpm_dir/gzip-1.9-4.el8.src.rpm
```

結果が正しい場合は、イメージを使用できる状態になります。



注記

コンテナイメージがリリースされてから、関連するソースコンテナが利用可能になるまでに数時間かかる場合があります。

関連情報

- **skopeo-copy** の man ページ

- **skopeo-inspect** の man ページ

第8章 コンテナイメージへの署名

GNU Privacy Guard (GPG) 署名または sigstore 署名を使用して、コンテナイメージに署名できます。両方の署名手法は、通常、OCI 準拠のコンテナレジストリーと互換性があります。Podman を使用して、リモートレジストリーにプッシュする前にイメージに署名し、署名されていないイメージが拒否されるようにコンシューマーを設定できます。コンテナイメージに署名すると、サプライチェーンへの攻撃を防ぐことができます。

GPG キーを使用して署名するには、署名を配布するために別のルックアサイドサーバーをデプロイメントする必要があります。ルックアサイドサーバーは、任意の HTTP サーバーにすることができます。Podman バージョン 4.2 以降では、コンテナ署名の sigstore 形式を使用できます。GPG キーと比較して、sigstore 署名はコンテナレジストリーに格納されるため、個別のルックアサイドサーバーは必要ありません。

8.1. GPG 署名を使用したコンテナイメージへの署名

GNU Privacy Guard (GPG) キーを使用してイメージに署名できます。

前提条件

- **container-tools** メタパッケージがインストールされている。
- GPG ツールがインストールされます。
- ルックアサイド Web サーバーがセットアップされ、その上でファイルを公開できます。
 - システム全体のレジストリー設定は、`/etc/containers/registries.d/default.yaml` ファイルで確認できます。**lookaside-staging** オプションは、署名を書き込むためのファイルパスを参照し、通常、署名を発行するホストで設定されます。

```
# cat /etc/containers/registries.d/default.yaml
docker:
  <registry>:
    lookaside: https://registry-lookaside.example.com
    lookaside-staging: file:///var/lib/containers/sigstore
  ...
```

手順

1. GPG キーを生成します。

```
# gpg --full-gen-key
```

2. 公開鍵をエクスポートします。

```
# gpg --output <path>/key.gpg --armor --export <username@domain.com>
```

3. 現在のディレクトリーで **Containerfile** を使用してコンテナイメージをビルドします。

```
$ podman build -t <registry>/<namespace>/<image>
```

`<registry>`、`<namespace>`、および `<image>` をコンテナイメージ識別子に置き換えます。詳細については、[コンテナレジストリー](#) を参照してください。

4. イメージに署名し、レジストリーにプッシュします。

```
$ podman push \
  --sign-by <username@domain.com> \
  <registry>/<namespace>/<image>
```



注記

既存のイメージをコンテナレジストリー間で移動するときに署名する必要がある場合は、**skopeo copy** コマンドを使用できます。

5. オプション: 新しいイメージングネチャーを表示します。

```
# (cd /var/lib/containers/sigstore/; find . -type f)
./<image>@sha256=<digest>/signature-1
```

6. ローカル署名をルックアサイド Web サーバーにコピーします。

```
# rsync -a /var/lib/containers/sigstore <user@registry-lookaside.example.com>:/registry-lookaside/webroot/sigstore
```

署名は、**lookaside-staging** オプションによって決定される場所 (この場合は **/var/lib/containers/sigstore** ディレクトリー) に保存されます。

検証

- 詳細は、[GPG イメージ署名の検証](#) を参照してください。

関連情報

- **podman-image-trust** の man ページ
- **podman-push** の man ページ
- **podman-build** の man ページ
- [GPG キーペアの生成方法](#)

8.2. GPG イメージ署名の検証

次の手順を使用して、コンテナイメージが GPG キーで正しく署名されていることを確認できます。

前提条件

- **container-tools** メタパッケージがインストールされている。
- 署名読み取り用の Web サーバーがセットアップされ、その上でファイルを公開できます。
 - システム全体のレジストリー設定は、**/etc/containers/registries.d/default.yaml** ファイルで確認できます。**lookaside** オプションは、署名読み取り用の Web サーバーを参照します。署名を検証するには、**lookaside** オプションを設定する必要があります。

```
# cat /etc/containers/registries.d/default.yaml
```



```
docker:
  <registry>:
    lookaside: https://registry-lookaside.example.com
    lookaside-staging: file:///var/lib/containers/sigstore
  ...
```

手順

1. **<registry>** の信頼範囲を更新します。

```
$ podman image trust set -f <path>/key.gpg <registry>/<namespace>
```

2. オプション: **/etc/containers/policy.json** ファイルを表示して、信頼ポリシーの設定を確認します。

```
$ cat /etc/containers/policy.json
{
  ...
  "transports": {
    "docker": {
      "<registry>/<namespace>": [
        {
          "type": "signedBy",
          "keyType": "GPGKeys",
          "keyPath": "<path>/key.gpg"
        }
      ]
    }
  }
}
```



注記

通常、**/etc/containers.policy.json** ファイルは、同じキーが使用される組織のレベルで設定されます。たとえば、公開レジストリーの場合は **<registry>/<namespace>**、単一企業専用レジストリーの場合は単に **<registry>** です。

3. イメージをプルします:

```
# podman pull <registry>/<namespace>/<image>
...
Storing signatures
e7d92cdc71feacf90708cb59182d0df1b911f8ae022d29e8e95d75ca6a99776a
```

podman pull コマンドは、設定どおりに署名の存在を強制します。追加のオプションは必要ありません。



注記

`/etc/containers/registries.d/default.yaml` ファイルで、システム全体のレジストリー設定を編集できます。`/etc/containers/registries.d` ディレクトリーにある任意の YAML ファイルのレジストリーまたはリポジトリー設定セクションを編集することもできます。すべての YAML ファイルが読み取られ、ファイル名は任意です。単一のスコープ (default-docker、レジストリー、または名前空間) は、`/etc/containers/registries.d` ディレクトリー内の1つのファイルにのみ存在できます。



重要

`/etc/containers/registries.d/default.yaml` ファイルのシステム全体のレジストリー設定により、公開された署名にアクセスできます。**sigstore** および **sigstore-staging** オプションは非推奨になりました。これらのオプションは署名ストレージを参照しており、sigstore 署名形式には関連付けられていません。代わりに、新しい同等の **lookaside** および **lookaside-staging** オプションを使用してください。

関連情報

- **podman-image-trust** の man ページ
- **podman-pull** の man ページ

8.3. 秘密鍵を使用した SIGSTORE 署名によるコンテナイメージの署名

Podman バージョン 4.2 以降では、コンテナ署名の sigstore 形式を使用できます。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. sigstore 公開鍵/秘密鍵ペアを生成します。

```
$ skopeo generate-sigstore-key --output-prefix myKey
```

- 公開鍵と秘密鍵 **myKey.pub** と **myKey.private** が生成されます。



注記

skopeo generate-sigstore-key コマンドは、RHEL 9.2 から利用できます。それ以外の場合は、アップストリームの Cosign プロジェクトを使用して、公開鍵と秘密鍵のペアを生成する必要があります。

- cosign ツールをインストールします。

```
$ git clone -b v2.0.0 https://github.com/sigstore/cosign
$ cd cosign
$ make ./cosign
```

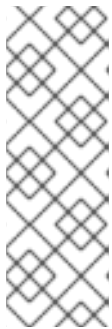
- 公開鍵/秘密鍵のペアを生成します。

```
$ ./cosign generate-key-pair
...
Private key written to cosign.key
Public key written to cosign.pub
```

2. 次の内容を **/etc/containers/registries.d/default.yaml** ファイルに追加します。

```
docker:
  <registry>:
    use-sigstore-attachments: true
```

use-sigstore-attachments オプションを設定することで、Podman と Skopeo はコンテナの sigstore 署名をイメージと共に読み書きし、署名されたイメージと同じリポジトリに保存できます。



注記

/etc/containers/registries.d/default.yaml ファイルで、システム全体のレジストリー設定を編集できます。**/etc/containers/registries.d** ディレクトリーにある任意の YAML ファイルのレジストリーまたはリポジトリ設定セクションを編集することもできます。すべての YAML ファイルが読み取られ、ファイル名は任意です。単一のスコープ (default-docker、レジストリー、または名前空間) は、**/etc/containers/registries.d** ディレクトリー内の1つのファイルにのみ存在できます。

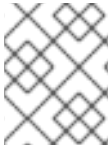
3. 現在のディレクトリーで **Containerfile** を使用してコンテナイメージをビルドします。

```
$ podman build -t <registry>/<namespace>/<image>
```

4. イメージに署名し、レジストリーにプッシュします。

```
$ podman push --sign-by-sigstore-private-key ./myKey.private
<registry>/<namespace>/<image>
```

podman push コマンドは、**<registry>/<namespace>/<image>** ローカルイメージを **<registry>/<namespace>/<image>** としてリモートレジストリーにプッシュします。**--sign-by-sigstore-private-key** オプションは、**myKey.private** 秘密鍵を使用して sigstore 署名を **<registry>/<namespace>/<image>** イメージに追加します。イメージと sigstore 署名がリモートレジストリーにアップロードされます。



注記

既存のイメージをコンテナレジストリー間で移動するときに署名する必要がある場合は、**skopeo copy** コマンドを使用できます。

検証

- 詳細は、[公開鍵を使用した sigstore イメージ署名の検証](#) を参照してください。

関連情報

- **podman-push** の man ページ
- **podman-build** の man ページ
- [Sigstore:ソフトウェアサプライチェーンの信頼とセキュリティーに対するオープンな答え](#)

8.4. 公開鍵を使用した SIGSTORE イメージ署名の検証

次の手順を使用して、コンテナイメージが正しく署名されていることを確認できます。

前提条件

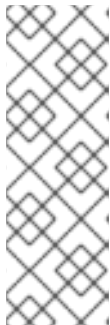
- **container-tools** メタパッケージがインストールされている。

手順

1. 次の内容を **/etc/containers/registries.d/default.yaml** ファイルに追加します。

```
docker:
  <registry>:
    use-sigstore-attachments: true
```

use-sigstore-attachments オプションを設定することで、Podman と Skopeo はコンテナの sigstore 署名をイメージと共に読み書きし、署名されたイメージと同じリポジトリーに保存できます。



注記

/etc/containers/registries.d/default.yaml ファイルで、システム全体のレジストリー設定を編集できます。**/etc/containers/registries.d** ディレクトリーにある任意の YAML ファイルのレジストリーまたはリポジトリー設定セクションを編集することもできます。すべての YAML ファイルが読み取られ、ファイル名は任意です。単一のスコープ (default-docker、レジストリー、または名前空間) は、**/etc/containers/registries.d** ディレクトリー内の1つのファイルにのみ存在できます。

2. **/etc/containers/policy.json** ファイルを編集して、**sigstore** 署名の存在を強制します。

```
...
"transports": {
  "docker": {
    "<registry>/<namespace>": [
      {
```

```

        "type": "sigstoreSigned",
        "keyPath": "/some/path/to/cosign.pub"
      }
    ]
  }
}
...

```

`/etc/containers/policy.json` 設定ファイルを変更することにより、信頼ポリシーの設定を変更します。Podman、Buildah、および Skopeo は、コンテナイメージの署名の存在を強制しません。

3. イメージをプルします:

```
$ podman pull <registry>/<namespace>/<image>
```

`podman pull` コマンドは、設定どおりに署名の存在を強制します。追加のオプションは必要ありません。

関連情報

- [Sigstore:ソフトウェアサプライチェーンの信頼とセキュリティーに対するオープンな答え](#)

8.5. FULCIO と REKOR を使用した SIGSTORE 署名によるコンテナイメージの署名

Fulcio および Rekor サーバーを使用すると、秘密鍵を手動で管理する代わりに、OpenID Connect (OIDC) サーバー認証に基づく短期証明書を使用して署名を作成できるようになりました。

前提条件

- **container-tools** メタパッケージがインストールされている。
- Fulcio (<https://<your-fulcio-server>>) サーバーと Rekor (<https://<your-rekor-server>>) サーバーが実行および設定されている。
- Podman v4.4 以降がインストールされている。

手順

1. 次の内容を `/etc/containers/registries.conf.d/default.yaml` ファイルに追加します。

```

docker:
  <registry>:
    use-sigstore-attachments: true

```

- **use-sigstore-attachments** オプションを設定することで、Podman と Skopeo はコンテナの sigstore 署名をイメージと共に読み書きし、署名されたイメージと同じリポジトリに保存できます。



注記

`/etc/containers/registries.d` ディレクトリー内の任意の YAML ファイルのレジストリーまたはリポジトリー設定セクションを編集できます。単一のスコープ (`default-docker`、レジストリー、または名前空間) は、`/etc/containers/registries.d` ディレクトリー内の1つのファイルにのみ存在できます。`/etc/containers/registries.d/default.yaml` ファイルでシステム全体のレジストリー設定を編集することもできます。すべての YAML ファイルが読み取られ、ファイル名は任意であることに注意してください。

2. `file.yml` ファイルを作成します。

```
fulcio:
  fulcioURL: "https://<your-fulcio-server>"
  oidcMode: "interactive"
  oidcIssuerURL: "https://<your-OIDC-provider>"
  oidcClientID: "sigstore"
  rekorURL: "https://<your-rekor-server>"
```

- `file.yml` は、sigstore 署名の作成に必要なオプションを保存するために使用される sigstore 署名パラメーター YAML ファイルです。
3. イメージに署名し、レジストリーにプッシュします。

```
$ podman push --sign-by-sigstore=file.yml <registry>/<namespace>/<image>
```

- あるいは、同様の `--sign-by-sigstore` オプションを指定して `skopeo copy` コマンドを使用して、既存のイメージをコンテナレジストリー間で移動しながら署名することもできます。



警告

パブリックサーバーへの送信には、公開鍵と証明書に関するデータ、署名に関するメタデータが含まれることに注意してください。

検証

- [Fulcio と Rekor を使用した sigstore 署名を持つコンテナイメージの検証](#)

関連情報

- `containers-sigstore-signing-params.yaml` の man ページ
- `podman-push` の man ページ
- `container-registries.d` man ページ

8.6. FULCIO と REKOR を使用した SIGSTORE 署名を持つコンテナイメージの検証

Fulcio および Rekor 関連の情報を **policy.json** ファイルに追加することで、イメージの署名を検証できます。コンテナイメージの署名を検証すると、イメージが信頼できるソースからのものであり、改ざんまたは変更されていないことが保証されます。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. 次の内容を **/etc/containers/registries.conf.d/default.yaml** ファイルに追加します。

```
docker:
  <registry>:
    use-sigstore-attachments: true
```

- **use-sigstore-attachments** オプションを設定することで、Podman と Skopeo はコンテナの sigstore 署名をイメージと共に読み書きし、署名されたイメージと同じリポジトリに保存できます。



注記

/etc/containers/registries.d ディレクトリー内の任意の YAML ファイルのレジストリーまたはリポジトリ設定セクションを編集できます。単一のスコープ (default-docker、レジストリー、または名前空間) は、**/etc/containers/registries.d** ディレクトリー内の1つのファイルにのみ存在できます。**/etc/containers/registries.d/default.yaml** ファイルでシステム全体のレジストリー設定を編集することもできます。すべての YAML ファイルが読み取られ、ファイル名は任意であることに注意してください。

2. **/etc/containers/policy.json** ファイルに **fulcio** セクションと **rekorPublicKeyPath** フィールドまたは **rekorPublicKeyData** フィールドを追加します。

```
{
  ...
  "transports": {
    "docker": {
      "<registry>/<namespace>": [
        {
          "type": "sigstoreSigned",
          "fulcio": {
            "caPath": "/path/to/local/CA/file",
            "oidcIssuer": "https://expected.OIDC.issuer/",
            "subjectEmail": "expected-signing-user@example.com",
          },
          "rekorPublicKeyPath": "/path/to/local/public/key/file",
        }
      ]
    }
  }
  ...
}
```

- **fulcio** セクションでは、署名が Fulcio 発行の証明書に基づいていることが規定されています。
- Fulcio インスタンスの CA 証明書を含む **caPath** フィールドと **caData** フィールドのいずれかを指定する必要があります。
- **oidcIssuer** と **subjectEmail** は両方とも必須であり、予期される ID プロバイダー、および Fulcio 証明書を取得するユーザーの ID を正確に指定します。
- **rekorPublicKeyPath** フィールドと **rekorPublicKeyData** フィールドのいずれかを指定する必要があります。

3. イメージをプルします:

```
$ podman pull <registry>/<namespace>/<image>
```

podman pull コマンドは、設定どおりに署名の存在を強制します。追加のオプションは必要ありません。

関連情報

- **policy.json** man ページ
- **container-registries.d** man ページ

8.7. 秘密鍵と REKOR を使用した SIGSTORE 署名によるコンテナイメージの署名

Podman バージョン 4.4 以降、Rekor サーバーとともにコンテナ署名の sigstore 形式を使用できるようになりました。パブリック署名をパブリック rekor.sigstore.dev サーバーにアップロードすることもでき、これにより Cosign との相互運用性が向上します。その後、rekor を明示的に無効にしなくても、**cosign verify** コマンドを使用して署名を検証できます。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. sigstore 公開鍵/秘密鍵ペアを生成します。

```
$ skopeo generate-sigstore-key --output-prefix myKey
```

- 公開鍵と秘密鍵 **myKey.pub** と **myKey.private** が生成されます。

2. 次の内容を **/etc/containers/registries.conf.d/default.yaml** ファイルに追加します。

```
docker:
  <registry>:
    use-sigstore-attachments: true
```

- **use-sigstore-attachments** オプションを設定することで、Podman と Skopeo はコンテナの sigstore 署名をイメージと共に読み書きし、署名されたイメージと同じリポジトリに保存できます。



注記

/etc/containers/registries.d ディレクトリー内の任意の YAML ファイルのレジストリーまたはリポジトリー設定セクションを編集できます。単一のスコープ (default-docker、レジストリー、または名前空間) は、**/etc/containers/registries.d** ディレクトリー内の1つのファイルにのみ存在できます。**/etc/containers/registries.d/default.yaml** ファイルでシステム全体のレジストリー設定を編集することもできます。すべての YAML ファイルが読み取られ、ファイル名は任意であることに注意してください。

- 現在のディレクトリーで **Containerfile** を使用してコンテナイメージをビルドします。

```
$ podman build -t <registry>/<namespace>/<image>
```

- file.yml** ファイルを作成します。

```
privateKeyFile: "/home/user/sigstore/myKey.private"
privateKeyPassphraseFile: "/mnt/user/sigstore-myKey-passphrase"
rekorURL: "https://<your-rekor-server>"
```

- file.yml** は、sigstore 署名の作成に必要なオプションを保存するために使用される sigstore 署名パラメーター YAML ファイルです。
- イメージに署名し、レジストリーにプッシュします。

```
$ podman push --sign-by-sigstore=file.yml <registry>/<namespace>/<image>
```

- あるいは、同様の **--sign-by-sigstore** オプションを指定して **skopeo copy** コマンドを使用して、既存のイメージをコンテナレジストリー間で移動しながら署名することもできます。



警告

パブリックサーバーへの送信には、公開鍵に関するデータと署名に関するメタデータが含まれることに注意してください。

検証

- 次のいずれかの方法を使用して、コンテナイメージが正しく署名されていることを確認します。
 - cosign verify** コマンドを使用します。

```
$ cosign verify <registry>/<namespace>/<image> --key myKey.pub
```

- podman pull** コマンドを使用します。
 - /etc/containers/policy.json** ファイルに **rekorPublicKeyPath** フィールドまたは **rekorPublicKeyData** フィールドを追加します。

```
{
  ...
  "transports": {
    "docker": {
      "<registry>/<namespace>": [
        {
          "type": "sigstoreSigned",
          "rekorPublicKeyPath": "/path/to/local/public/key/file",
        }
      ]
    }
  }
  ...
}
```

- イメージをプルします:

```
$ podman pull <registry>/<namespace>/<image>
```

- **podman pull** コマンドは、設定どおりに署名の存在を強制します。追加のオプションは必要ありません。

関連情報

- **podman-push** の man ページ
- **podman-build** の man ページ
- **container-registries.d** man ページ
- [Sigstore:ソフトウェアサプライチェーンの信頼とセキュリティーに対するオープンな答え](#)

第9章 コンテナネットワークの管理

本章では、コンテナ間の通信方法について説明します。

9.1. コンテナネットワークのリストアップ

Podman には、ルートレスとルートフルという 2 つのネットワークの動作があります。

- ルートレスネットワーク: ネットワークは自動的に設定され、コンテナには IP アドレスがありません。
- ルートフルネットワーク: コンテナには IP アドレスがあります。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

- ルートユーザーで全ネットワークをリスト表示します。

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

- デフォルトでは、Podman はブリッジネットワークを提供します。
- ルートレスユーザーのネットワークリストは、ルートフルユーザーと同じです。

関連情報

- **podman-network-ls** の man ページ

9.2. ネットワークの検査

podman network ls コマンドでリストアップされた特定のネットワークの IP 範囲、有効なプラグイン、ネットワークのタイプなどを表示します。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

- デフォルトの **Podman** ネットワークを検査します。

```
$ 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 範囲、有効なプラグイン、ネットワークの種類など、ネットワークの設定を確認できます。

関連情報

- `podman-network-inspect` の man ページ

9.3. ネットワークの作成

新しいネットワークを作成するには、**Podman network create** コマンドを使用します。



注記

デフォルトでは、Podman は外部ネットワークを作成します。**podman network create --internal** コマンドで内部ネットワークを作成することができます。内部ネットワーク内のコンテナは、ホスト上の他のコンテナと通信できますが、ホスト外のネットワークに接続したり、ホストから到達したりすることはできません。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

- **mynet** という名前の外部ネットワークを作成します。

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

検証

- すべてのネットワークをリストアップします。

```
# 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
```

作成された **mynet** ネットワークとデフォルトの **Podman** ネットワークが表示されます。



注記

Podman 4.0 以降、**podman network create** コマンドを使用して新しい外部ネットワークを作成すると、デフォルトで DNS プラグインが有効になります。

関連情報

- **podman-network-create** man ページ

9.4. コンテナのネットワークへの接続

コンテナをネットワークに接続するには、**Podman network connect** コマンドを使用します。

前提条件

- **container-tools** メタパッケージがインストールされている。
- **podman network create** コマンドでネットワークが作成されている。
- コンテナが作成されている。

手順

- **mycontainer** という名前のコンテナを **mynet** という名前のネットワークに接続します。

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

検証

- **mycontainer** が **mynet** ネットワークに接続されていることを確認します。

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

mycontainer が **mynet** と **podman** のネットワークに接続されていることがわかります。

関連情報

- **podman-network-connect** の man ページ

9.5. コンテナのネットワークからの切断

コンテナをネットワークから切断するには、**podman network disconnect** コマンドを使用します。

前提条件

- **container-tools** メタパッケージがインストールされている。
- **podman network create** コマンドでネットワークが作成されている。
- コンテナがネットワークに接続されている。

手順

- **mycontainer** というコンテナを **mynet** というネットワークから切り離します。

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

検証

- **mycontainer** が **mynet** ネットワークから切断されていることを確認します。

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

mycontainer が **mynet** ネットワークから切断され、**mycontainer** がデフォルトの **Podman** ネットワークにのみ接続されていることがわかります。

関連情報

- **podman-network-disconnect** の man ページ

9.6. ネットワークの削除

指定したネットワークを削除するには、**podman network rm** コマンドを使用します。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. すべてのネットワークをリストアップします。

-

```
# 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** ネットワークを削除します。

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



注記

削除されたネットワークに関連するコンテナがある場合は、**Podman network rm -f** コマンドを使用してコンテナと Pod を削除する必要があります。

検証

- **mynet** ネットワークが削除されたかどうかを確認します。

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

関連情報

- **podman-network-rm** の man ページ

9.7. 未使用の全ネットワークの削除

podman network prune で、使われていないネットワークをすべて削除してください。未使用のネットワークとは、コンテナが接続されていないネットワークのことです。**podman network prune** コマンドでは、デフォルトの **podman** ネットワークは削除されません。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

- 未使用のネットワークをすべて削除します。

```
# 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
```

検証

- すべてのネットワークが削除されたことを確認します。

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

関連情報

- [podman-network-prune](#) の man ページ

第10章 POD の使用

コンテナは、Podman、Skopeo、および Buildah のコンテナツールで管理できる最小単位です。Podman Pod は、1つ以上のコンテナのグループです。Pod の概念は Kubernetes により導入されました。Podman Pod は Kubernetes 定義に似ています。Pod は、OpenShift または Kubernetes 環境で作成、デプロイ、管理できる最小のコンピュータ単位です。すべての Podman Pod には infra コンテナが含まれます。このコンテナは Pod に関連付けられた namespace を保持し、Podman が他のコンテナを Pod に接続できるようにします。これにより、Pod を稼働したまま、Pod 内のコンテナの起動や停止が可能になります。デフォルトの infra コンテナは、**registry.access.redhat.com/ubi9/pause** イメージ上にあります。

10.1. POD の作成

1つのコンテナで Pod を作成できます。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. 空の Pod を作成します。

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

Pod の初期状態は Created です。

2. オプション: すべての Pod をリスト表示します。

```
$ podman pod ps
POD ID      NAME      STATUS      CREATED              # OF CONTAINERS  INFRA ID
223df6b390b4  mypod    Created    Less than a second ago  1                3afdcd93de3e
```

Pod にはコンテナが1つあることに注意してください。

3. オプション: 関連付けられている全 Pod およびコンテナをリスト表示します。

```
$ podman ps -a --pod
CONTAINER ID  IMAGE              COMMAND          CREATED              STATUS  PORTS
NAMES        POD
3afdcd93de3e registry.access.redhat.com/ubi9/pause    Less than a second ago
Created      223df6b390b4-infra 223df6b390b4
```

podman ps コマンドからの Pod ID と **podman pod ps** コマンドの Pod ID が一致することが確認できます。デフォルトの infra コンテナは **registry.access.redhat.com/ubi9/pause** イメージをベースとしています。

4. **mypod** という名前の既存 Pod 内で、名前が **myubi** のコンテナを実行します。

```
$ podman run -dt --name myubi --pod mypod registry.access.redhat.com/ubi9/ubi
/bin/bash
5df5c48fea87860cf75822ceab8370548b04c78be9fc156570949013863ccf71
```

5. オプション: すべての Pod をリスト表示します。

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

Pod にはコンテナが 2 つあることがわかります。

6. オプション: 関連付けられている全 Pod およびコンテナをリスト表示します。

```
$ podman ps -a --pod
CONTAINER ID IMAGE                                COMMAND    CREATED
STATUS      PORTS NAMES          POD
5df5c48fea87 registry.access.redhat.com/ubi9/ubi:latest /bin/bash Less than a second ago
Up Less than a second ago          myubi      223df6b390b4
3afdc93de3e registry.access.redhat.com/ubi9/pause                               Less than a
second ago Up Less than a second ago          223df6b390b4-infra 223df6b390b4
```

関連情報

- **podman-pod-create** man ページ
- [Podman:Managing pods and containers in a local container runtime](#)

10.2. POD 情報の表示

Pod 情報を表示する方法について説明します。

前提条件

- **container-tools** メタパッケージがインストールされている。
- Pod が作成されている。詳細は、[Pod の作成](#) を参照してください。

手順

- Pod で実行されているアクティブなプロセスを表示します。
 - Pod 内のコンテナで実行中のプロセスを表示するには、以下を入力します。

```
$ 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
```

- 1 つ以上の Pod のコンテナにおけるリソース使用状況の統計をライブストリームで表示するには、以下を入力します。

```
$ 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 に関する情報を表示するには、以下を入力します。

```
$ 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/db99446fa9c6d10b973d1ce55a42a6850357e0cd447d9bac5627bb2516b5
b19a",
  "CreateInfra": false,
  "InfraContainerID":
"891c54f70783dcad596d888040700d93f3ead01921894bc19c10b0a03c738ff7",
  "SharedNamespaces": [
    "uts",
    "ipc",
    "net"
  ],
  "NumContainers": 2,
  "Containers": [
    {
      "Id":
"891c54f70783dcad596d888040700d93f3ead01921894bc19c10b0a03c738ff7",
      "Name": "db99446fa9c6-infra",
      "State": "running"
    },
    {
      "Id":
"effc5bbcf505b522e3bf8fbb5705a39f94a455a66fd81e542bcc27d39727d2d",
      "Name": "myubi",
      "State": "running"
    }
  ]
}
```

Pod 内のコンテナについての情報を確認できます。

関連情報

- [podman pod top man ページ](#)
- [podman-pod-stats man ページ](#)
- [podman-pod-inspect man ページ](#)

10.3. POD の停止

1つ以上の Pod を停止するには、**podman pod stop** コマンドを使用します。

前提条件

- **container-tools** メタパッケージがインストールされている。
- Pod が作成されている。詳細は、[Pod の作成](#) を参照してください。

手順

1. Pod **mypod** を停止します。

```
$ podman pod stop mypod
```

2. オプション: 関連付けられている全 Pod およびコンテナをリスト表示します。

```
$ podman ps -a --pod
CONTAINER ID IMAGE COMMAND CREATED STATUS
PORTS NAMES POD ID PODNAME
5df5c48fea87 registry.redhat.io/ubi9/ubi:latest /bin/bash About a minute ago Exited (0) 7
seconds ago myubi 223df6b390b4 mypod
3afdcd93de3e registry.access.redhat.com/9/pause About a minute ago
Exited (0) 7 seconds ago 8a4e6527ac9d-infra 223df6b390b4 mypod
```

Pod **mypod** およびコンテナ **myubi** のステータスが Exited であることが分かります。

関連情報

- **podman-pod-stop** man ページ

10.4. POD の削除

停止されている Pod またはコンテナを削除するには、**podman pod rm** コマンドを使用します。

前提条件

- **container-tools** メタパッケージがインストールされている。
- Pod が作成されている。詳細は、[Pod の作成](#) を参照してください。
- Pod が停止されている。詳細は、[Pod の停止](#) を参照してください。

手順

1. 以下を入力して、Pod **mypod** を削除します。

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

Pod を削除すると、その Pod 中の全コンテナが自動的に削除されることに注意してください。

- オプション: すべてのコンテナおよび Pod が削除されたことを確認します。

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

関連情報

- [podman-pod-rm man ページ](#)

第11章 コンテナ間の通信

ポートマッピング、DNS 解決、または Pod 内の通信のオーケストレーションを活用して、コンテナ、アプリケーション、ホストシステム間の通信を確立する方法を学習します。

11.1. ネットワークモードとレイヤー

Podman には、いくつかの異なるネットワークモードがあります。

- **bridge** - デフォルトのブリッジネットワーク上に別のネットワークを作成します。
- **container:<id>** - `id` が `<id>` のコンテナと同じネットワークを使用します。
- **host** - ホストネットワークスタックを使用します。
- **network-id** - `podman network create` コマンドで作成されたユーザー定義のネットワークを使用します。
- **private** - コンテナの新しいネットワークを作成します。
- **slirp4nets** - ルートレスコンテナのデフォルトオプションである `slirp4netns` を使用してユーザーネットワークスタックを作成します。
- **pasta** - `slirp4netns` の高性能な代替品。Podman v4.4.1 以降では **pasta** を使用できます。
- **none** - コンテナのネットワークの名前空間を作成しますが、ネットワークインターフェイスは設定しません。コンテナにはネットワーク接続がありません。
- **ns:<path>** - 参加するネットワーク名前空間へのパス



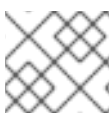
注記

ホストモードでは、D-bus (コンテナはプロセス間通信 (IPC) システム) などのローカルシステムサービスにフルアクセスできるため、安全でないと考えられています。

11.2. SLIRP4NETNS と PASTA の違い

`slirp4netns` と比較した `pasta` ネットワークモードの顕著な違いは次のとおりです。

- **pasta** は、Pv6 ポート転送をサポートしています。
- **pasta** は、**slirp4netns** よりも効率的です。
- **pasta** は、ホストから IP アドレスをコピーしますが、`slirp4netns` は定義済みの IPv4 アドレスを使用します。
- **pasta** は、ホストからのインターフェイス名を使用しますが、`slirp4netns` はインターフェイス名として `tap0` を使用します。
- **pasta** は、ホストからのゲートウェイアドレスを使用しますが、**slirp4netns** は、独自のゲートウェイアドレスを定義して NAT を使用します。



注記

ルートレスコンテナのデフォルトのネットワークモードは **slirp4netns** です。

11.3. ネットワークモードの設定

関連情報

`--network` オプションを指定した `podman run` コマンドを使用して、ネットワークモードを選択できます。

前提条件

- `container-tools` モジュールがインストールされている。

手順

1. オプション: `pasta` ネットワークモードを使用する場合は、`passt` パッケージをインストールします。

```
$ {PackageManager} install passt
```

2. `registry.access.redhat.com/ubi9/ubi` イメージに基づいてコンテナを実行します。

```
$ podman run --network=<netwok_mode> -d --name=myubi
registry.access.redhat.com/ubi9/ubi
```

`<netwok_mode>` は必要なネットワークモードです。あるいは、`containers.conf` ファイルの `default_rootless_network_cmd` オプションを使用して、デフォルトのネットワークモードを切り替えることもできます。



注記

ルートレスコンテナのデフォルトのネットワークモードは `slirp4netns` です。

検証

- ネットワークモードの設定を確認します。

```
$ podman inspect --format {{.HostConfig.NetworkMode}} myubi
<netwok_mode>
```

11.4. コンテナのネットワーク設定の検査

関連情報

`podman inspect` コマンドに `--format` オプションを付けると、`podman inspect` の出力から個々の項目を表示できます。

前提条件

- `container-tools` メタパッケージがインストールされている。

手順

1. コンテナの IP アドレスを表示します。

■

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

2. コンテナが接続されている全ネットワークを表示します。

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

3. ポートマッピングを表示します。

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

関連情報

- `podman-inspect` の man ページ

11.5. コンテナとアプリケーション間の通信

コンテナとアプリケーションの間で通信を行うことができます。アプリケーションのポートは、リスニング状態かオープン状態のどちらかです。これらのポートは自動的にコンテナネットワークに公開されるため、このようなネットワークを使用してコンテナに到達できます。デフォルトでは、Web サーバーはポート 80 でリッスンします。この手順で、**myubi** コンテナは **web-container** アプリケーションと通信を行います。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. **web-container** という名前のコンテナを起動します。

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

2. すべてのコンテナをリスト表示します。

```
# podman ps -a
```

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

3. コンテナを点検し、IP アドレスを表示します。

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

```
10.88.0.2
```

4. **myubi** コンテナを実行し、Web サーバーが動作していることを確認します。

```
# podman run -it --name=myubi ubi9/ubi curl 10.88.0.2:80
```

```
<html><body><h1>It works!</h1></body></html>
```


11.6. コンテナとホスト間の通信

デフォルトでは、**Podman** ネットワークはブリッジネットワークです。つまり、ネットワークデバイスがコンテナネットワークとホストネットワークをつなぎます。

前提条件

- **container-tools** メタパッケージがインストールされている。
- **web-container** が動作している。詳細は、[コンテナとアプリケーション間の通信](#)のセクションを参照してください。

手順

1. ブリッジが設定されていることを確認します。

```
# podman network inspect podman | grep bridge

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

2. ホストネットワークの設定を表示します。

```
# 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-container には **cni-podman0** ネットワークの IP が指定されており、そのネットワークがホストにブリッジされていることがわかります。

3. **web-container** をチェックし、その IP アドレスを表示します。

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

10.88.0.2
```

4. ホストから直接 **web-container** にアクセスします。

```
$ curl 10.88.0.2:80

<html><body><h1>It works!</h1></body></html>
```

関連情報

- **podman-network** の man ページ

11.7. ポートマッピングによるコンテナ間通信

2つのコンテナ間で通信する最も便利な方法は、公開ポートを使用することです。ポートの公開は、自動と手動の2つの方法で行うことができます。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. 未公開のコンテナを実行します。

```
# podman run -dt --name=web1 ubi9/httpd-24
```

2. 自動的に公開されたコンテナを実行します。

```
# podman run -dt --name=web2 -P ubi9/httpd-24
```

3. 手動で公開したコンテナを実行し、コンテナポート 80 を公開します。

```
# podman run -dt --name=web3 -p 9090:80 ubi9/httpd-24
```

4. すべてのコンテナをリスト表示します。

```
# podman ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS	NAMES	
f12fa79b8b39	registry.access.redhat.com/ubi9/httpd-24:latest	/usr/bin/run-http...	23 seconds ago
	Up 24 seconds ago	web1	
9024d9e815e2	registry.access.redhat.com/ubi9/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/ubi9/httpd-24:latest	/usr/bin/run-http...	2 seconds ago
	Up 2 seconds ago	0.0.0.0:9090->80/tcp	web3

以下が確認できます。

- コンテナ **web1** には公開ポートがなく、コンテナネットワークまたはブリッジにのみアクセスできます。
- コンテナ **web2** は、ポート 43595 と 42423 を自動的にマッピングして、それぞれアプリケーションポート 8080 と 8443 を公開します。



注記

registry.access.redhat.com/9/httpd-24 イメージの [Containerfile](#) に **EXPOSE 8080** と **EXPOSE 8443** コマンドがあるため、自動ポートマッピングが可能です。

- コンテナ **web3** は手動でポートを公開しています。ホストポート 9090 は、コンテナポート 80 にマッピングされます。

5. **web1**、**web3** コンテナの IP アドレスを表示します。

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

6. <IP>:<port> 表記を使用して **web1** コンテナに到達します。

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

7. localhost:<port> 表記を使用して **web2** コンテナに到達します。

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

8. <IP>:<port> 表記を使用して **web3** コンテナに到達します:

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

11.8. DNS を利用したコンテナ間の通信

DNS プラグインが有効な場合に、コンテナ名を使用してコンテナのアドレスを指定します。

前提条件

- **container-tools** メタパッケージがインストールされている。
- DNS プラグインを有効にしたネットワークが **podman network create** コマンドで作成されている。

手順

1. **myinet** ネットワークに接続された **receiver** コンテナを実行します。

```
# podman run -d --net myinet --name receiver ubi9 sleep 3000
```

2. **sender** コンテナを実行し、その名前で **receiver** コンテナにアクセスします。

```
# podman run -it --rm --net myinet --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 で終了します。

sender コンテナは、**receiver** コンテナの名前を使用して ping を送信できることがわかります。

11.9. POD 内の 2 つのコンテナ間での通信

同じ Pod 内のコンテナはすべて、IP アドレス、MAC アドレス、およびポートマッピングを共有します。同じ Pod 内のコンテナ間では、localhost:port の表記で通信が可能です。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. **web-pod** という名前の Pod を作成します。

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

2. **web-container** という名前の Web コンテナを Pod で実行します。

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

3. 関連付けられている全 Pod およびコンテナをリスト表示します。

```
$ 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 イメージを元に、**web-pod** でコンテナを実行します。

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

コンテナが **web-container** に到達できることがわかります。

11.10. POD での通信

Pod 作成時に、Pod 内のコンテナ用のポートを公開する必要があります。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. **web-pod** という名前の Pod を作成します。

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

- すべての Pod をリスト表示します。

```
# podman pod ls

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

- web-pod** で **web-container** という名前の Web コンテナを実行します。

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

- コンテナの一覧を表示します。

```
# podman ps

CONTAINER ID  IMAGE          COMMAND          CREATED          STATUS
PORTS        NAMES
7de09076d2b3  k8s.gcr.io/pause:3.5          About a minute ago  Up 23 seconds ago
0.0.0.0:80->80/tcp  26fe5de43ab3-infra
088befb90e59  docker.io/library/httpd  httpd-foreground  23 seconds ago    Up 23 seconds
ago  0.0.0.0:80->80/tcp  web-container
```

- web-container** に到達できることを確認します。

```
$ curl localhost:80

<html><body><h1>It works!</h1></body></html>
```

11.11. コンテナネットワークへの POD のアタッチ

Pod 作成時に Pod 内のコンテナをネットワークにアタッチします。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

- pod-net** という名前のネットワークを作成します。

```
# podman network create pod-net

/etc/cni/net.d/pod-net.conflist
```

- Pod **web-pod** を作成します。

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

- web-pod** で **web-container** という名前のコンテナを実行します。

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

- オプション: コンテナが関連付けられている Pod を表示します。

```
# podman ps -p
```

```
CONTAINER ID IMAGE COMMAND CREATED STATUS
PORTS NAMES POD ID PODNAME
b7d6871d018c registry.access.redhat.com/ubi9/pause:latest 9 minutes
ago Up 6 minutes ago a8e7360326ba-infra a8e7360326ba web-pod
645835585e24 docker.io/library/httpd:latest httpd-foreground 6 minutes ago Up 6 minutes
ago web-container a8e7360326ba web-pod
```

検証

- コンテナに接続されているすべてのネットワークを表示します。

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

```
pod-net
```

第12章 コンテナネットワークモードの設定

本章では、さまざまなネットワークモードを設定する方法を説明します。

12.1. 静的 IP でのコンテナ実行

podman run コマンドで **--ip** オプションを指定すると、コンテナのネットワークインターフェイスが特定の IP アドレスに設定されます (例: 10.88.0.44)。**podman inspect** コマンドを実行して、IP アドレスが正しく設定されていることを確認します。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

- コンテナのネットワークインターフェイスを IP アドレス 10.88.0.44 に設定します。

```
# podman run -d --name=myubi --ip=10.88.0.44 registry.access.redhat.com/ubi9/ubi
efde5f0a8c723f70dd5cb5dc3d5039df3b962fae65575b08662e0d5b5f9fbe85
```

検証

- IP アドレスが正しく設定されていることを確認します。

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

12.2. SYSTEMD なしでの DHCP プラグイン実行

ユーザー定義のネットワークに接続するには、**Podman run --network** コマンドを使用します。ほとんどのコンテナイメージには DHCP クライアントがありませんが、**dhcp** プラグインは、コンテナが DHCP サーバーを操作するためのプロキシ DHCP クライアントとして機能します。



注記

この手順は、ルートフルコンテナにのみ適用されます。ルートレスコンテナでは、**dhcp** プラグインは使用されません。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. **dhcp** プラグインを手動で実行します。

```
# /usr/libexec/cni/dhcp daemon &
[1] 4966
```

2. **dhcp** プラグインが動作していることを確認します。

```
# ps -a | grep dhcp
4966 pts/1 00:00:00 dhcp
```

3. **alpine** コンテナを実行します。

```
# 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
    ...
```

この例では、以下が適用されます。

- **network=example** オプションは、接続するネットワークを **example** という名前で指定します。
- **alpine** コンテナで **ip addr show enp1s0** コマンドを実行すると、ネットワークインターフェイス **enp1s0** の IP アドレスを確認します。
- ホストネットワークは 192.168.1.0/24 です。
- **eth0** インターフェイスは、**alpine** コンテナ用に 192.168.1.122 の IP アドレスをリースしています。



注記

この設定では、有効期限の短いコンテナとリースの長い DHCP サーバーが多数ある場合に、利用可能な DHCP アドレスが枯渇する可能性があります。

関連情報

- [Leasing routable IP addresses with Podman containers](#)

12.3. SYSTEMD を使用した DHCP プラグインの実行

systemd ユニットファイルを使用して **dhcp** プラグインを実行できます。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. ソケットユニットファイルを作成します。

```
# 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. サービスユニットファイルを作成します。

```
# 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. サービスをすぐに起動します。

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

検証

- ソケットのステータスを確認します。

```
# 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
```

関連情報

- [Leasing routable IP addresses with Podman containers](#)

12.4. MACVLAN プラグイン

ほとんどのコンテナイメージには DHCP クライアントがありませんが、**dhcp** プラグインは、コンテナが DHCP サーバーを操作するためのプロキシ DHCP クライアントとして機能します。

ホストシステムには、コンテナへのネットワークアクセス権がありません。ホスト外部からコンテナへのネットワーク接続を許可するには、コンテナには、ホストと同じネットワーク上に IP が必

要です。**macvlan** プラグインを使用すると、コンテナをホストと同じネットワークに接続することができます。



注記

この手順は、ルートフルコンテナにのみ適用されます。ルートレスコンテナは、**macvlan** および **dhcp** プラグインを使用することができません。



注記

macvlan ネットワークは、**podman network create --macvlan** コマンドで作成することができます。

関連情報

- [Leasing routable IP addresses with Podman containers](#)
- **podman-network-create** man ページ

12.5. ネットワークスタックの CNI から NETAVARK への切り替え

これまで、コンテナは単一の Container Network Interface (CNI) プラグインに接続されている場合のみ DNS を使用できました。Netavark は、コンテナ用のネットワークスタックです。Netavark は、Podman やその他の OCI (Open Container Initiative) コンテナ管理アプリケーションで使用できます。Podman の高度なネットワークスタックは、Docker の詳細機能にも対応しています。現在、複数のネットワーク上にあるコンテナは、これらのネットワークにあるコンテナにアクセスします。

Netavark は以下のことが可能です。

- ブリッジおよび MACVLAN インターフェイスなどのネットワークインターフェイスの作成、管理、削除。
- ネットワークアドレス変換 (NAT) やポートマッピングルールなどのファイアウォールの設定。
- IPv4、IPv6 のサポート。
- 複数のネットワークにおけるコンテナへの対応改善。



警告

CNI ネットワークスタックは非推奨であり、Podman v5.0 では削除されます。代わりに Netavark ネットワークスタックを使用してください。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. `/etc/containers/containers.conf` ファイルが存在しない場合は、`/usr/share/containers/containers.conf` ファイルを `/etc/containers/` ディレクトリーにコピーします。

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

2. `/etc/containers/containers.conf` ファイルを編集し、`[network]` セクションに以下の内容を追加します。

```
network_backend="netavark"
```

3. コンテナや Pod がある場合は、ストレージをリセットして初期状態に戻します。

```
# podman system reset
```

4. システムを再起動します。

```
# reboot
```

検証

- ネットワークスタックが Netavark に変更されていることを確認します。

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



注記

Podman 4.0.0 以降をお使いの場合は、`podman info` コマンドでネットワークスタックの設定を確認してください。

関連情報

- [Podman 4.0's new network stack:What you need to know](#)
- `podman-system-reset` の man ページ

12.6. NETAVARK から CNI へのネットワークスタックの切り替え

ネットワークスタックを Netavark から CNI に切り替えることができます。



警告

CNI ネットワークスタックは非推奨であり、Podman v5.0 では削除されます。代わりに Netavark ネットワークスタックを使用してください。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. `/etc/containers/containers.conf` ファイルが存在しない場合は、`/usr/share/containers/containers.conf` ファイルを `/etc/containers/` ディレクトリーにコピーします。

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

2. `/etc/containers/containers.conf` ファイルを編集し、**[network]** セクションに以下の内容を追加します。

```
network_backend="cni"
```

3. コンテナや Pod がある場合は、ストレージをリセットして初期状態に戻します。

```
# podman system reset
```

4. システムを再起動します。

```
# reboot
```

検証

- ネットワークスタックが CNI に変更されていることを確認します。

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



注記

Podman 4.0.0 以降をお使いの場合は、**podman info** コマンドでネットワークスタックの設定を確認してください。

関連情報

- [Podman 4.0's new network stack:What you need to know](#)
- **podman-system-reset** の man ページ

第13章 PODMAN を使用した OPENSIFT へのコンテナの移植

YAML (YAML Ain't Markup Language) 形式を使用して、コンテナおよび Pod の移植可能な記述を生成できます。YAML は、設定データの記述に使用されるテキスト形式です。

YAML ファイルは以下のとおりです。

- 読み取り可能
- 生成が簡単
- 環境間の移植性 (RHEL と OpenShift の間など)
- プログラミング言語間で移植が可能
- 使いやすい (全パラメーターをコマンドラインに追加する必要はない)。

YAML ファイルを使用する理由:

1. 必要最小限の入力でオーケストレーションされたコンテナおよび Pod のローカルオーケストレーションセットを再実行でき、反復型開発に役立ちます。
2. 同じコンテナおよび Pod を別のマシンで実行できます。たとえば、OpenShift 環境でアプリケーションを実行し、アプリケーションが正常に機能していることを確認します。**podman generate kube** コマンドを使用して、Kubernetes YAML ファイルを生成できます。次に、**podman play** コマンドを使用して、生成された YAML ファイルを Kubernetes または OpenShift 環境に転送する前に、ローカルシステムで Pod およびコンテナの作成をテストできます。**podman play** コマンドを使用して、OpenShift または Kubernetes 環境で作成された Pod およびコンテナを再作成することもできます。



注記

podman kube play コマンドは、Kubernetes YAML 機能のサブセットをサポートしません。詳細は、[サポートされる YAML フィールドのサポートマトリックス](#) を参照してください。

13.1. PODMAN を使用した KUBERNETES YAML ファイルの生成

1つのコンテナで Pod を作成し、**podman generate kube** コマンドを使用して Kubernetes YAML ファイルを生成できます。

前提条件

- **container-tools** メタパッケージがインストールされている。
- Pod が作成されている。詳細は、[Pod の作成](#) を参照してください。

手順

1. 関連付けられている全 Pod およびコンテナをリスト表示します。

```
$ podman ps -a --pod
CONTAINER ID IMAGE COMMAND CREATED
STATUS PORTS NAMES POD
5df5c48fea87 registry.access.redhat.com/ubi9/ubi:latest /bin/bash Less than a second ago
```

```
Up Less than a second ago    myubi    223df6b390b4
3afdcd93de3e k8s.gcr.io/pause:3.1      Less than a second ago Up Less
than a second ago    223df6b390b4-infra 223df6b390b4
```

- Pod 名または ID を使用して Kubernetes YAML ファイルを生成します。

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

podman generate コマンドは、コンテナに接続されている論理ボリュームマネージャー (LVM) の論理ボリュームまたは物理ボリュームへは反映されないので注意してください。

- mypod.yaml** ファイルを表示します。

```
$ 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/ubi9/ubi:latest
    name: myubi
    resources: {}
    securityContext:
      allowPrivilegeEscalation: true
      capabilities: {}
      privileged: false
      readOnlyRootFilesystem: false
    tty: true
    workingDir: /
  status: {}
```

関連情報

- **podman-generate-kube** の man ページ

- [Podman:Managing pods and containers in a local container runtime](#)

13.2. OPENSIFT 環境での KUBERNETES YAML ファイルの生成

OpenShift 環境では、**oc create** コマンドを使用して、アプリケーションを記述する YAML ファイルを生成します。

手順

- **myapp** アプリケーションの YAML ファイルを生成します。

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

oc create コマンドは **myapp** イメージを作成して実行します。オブジェクトは **--dry-run** オプションを使用して出力され、**myapp.yaml** 出力ファイルにリダイレクトされます。



注記

Kubernetes 環境では、同じフラグを指定して **kubectl create** コマンドを使用できます。

13.3. PODMAN でのコンテナおよび POD の起動

生成された YAML ファイルを使用すると、任意の環境でコンテナおよび Pod を自動的に起動できます。YAML ファイルは、Kubernetes や Openshift など、Podman 以外のツールを使用して生成できます。**podman play kube** コマンドを使用すると、YAML 入力ファイルに基づいて Pod およびコンテナを再作成できます。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. **mypod.yaml** ファイルから Pod およびコンテナを作成します。

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

2. すべての Pod をリスト表示します。

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

3. 関連付けられている全 Pod およびコンテナをリスト表示します。

```
$ podman ps -a --pod
CONTAINER ID  IMAGE                                     COMMAND          CREATED          STATUS
PORTS        NAMES          POD
848179395ebd  registry.access.redhat.com/ubi9/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 コマンドからの Pod ID と、 **podman pod ps** コマンドからの Pod ID が一致します。

関連情報

- **podman-play-kube** の man ページ
- [Podman can now ease the transition to Kubernetes and CRI-O](#)

13.4. OPENSIFT 環境でのコンテナおよび POD の起動

oc create コマンドを使用して、OpenShift 環境で Pod およびコンテナを作成できます。

手順

- OpenShift 環境で YAML ファイルから Pod を作成します。

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



注記

Kubernetes 環境では、同じフラグを指定して **kubectl create** コマンドを使用できます。

13.5. PODMAN を使用したコンテナと POD の手動実行

以下の手順は、Podman を使用して MariaDB データベースと対になる WordPress コンテンツマネジメントシステムを手動で作成する方法を説明します。

次のようなディレクトリレイアウトがあるとします。

```
|
|--- mariadb-conf
|   |--- Containerfile
|   |--- my.cnf
```

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. **mariadb-conf/Containerfile** ファイルを表示します。

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

2. **mariadb-conf/my.cnf** ファイルを表示します。


```
[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** を使用して **docker.io/library/mariadb** イメージをビルドします。

```
$ 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. オプション: すべてのイメージをリスト表示します。

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

5. **wordpresspod** という名前の pod を作成し、コンテナとホストシステム間のポートマッピングを設定します。

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

6. **wordpresspod** pod の中に **mydb** コンテナを作成します。

```
$ 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** pod の中に **myweb** コンテナを作成します。

```
$ 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. オプション:関連付けられている全 Pod およびコンテナをリスト表示します。

```
$ 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
60e8dbbabc5 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
```

検証

- Pod が実行されていることを確認します。http://localhost:8080/wp-admin/install.php のページにアクセスするか、**curl** コマンドを使用してください。

```
$ curl http://localhost:8080/wp-admin/install.php
<!DOCTYPE html>
<html xml:lang="en-US">
<head>
...
</head>
<body class="wp-core-ui">
<p id="logo">WordPress</p>
<h1>Welcome</h1>
...
```

関連情報

- [Build Kubernetes pods with Podman play kube](#)
- podman-play-kube** の man ページ

13.6. PODMAN を使用した YAML ファイルの生成

Kubernetes の YAML ファイルは、**podman generate kube** コマンドで生成できます。

前提条件

- container-tools** メタパッケージがインストールされている。

- **wordpresspod** という名前の Pod が作成されている。詳細は、[Pod の作成](#) を参照してください。

手順

1. 関連付けられている全 Pod およびコンテナをリスト表示します。

```
$ 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 名または ID を使用して Kubernetes YAML ファイルを生成します。

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

検証

- **wordpresspod.yaml** ファイルを表示します。

```
$ 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: mywpuuser
    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

```

関連情報

- [Build Kubernetes pods with Podman play kube](#)
- `podman-play-kube` の man ページ

13.7. PODMAN を使用したコンテナと POD の自動実行

次に、`podman play kube` コマンドを使用して、生成された YAML ファイルを Kubernetes または OpenShift 環境に転送する前に、ローカルシステムで Pod およびコンテナの作成をテストできます。

`podman play kube` コマンドは、`docker compose` コマンドと同様に YAML ファイルを使用して、コンテナが複数ある Pod を自動的に構築して実行することも可能です。以下の条件が該当する場合には、自動的にイメージが構築されます。

1. YAML ファイルで使用されているイメージと同じ名前のディレクトリが存在する
2. そのディレクトリには Containerfile が含まれている

前提条件

- `container-tools` メタパッケージがインストールされている。
- `wordpresspod` という名前の Pod が作成されている。詳細は、[Podman を使用したコンテナと Pod の手動実行](#) を参照してください。
- YAML ファイルが生成されている。詳細は、[Podman を使用した YAML ファイルの生成のセクション](#)をご覧ください。
- 最初からやり直す場合は、ローカルに保存されているイメージを削除してください。

```

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

```

手順

1. `wordpress.yaml` ファイルを使用して、wordpress Pod を作成します。

```

$ podman play kube wordpress.yaml
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 コマンド:

- **docker.io/library/mariadb** イメージを元に **localhost/mariadb-conf:latest** イメージを自動構築します。
- **docker.io/library/wordpress:latest** のイメージをプルします。
- **wordpresspod-mydb** と **wordpresspod-myweb** の2つのコンテナが含まれる、**wordpresspod** という名前の Pod を作成します。

2. すべてのコンテナと Pod を表示します。

```

$ 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-mydb 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

```

検証

- Pod が実行されていることを確認します。 <http://localhost:8080/wp-admin/install.php> のページにアクセスするか、**curl** コマンドを使用してください。

```

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

```

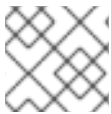
```
<p id="logo">WordPress</p>
<h1>Welcome</h1>
...
```

関連情報

- [Build Kubernetes pods with Podman play kube](#)
- `podman-play-kube` の man ページ

13.8. PODMAN を使用した POD の自動停止/削除

`podman play kube --down` コマンドは、すべての Pod とそのコンテナを停止して削除します。



注記

ボリュームが使用中の場合には削除されません。

前提条件

- **container-tools** メタパッケージがインストールされている。
- **wordpresspod** という名前の Pod が作成されている。詳細は、[Podman を使用したコンテナと Pod の手動実行](#) を参照してください。
- YAML ファイルが生成されている。詳細は、[Podman を使用した YAML ファイルの生成のセクション](#)をご覧ください。
- Pod が実行中である。詳細は、[Podman を使用したコンテナや Pod の自動実行](#) を参照してください。

手順

- **wordpresspod.yaml** ファイルで作成された全 Pod とコンテナを削除します。

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

検証

- **wordpresspod.yaml** ファイルで作成された全 Pod とコンテナが削除されたことを確認します。

```
$ podman ps --pod -a
CONTAINER ID IMAGE COMMAND CREATED
STATUS PORTS NAMES POD ID PODNAME
```

関連情報

- [Build Kubernetes pods with Podman play kube](#)

- **podman-play-kube** の man ページ

第14章 PODMAN を使用した SYSTEMD へのコンテナの移植

Podman (Pod Manager) は、シンプルなデーモンレスツールである、フル機能のコンテナエンジンです。Podman は、他のコンテナエンジンからの移行を容易にし、Pod、コンテナ、およびイメージの管理を可能にする Docker-CLI と同等のコマンドラインを備えています。

もともと、Podman は、Linux システム全体を提供したり、起動順序、依存関係のチェック、失敗したサービスの回復などのサービスを管理したりするように設計されたものではありませんでした。完全なシステム初期化は、**systemd** が担当していました。Red Hat はコンテナを **systemd** と統合しているため、Linux システムで他のサービスや機能が管理するのと同じ方法で、Podman によってビルドされた OCI および Docker 形式のコンテナを管理できます。**systemd** 初期化サービスを使用して、Pod とコンテナを操作できます。

systemd ユニットファイルを使用すると、次のことが可能になります。

- **systemd** サービスとして開始するようにコンテナまたは Pod をセットアップします。
- コンテナ化されたサービスを実行して依存関係を確認する順序を定義します (たとえば、別のサービスが実行されていること、ファイルが利用可能であること、またはリソースがマウントされていることなど)。
- **systemctl** コマンドを使用して、**systemd** システムの状態を制御します。

systemd ユニットファイルを使用して、コンテナと Pod の移植可能な説明を生成できます。

14.1. QUADLET を使用した SYSTEMD ユニットファイルの自動生成

Quadlet を使用する場合、通常の **systemd** ユニットファイルによく似た形式でコンテナを実行する方法を記述します。コンテナの記述は、関連するコンテナの詳細を中心に行うため、**systemd** でのコンテナ実行に関する技術的詳細を意識せずに済みます。次のいずれかのディレクトリーに **<CTRNAME>.container** ユニットファイルを作成します。

- root ユーザーの場合: **/usr/share/containers/systemd/** または **/etc/containers/systemd/**
- ルートレスユーザーの場合:
\$HOME/.config/containers/systemd/、**\$XDG_CONFIG_HOME/containers/systemd/**、**/etc/containers/systemd/users/\$(UID)**、または **/etc/containers/systemd/users/**



注記

Quadlet は、Podman v4.6 以降で利用可能です。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. **mysleep.container** ユニットファイルを作成します。

```
$ cat $HOME/.config/containers/systemd/mysleep.container
[Unit]
Description=The sleep container
After=local-fs.target
```



```
[Container]
Image=registry.access.redhat.com/ubi9-minimal:latest
Exec=sleep 1000
```

```
[Install]
# Start by default on boot
WantedBy=multi-user.target default.target
```

[Container] セクションでは、以下を指定する必要があります。

- **Image** - 実行するコンテナイメージ
- **Exec** - コンテナ内で実行するコマンド
これにより、**systemd** ユニットファイルで指定する他のすべてのフィールドを使用できるようになります。

2. **mysleep.container** ファイルに基づいて **mysleep.service** を作成します。

```
$ systemctl --user daemon-reload
```

3. オプション: **mysleep.service** のステータスを確認します。

```
$ systemctl --user status mysleep.service
○ mysleep.service - The sleep container
  Loaded: loaded (/home/username/.config/containers/systemd/mysleep.container;
         generated)
  Active: inactive (dead)
```

4. **mysleep.service** を起動します。

```
$ systemctl --user start mysleep.service
```

検証

1. **mysleep.service** のステータスを確認します。

```
$ systemctl --user status mysleep.service
● mysleep.service - The sleep container
  Loaded: loaded (/home/username/.config/containers/systemd/mysleep.container;
         generated)
  Active: active (running) since Thu 2023-02-09 18:07:23 EST; 2s ago
    Main PID: 265651 (common)
      Tasks: 3 (limit: 76815)
     Memory: 1.6M
        CPU: 94ms
       CGroup: ...
```

2. すべてのコンテナをリスト表示します。

```
$ podman ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS
PORTS NAMES
421c8293fc1b registry.access.redhat.com/ubi9-minimal:latest sleep 1000 30
seconds ago Up 10 seconds ago systemd-mysleep
```

作成されたコンテナの名前は次の要素で構成されています。

- **systemd-** 接頭辞
- **systemd** ユニットの名前、つまり **systemd-mysleep**
この名前は、一般的なコンテナと **systemd** ユニットで実行されているコンテナを区別するのに役立ちます。また、コンテナがどのユニットで実行されるかを判断するのにも役立ちます。コンテナの名前を変更する場合は、**[Container]** セクションの **ContainerName** フィールドを使用します。

関連情報

- [Make systemd better for Podman with Quadlet](#)
- [Quadlet アップストリームのドキュメント](#)

14.2. SYSTEMD サービスの有効化

サービスの有効化には、さまざまなオプションがあります。

手順

- サービスの有効化:
 - システムの起動時にサービスを有効にするには、ユーザーがログインしているかどうかにかかわらず、次のコマンドを実行します。

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

systemd ユニットファイルを **/etc/systemd/system** ディレクトリーにコピーする必要があります。

- ユーザーのログイン時にサービスを起動し、ユーザーのログアウト時に停止するには、次のコマンドを実行します。

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

systemd ユニットファイルを **\$HOME/.config/systemd/user** ディレクトリーにコピーする必要があります。

- システムの起動時にサービスを起動し、ログアウト後もそのまま起動した状態を保つには、次のコマンドを実行します。

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

関連情報

- **systemctl** の man ページ
- **loginctl** の man ページ
- [ブート時のシステムサービス起動の有効化](#)

14.3. SYSTEMD を使用したコンテナの自動起動

systemctl コマンドを使用して、**systemd** システムおよびサービスマネージャーの状態を制御できます。root 以外のユーザーでサービスを有効化、起動、停止できます。root ユーザーとしてサービスをインストールするには、**--user** オプションを省略します。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. **systemd** マネージャー設定をリロードします。

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

2. サービス **container.service** を有効にし、起動時に開始します。

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

3. サービスをすぐに起動します。

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

4. サービスの状況を表示するには、次のコマンドを実行します。

```
$ 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 --conmon-pidfile
//run/user/1000/container.service-pid --cidfile //run/user/1000/container.service-cid -d ubi9-
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 (conmon)
    CGroup: /user.slice/user-1000.slice/user@1000.service/container.service
            └─ 2870 /usr/bin/podman
               └─ 80612 /usr/bin/slipr4netns --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/YJSPGXM2OCDZPLMLXJOW3N
RF6Q:/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 コマンドを使用して、サービスが有効であるかどうかを確認できます。

検証手順

- 実行中または終了したコンテナをリスト表示します。

```
# podman ps
CONTAINER ID IMAGE COMMAND CREATED STATUS
PORTS NAMES
f20988d59920 registry.access.redhat.com/ubi9-minimal:latest top 12 seconds ago Up 11
seconds ago funny_zhukovsky
```



注記

`container.service` を停止するには、以下を入力します。

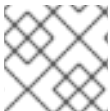
```
# systemctl --user stop container.service
```

関連情報

- [systemctl の man ページ](#)
- [Running containers with Podman and shareable systemd services](#)
- [ブート時のシステムサービス起動の有効化](#)

14.4. PODMAN GENERATED SYSTEMD コマンドではなく QUADLET を使用する利点

通常の `systemd` ユニットファイルと同様の形式でコンテナを実行する方法を記述する Quadlets ツールを使用できます。



注記

Quadlet は、Podman v4.6 以降で利用可能です。

Quadlet には、`podman generated systemd` コマンドを使用してユニットファイルを生成する場合に比べて、次のような多くの利点があります。

- **メンテナンスが簡単:** コンテナの記述は、関連するコンテナの詳細を中心に行うため、`systemd` でのコンテナ実行に関する技術的詳細を意識せずに済みます。
- **自動更新:** Quadlet では、更新後にユニットファイルを手動で再生成する必要はありません。Podman の新しいバージョンがリリースされた場合、起動時などに `systemctl daemon-reload` コマンドを実行すると、サービスが自動的に更新されます。
- **シンプルなワークフロー:** 簡素化された構文のおかげで、Quadlet ファイルをゼロから作成して、どこにでもデプロイできます。
- **標準の systemd オプションのサポート:** Quadlet は、既存の `systemd-unit` 構文を新しいテーブル (コンテナを設定するテーブルなど) で拡張します。



注記

Quadlet は Kubernetes YAML 機能のサブセットをサポートします。詳細は、[サポートされる YAML フィールドのサポートマトリックス](#) を参照してください。次のツールのいずれかを使用して YAML ファイルを生成できます。

- Podman: **podman generate kube** コマンド
- OpenShift: **oc generated** コマンドと **--dry-run** オプション
- Kubernetes: **kubectl create** コマンドと **--dry-run** オプション

Quadlet は次のユニットファイルタイプをサポートします。

- **コンテナユニット**: **podman run** コマンドを実行してコンテナを管理するために使用します。
 - ファイル拡張子: **.container**
 - セクション名: **[Container]**
 - 必須フィールド: サービスによって実行されるコンテナイメージを記述する **Image**
- **Kube ユニット**: **podman kube play** コマンドを実行して、Kubernetes YAML ファイルで定義したコンテナを管理するために使用します。
 - ファイル拡張子: **.kube**
 - セクション名: **[Kube]**
 - 必須フィールド: Kubernetes YAML ファイルへのパスを定義する **Yaml**
- **ネットワークユニット**: **.container** または **.kube** ファイルで参照できる Podman ネットワークを作成するために使用します。
 - ファイル拡張子: **.network**
 - セクション名: **[Network]**
 - 必須フィールド: なし
- **ボリュームユニット**: **.container** ファイルで参照できる Podman ボリュームを作成するために使用します。
 - ファイル拡張子: **.volume**
 - セクション名: **[Volume]**
 - 必須フィールド: なし

関連情報

- [Quadlet アップストリームのドキュメント](#)

14.5. PODMAN を使用した SYSTEMD ユニットファイルの生成

Podman により、**systemd** はコンテナプロセスを制御および管理できます。**podman generate**

systemd コマンドを使用して、既存のコンテナと Pod の **systemd** ユニットファイルを生成できます。生成されたユニットファイルは頻繁に変更され (Podman に対する更新)、**podman generate systemd** で最新のユニットファイルの取得を確認できるので、**podman generate systemd** の使用を推奨します。



注記

Podman v4.6 以降では、Quadlet を使用できるようになりました。Quadlet を使用すると、通常の **systemd** ユニットファイルと同様の形式でコンテナを実行する方法を記述でき、**systemd** でコンテナを実行する際の複雑さを意識せずに済みます。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. コンテナ (例: **myubi**) を作成します。

```
$ podman create --name myubi registry.access.redhat.com/ubi9:latest sleep infinity
0280afe98bb75a5c5e713b28de4b7c5cb49f156f1cce4a208f13fee2f75cb453
```

2. コンテナ名または ID を使用して **systemd** ユニットファイルを生成し、それを `~/.config/systemd/user/container-myubi.service` ファイルに送ります。

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

検証手順

- 生成された **systemd** ユニットファイルの内容を表示します。

```
$ 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/user
data/conmon.pid
```

```
Type=forking
```

```
[Install]
```

```
WantedBy=multi-user.target default.target
```

- **Restart=on-failure** 行は再起動ポリシーを設定し、サービスを正常に開始または停止できない場合、またはプロセスがゼロ以外で終了した場合に再起動するように **systemd** に指示します。
- **ExecStart** 行は、コンテナの開始方法を示しています。
- **ExecStop** 行は、コンテナを停止して削除する方法を示しています。

関連情報

- [Running containers with Podman and shareable systemd services](#)

14.6. PODMAN を使用した SYSTEMD ユニットファイルの自動生成

デフォルトで、Podman は既存のコンテナまたは Pod のユニットファイルを生成します。**podman generate systemd --new** を使用すると、より移植性の高い **systemd** ユニットファイルを生成できます。**--new** フラグでは、コンテナの作成、起動、削除を行うユニットファイルを生成するように Podman に指示します。



注記

Podman v4.6 以降では、Quadlet を使用できるようになりました。Quadlet を使用すると、通常の **systemd** ユニットファイルと同様の形式でコンテナを実行する方法を記述でき、**systemd** でコンテナを実行する際の複雑さを意識せずに済みます。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. システムで使用するイメージをプルします。たとえば、**httpd-24** イメージをプルするには、以下を実行します。

```
# podman pull registry.access.redhat.com/ubi9/httpd-24
```

2. オプション: システムで利用可能なイメージのリストを表示します。

```
# podman images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
registry.access.redhat.com/ubi9/httpd-24	latest	8594be0a0b57	2 weeks ago	462 MB

3. **httpd** コンテナを作成します。

```
# podman create --name httpd -p 8080:8080 registry.access.redhat.com/ubi9/httpd-24 cdb9f981cf143021b1679599d860026b13a77187f75e46cc0eac85293710a4b1
```

4. オプション: コンテナが作成されたことを確認します。

```
# podman ps -a
CONTAINER ID IMAGE COMMAND CREATED
STATUS PORTS NAMES
cdb9f981cf14 registry.access.redhat.com/ubi9/httpd-24:latest /usr/bin/run-http... 5 minutes
ago Created 0.0.0.0:8080->8080/tcp httpd
```

5. **httpd** コンテナの **systemd** ユニットファイルを生成します。

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

6. 生成された **container-httpd.service systemd** ユニットファイルの内容を表示します。

```
# 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/ubi9/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
```

注記

--new オプションを使用してユニットファイルを生成した場合には、コンテナと Pod の存在は想定されていません。したがって、サービスの起動時に (**ExecStart** の行を参照)、**podman start** コマンドではなく、**podman run** コマンドを実行します。たとえば、[Generating a systemd unit file using Podman](#) のセクションを参照してください。

- **podman run** コマンドは、以下のコマンドラインオプションを使用します。
 - **--common-pidfile** オプションは、ホストで実行している **common** プロセスのプロセス ID を格納するパスを指します。**common** プロセスはコンテナと同じ終了ステータスで終了

します。これにより、**systemd** は正しいサービスステータスを報告し、必要に応じてコンテナを再起動できます。

- **--cidfile** オプションは、コンテナ ID を格納するパスを指します。
 - **%t** は、ランタイムディレクトリーのルートへのパスです (例: `/run/user/$UserID`)。
 - **%n** は、サービスの完全な名前です。
1. `/etc/systemd/system` にユニットファイルをコピーして root ユーザーとしてインストールします。

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

2. **container-httpd.service** を有効にして起動します。

```
# 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.
```

検証手順

- **container-httpd.service** のステータスを確認します。

```
# 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 --conmon-pidfile /run/container-
httpd.pid --cidfile /run/container-httpd.ctr-id --cgrouops=no-conmon -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 (conmon)
   ...
```

関連情報

- [Systemd Integration with Podman 2.0](#)
- [ブート時のシステムサービス起動の有効化](#)

14.7. SYSTEMD を使用した POD の自動起動

複数のコンテナを **systemd** サービスとして起動できます。**systemctl** コマンドは、Pod でだけ使用するようにしてください。コンテナは Pod サービスと内部の `infra-container` で管理されているので **systemctl** を使用して個別にコンテナを開始または停止しないでください。



注記

Podman v4.6 以降では、Quadlet を使用できるようになりました。Quadlet を使用すると、通常の **systemd** ユニットファイルと同様の形式でコンテナを実行する方法を記述でき、**systemd** でコンテナを実行する際の複雑さを意識せずに済みます。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. たとえば、**systemd-pod** などの空の Pod を作成します。

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

2. オプション: すべての Pod をリスト表示します。

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

3. 空の Pod に 2 つのコンテナを作成します。たとえば、**container0** と **container1** を **systemd-pod** に作成するには、以下を実行します。

```
$ podman create --pod systemd-pod --name container0
registry.access.redhat.com/ubi9 top
$ podman create --pod systemd-pod --name container1
registry.access.redhat.com/ubi9 top
```

4. オプション: 関連付けられている全 Pod およびコンテナをリスト表示します。

```
$ podman ps -a --pod
CONTAINER ID  IMAGE                                COMMAND  CREATED      STATUS
PORTS        NAMES                                POD ID   PODNAME
24666f47d9b2  registry.access.redhat.com/ubi9:latest  top     3 minutes ago  Created
container0    3130f724e229  systemd-pod
56eb1bf0cdfc  k8s.gcr.io/pause:3.2                    4 minutes ago  Created
3130f724e229-infra  3130f724e229  systemd-pod
62118d170e43  registry.access.redhat.com/ubi9:latest  top     3 seconds ago  Created
container1    3130f724e229  systemd-pod
```

5. 新しい Pod の **systemd** ユニットファイルを生成します。

```
$ podman generate systemd --files --name systemd-pod
/home/user1/pod-systemd-pod.service
/home/user1/container-container0.service
/home/user1/container-container1.service
```

3 つの **systemd** ユニットファイルが生成されることに注意してください。1 つは **systemd-pod** Pod 用、2 つはコンテナ **container0** および **container1** 用です。

6. **pod-systemd-pod.service** ユニットファイルを表示します。

```

$ 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/1dfd20e35043939ea3f80f002c65c00d560e47223685dbc3230e26fe001b29/userda
ta/conmon.pid
Type=forking

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

```

- **[Unit]** セクションの **Requires** 行は、**container-container0.service** と **container-container1.service** ユニットファイルの依存関係を定義します。両方のユニットファイルがアクティベートされます。
- **[Service]** セクションの **ExecStart** 行および **ExecStop** 行はそれぞれ infra-container を開始して停止します。

7. **container-container0.service** ユニットファイルを表示します。

```

$ 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/userdat
a/common.pid
Type=forking
```

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

- **[Unit]** セクションの **Bindsto** | 行は、 **pod-systemd-pod.service** ユニットファイルの依存関係を定義します。
- **[Service]** セクションの **ExecStart** 行および **ExecStop** 行では、それぞれ **container0** を起動および停止します。

8. **container-container1.service** ユニットファイルを表示します。

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

9. 生成されたファイルをすべて **\$HOME/.config/systemd/user** にコピーして、root 以外のユーザーとしてインストールします。

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

10. ユーザーのログイン時に、サービスを有効にして開始します。

```
$ 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.
```

サービスは、ユーザーのログアウト時に停止される点に注意してください。

検証手順

- サービスが有効になっているかどうかを確認します。

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

関連情報

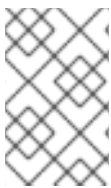
- **podman-create** の man ページ
- **podman-generate-systemd** の man ページ
- **systemctl** の man ページ

- [Running containers with Podman and shareable systemd services](#)
- ブート時のシステムサービス起動の有効化

14.8. PODMAN を使用したコンテナの自動更新

podman auto-update コマンドを使用すると、自動更新のポリシーに従ってコンテナを自動的に更新できます。**podman auto-update** コマンドは、コンテナイメージがレジストリーで更新されるとサービスを更新します。自動更新を使用するには、コンテナを **--label "io.containers.autoupdate=image"** ラベルで作成し、**podman generate systemd --new** コマンドによって生成された **systemd** ユニットで実行する必要があります。

Podman は、**"io.containers.autoupdate"** ラベルが **"image"** に設定されている実行中のコンテナを検索して、コンテナのレジストリーと通信します。イメージが変更された場合、Podman は対応する **systemd** ユニットを再起動して古いコンテナを停止し、新しいイメージで新しいコンテナを作成します。そのため、コンテナ、その環境、およびすべての依存関係が再起動されます。



注記

Podman v4.6 以降では、Quadlet を使用できるようになりました。Quadlet を使用すると、通常の **systemd** ユニットファイルと同様の形式でコンテナを実行する方法を記述でき、**systemd** でコンテナを実行する際の複雑さを意識せずに済みます。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. **registry.access.redhat.com/ubi9/ubi-init** イメージをもとに、**myubi** コンテナを起動します。

```
# podman run --label "io.containers.autoupdate=image" \
--name myubi -dt registry.access.redhat.com/ubi9/ubi-init top
bc219740a210455fa27deacc96d50a9e20516492f1417507c13ce1533dbdcd9d
```

2. オプション: 実行中または終了したコンテナをリスト表示します。

```
# podman ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS
PORTS NAMES
76465a5e2933 registry.access.redhat.com/9/ubi-init:latest top 24 seconds ago Up 23
seconds ago myubi
```

3. **myubi** コンテナの **systemd** ユニットファイルを生成します。

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

4. **/usr/lib/systemd/system** にユニットファイルをコピーして root ユーザーとしてインストールします。

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

5. **systemd** マネージャー設定をリロードします。

```
# systemctl daemon-reload
```

6. コンテナを起動して、ステータスを確認します。

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

7. コンテナを自動更新します。

```
# podman auto-update
```

関連情報

- [Systemd Integration with Podman 2.0](#)
- [Running containers with Podman and shareable systemd services](#)
- [ブート時のシステムサービス起動の有効化](#)

14.9. SYSTEMD を使用したコンテナの自動更新

前述のように、[Podman を使用したコンテナの自動更新](#) のセクションに記載のとおり、

podman auto-update コマンドを使用してコンテナを更新できます。カスタムスクリプトに組み込み、必要に応じて呼び出すことができます。コンテナを自動更新するもう1つの方法は、プリインストールされている **podman-auto-update.timer** および **podman-auto-update.service** **systemd** サービスを使用することです。**podman-auto-update.timer** は、特定の日付と時刻で自動更新をトリガーするように設定できます。**podman-auto-update.service** は、**systemctl** コマンドによってさらに開始することも、他の **systemd** サービスによる依存関係として使用することもできます。その結果、時間およびイベントに基づく自動更新は、個々のニーズやユースケースを満たすためにさまざまな方法でトリガーできます。



注記

Podman v4.6 以降では、Quadlet を使用できるようになりました。Quadlet を使用すると、通常の **systemd** ユニットファイルと同様の形式でコンテナを実行する方法を記述でき、**systemd** でコンテナを実行する際の複雑さを意識せずに済みます。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. **podman-auto-update.service** ユニットファイルを表示します。

```
# 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** ユニットファイルを表示します。

```

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

[Unit]
Description=Podman auto-update timer

[Timer]
OnCalendar=daily
Persistent=true

[Install]
WantedBy=timers.target

```

この例では、**podman auto-update** コマンドは、毎日、深夜に起動します。

3. システム起動時に **podman-auto-update.timer** サービスを有効にします。

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

4. **systemd** サービスを開始します。

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

5. オプション: タイマーをリスト表示します。

```

# systemctl list-timers --all
NEXT                LEFT  LAST                PASSED  UNIT
ACTIVATES
Wed 2020-12-09 00:00:00 CET 9h left  n/a          n/a      podman-auto-
update.timer  podman-auto-update.service

```

podman-auto-update.timer が **podman-auto-update.service** を有効化したことが確認できます。

関連情報

- [Systemd Integration with Podman 2.0](#)
- [Running containers with Podman and shareable systemd services](#)
- [ブート時のシステムサービス起動の有効化](#)

第15章 ANSIBLE PLAYBOOK を使用したコンテナの管理

Podman 4.2 以降では、Podman RHEL システムロールを使用して、Podman 設定、コンテナ、および Podman コンテナを実行する systemd サービスを管理できます。

RHEL システムロールは、複数の RHEL システムをリモートで管理するための設定インターフェイスを提供します。このインターフェイスを使用すると、RHEL の複数のバージョンにわたるシステム設定を管理したり、新しいメジャーリリースを導入したりできます。詳細は、[RHEL System Roles を使用したシステム管理の自動化](#) を参照してください。

15.1. バインドマウントを使用したルートレスコンテナの作成

podman RHEL システムロールを使用すると、Ansible Playbook を実行してバインドマウントによりルートレスコンテナを作成し、アプリケーション設定を管理できます。

前提条件

- [制御ノードと管理ノードを準備している](#)
- 管理対象ノードで Playbook を実行できるユーザーとしてコントロールノードにログインしている。
- 管理対象ノードへの接続に使用するアカウントに、そのノードに対する **sudo** 権限がある。

手順

1. 次の内容を含む Playbook ファイル (例: `~/playbook.yml`) を作成します。

```
- hosts: managed-node-01.example.com
vars:
  podman_create_host_directories: true
  podman_firewall:
    - port: 8080-8081/tcp
      state: enabled
    - port: 12340/tcp
      state: enabled
  podman_selinux_ports:
    - ports: 8080-8081
      setype: http_port_t
  podman_kube_specs:
    - state: started
      run_as_user: dbuser
      run_as_group: dbgroup
      kube_file_content:
        apiVersion: v1
        kind: Pod
        metadata:
          name: db
        spec:
          containers:
            - name: db
              image: quay.io/db/db:stable
              ports:
                - containerPort: 1234
                  hostPort: 12340
```



```

    volumeMounts:
      - mountPath: /var/lib/db:Z
        name: db
    volumes:
      - name: db
        hostPath:
          path: /var/lib/db
    - state: started
      run_as_user: webapp
      run_as_group: webapp
      kube_file_src: /path/to/webapp.yml
  roles:
    - linux-system-roles.podma

```

この手順では、2つのコンテナを持つ Pod を作成します。**podman_kube_specs** ロール変数は Pod を記述します。

- **run_as_user** フィールドと **run_as_group** フィールドは、コンテナがルートレスであることを指定します。
- Kubernetes YAML ファイルを含む **kube_file_content** フィールドは、**db** という名前の最初のコンテナを定義します。**podman kube generate** コマンドを使用して Kubernetes YAML ファイルを生成できます。
 - **db** コンテナは、**quay.io/db/db:stable** コンテナイメージに基づいています。
 - **db** バインドマウントは、ホスト上の **/var/lib/db** ディレクトリーをコンテナ内の **/var/lib/db** ディレクトリーにマップします。**Z** フラグはコンテンツにプライベート非共有ラベルを付けるため、**db** コンテナのみがコンテンツにアクセスできます。
- **kube_file_src** フィールドは2番目のコンテナを定義します。コントローラーノードの **/path/to/webapp.yml** ファイルの内容は、マネージドノードの **kube_file** フィールドにコピーされます。
- ホスト上にディレクトリーを作成するには、**podman_create_host_directories: true** を設定します。これにより、**hostPath** ボリュームの kube 仕様を確認し、ホスト上にそれらのディレクトリーを作成するようにロールに指示します。所有権と権限をさらに細かく制御する必要がある場合は、**podman_host_directories** を使用します。

2. Playbook の構文を検証します。

```
$ ansible-playbook --syntax-check --ask-vault-pass ~/playbook.yml
```

このコマンドは構文を検証するだけであり、有効だが不適切な設定から保護するものではないことに注意してください。

3. Playbook を実行します。

```
$ ansible-playbook --ask-vault-pass ~/playbook.yml
```

関連情報

- **/usr/share/ansible/roles/rhel-system-roles.podman/README.md** ファイル
- **/usr/share/doc/rhel-system-roles/podman/** ディレクトリー

15.2. PODMAN ボリュームを使用した ROOTFUL コンテナの作成

podman RHEL システムロールを使用すると、Ansible Playbook を実行して Podman ボリュームを持つルートフルコンテナを作成し、アプリケーション設定を管理できます。

前提条件

- [制御ノードと管理ノードを準備している](#)
- 管理対象ノードで Playbook を実行できるユーザーとしてコントロールノードにログインしている。
- 管理対象ノードへの接続に使用するアカウントに、そのノードに対する **sudo** 権限がある。

手順

1. 次の内容を含む Playbook ファイル (例: `~/playbook.yml`) を作成します。

```
- hosts: managed-node-01.example.com
vars:
  podman_firewall:
    - port: 8080/tcp
      state: enabled
  podman_kube_specs:
    - state: started
      kube_file_content:
        apiVersion: v1
        kind: Pod
        metadata:
          name: ubi8-httpd
        spec:
          containers:
            - name: ubi8-httpd
              image: registry.access.redhat.com/ubi8/httpd-24
              ports:
                - containerPort: 8080
                  hostPort: 8080
              volumeMounts:
                - mountPath: /var/www/html:Z
                  name: ubi8-html
          volumes:
            - name: ubi8-html
              persistentVolumeClaim:
                claimName: ubi8-html-volume
  roles:
    - linux-system-roles.podman
```

この手順では、1つのコンテナを含む Pod を作成します。**podman_kube_specs** ロール変数は Pod を記述します。

- デフォルトでは、**podman** ロールはルートフルコンテナを作成します。
- Kubernetes YAML ファイルを含む **kube_file_content** フィールドは、**ubi8-httpd** という名前のコンテナを定義します。
 - **ubi8-httpd** コンテナは、**registry.access.redhat.com/ubi8/httpd-24** コンテナイ

メッセージに基づいています。

- **ubi8-html-volume** は、ホスト上の **/var/www/html** ディレクトリーをコンテナにマップします。**Z** フラグはコンテンツにプライベート非共有ラベルを付けるため、**ubi8-httpd** コンテナのみがコンテンツにアクセスできます。
- Pod は、マウントパス **/var/www/html** を使用して、**ubi8-html-volume** という名前の既存の永続ボリュームをマウントします。

2. Playbook の構文を検証します。

```
$ ansible-playbook --syntax-check ~/playbook.yml
```

このコマンドは構文を検証するだけであり、有効だが不適切な設定から保護するものではないことに注意してください。

3. Playbook を実行します。

```
$ ansible-playbook ~/playbook.yml
```

関連情報

- **/usr/share/ansible/roles/rhel-system-roles.podman/README.md** ファイル
- **/usr/share/doc/rhel-system-roles/podman/** ディレクトリー

15.3. シークレットを使用した QUADLET アプリケーションの作成

podman RHEL システムロールを使用して、Ansible Playbook を実行することで、シークレットを含む Quadlet アプリケーションを作成できます。

前提条件

- [制御ノードと管理ノードを準備している](#)
- 管理対象ノードで Playbook を実行できるユーザーとしてコントロールノードにログインしている。
- 管理対象ノードへの接続に使用するアカウントに、そのノードに対する **sudo** 権限がある。
- コンテナ内の Web サーバーが使用する証明書と対応する秘密鍵は、**~/certificate.pem** ファイルと **~/key.pem** ファイルに保存されます。

手順

1. 証明書と秘密鍵ファイルの内容を表示します。

```
$ cat ~/certificate.pem
-----BEGIN CERTIFICATE-----
...
-----END CERTIFICATE-----

$ cat ~/key.pem
```

```
-----BEGIN PRIVATE KEY-----
...
-----END PRIVATE KEY-----
```

この情報は後の手順で必要になります。

2. 機密性の高い変数を暗号化されたファイルに保存します。
 - a. vault を作成します。

```
$ ansible-vault create vault.yml
New Vault password: <vault_password>
Confirm New Vault password: <vault_password>
```

- b. **ansible-vault create** コマンドでエディターが開いたら、機密データを **<key>: <value>** 形式で入力します。

```
root_password: <root_password>
certificate: |-
  -----BEGIN CERTIFICATE-----
  ...
  -----END CERTIFICATE-----
key: |-
  -----BEGIN PRIVATE KEY-----
  ...
  -----END PRIVATE KEY-----
```

certificate 変数および **key** 変数のすべての行が 2 つのスペースで始まるようにします。

- c. 変更を保存して、エディターを閉じます。Ansible は vault 内のデータを暗号化します。
3. 次の内容を含む Playbook ファイル (例: **~/playbook.yml**) を作成します。

```
- name: Deploy a wordpress CMS with MySQL database
  hosts: managed-node-01.example.com
  vars_files:
    - vault.yml
  tasks:
    - name: Create and run the container
      ansible.builtin.include_role:
        name: rhel-system-roles.podman
      vars:
        podman_create_host_directories: true
        podman_activate_systemd_unit: false
        podman_quadlet_specs:
          - name: quadlet-demo
            type: network
            file_content: |
              [Network]
              Subnet=192.168.30.0/24
              Gateway=192.168.30.1
              Label=app=wordpress
          - file_src: quadlet-demo-mysql.volume
          - template_src: quadlet-demo-mysql.container.j2
          - file_src: envoy-proxy-configmap.yml
          - file_src: quadlet-demo.yml
```

```

- file_src: quadlet-demo.kube
  activate_systemd_unit: true
podman_firewall:
- port: 8000/tcp
  state: enabled
- port: 9000/tcp
  state: enabled
podman_secrets:
- name: mysql-root-password-container
  state: present
  skip_existing: true
  data: "{{ root_password }}"
- name: mysql-root-password-kube
  state: present
  skip_existing: true
  data: |
    apiVersion: v1
    data:
      password: "{{ root_password | b64encode }}"
    kind: Secret
    metadata:
      name: mysql-root-password-kube
- name: envoy-certificates
  state: present
  skip_existing: true
  data: |
    apiVersion: v1
    data:
      certificate.key: {{ key | b64encode }}
      certificate.pem: {{ certificate | b64encode }}
    kind: Secret
    metadata:
      name: envoy-certificates

```

この手順では、MySQL データベースと組み合わせた WordPress コンテンツ管理システムを作成します。**podman_quadlet_specs** ロール変数では、Quadlet の一連の設定を定義します。この設定は、特定の 방법으로連携するコンテナまたはサービスのグループを参照します。これには次の仕様を含めます。

- Wordpress ネットワークを、**quadlet-demo** ネットワークユニットで定義します。
- MySQL コンテナのボリューム設定を、**file_src: quadlet-demo-mysql.volume** フィールドで定義します。
- **template_src: quadlet-demo-mysql.container.j2** フィールドを使用して、MySQL コンテナの設定を生成します。
- その後に、2つのYAMLファイル **file_src: envoy-proxy-configmap.yml** および **file_src:quadlet-demo.yml** を指定します。.yml は有効な Quadlet ユニットタイプではないため、これらのファイルはコピーされるだけで、Quadlet 仕様としては処理されないことに注意してください。
- Wordpress および envoy プロキシコンテナと設定を、**file_src: quadlet-demo.kube** フィールドで定義します。kube ユニットは、**[Kube]** セクション内の上記のYAML ファイルを、**Yaml=quadlet-demo.yml** および **ConfigMap=envoy-proxy-configmap.yml** として参照します。

4. Playbook の構文を検証します。

```
$ ansible-playbook --syntax-check --ask-vault-pass ~/playbook.yml
```

このコマンドは構文を検証するだけであり、有効だが不適切な設定から保護するものではないことに注意してください。

5. Playbook を実行します。

```
$ ansible-playbook --ask-vault-pass ~/playbook.yml
```

関連情報

- [/usr/share/ansible/roles/rhel-system-roles.podman/README.md](#) ファイル
- [/usr/share/doc/rhel-system-roles/podman/](#) ディレクトリー

第16章 RHEL WEB コンソールを使用したコンテナイメージの管理

RHEL Web コンソールの Web ベースのインターフェイスを使用して、コンテナイメージをプル、プルーニング、または削除できます。

16.1. WEB コンソールでのコンテナイメージの取得

コンテナイメージをローカルシステムにダウンロードし、それを使用してコンテナを作成できます。

前提条件

- RHEL 8 Web コンソールをインストールし、アクセスできる。詳細は、[Web コンソールのインストール](#) および [Web コンソールへのログイン](#) を参照してください。
- **cockpit-podman** アドオンをインストールしている。

```
# dnf install cockpit-podman
```

手順

1. メインメニューで **Podman containers** をクリックします。
2. **Images** テーブルで、右上隅にあるオーバーフローメニューをクリックし、**Download new image** を選択します。
3. **Search for an image** ダイアログボックスが表示されます。
4. **Search for** フィールドに、イメージの名前を入力するか、その説明を指定します。
5. **in** ドロップダウンリストで、イメージを取得するレジストリーを選択します。
6. オプション: **Tag** フィールドに、イメージのタグを入力します。
7. **Download** をクリックします。

検証

- メインメニューで **Podman containers** をクリックします。新しくダウンロードしたイメージは、**Images** テーブルで確認できます。



注記

Images テーブルで **Create container** をクリックすると、ダウンロードしたイメージからコンテナを作成できます。コンテナを作成するには、[Web コンソールでのコンテナの作成](#) の手順 3 - 8 に従います。

16.2. WEB コンソールでのコンテナイメージのプルーニング

コンテナを持たない未使用のイメージをすべて削除できます。

前提条件

- 少なくとも1つのコンテナイメージがプルされます。
- RHEL 8 Web コンソールをインストールし、アクセスできる。詳細は、[Web コンソールのインストール](#) および [Web コンソールへのログイン](#) を参照してください。
- **cockpit-podman** アドオンをインストールしている。

```
# dnf install cockpit-podman
```

手順

1. メインメニューで **Podman containers** をクリックします。
2. **Images** テーブルで、右上隅のオーバーフローメニューをクリックし、**Prune unused images** を選択します。
3. イメージのリストを含むポップアップウィンドウが表示されます。**Prune** をクリックして選択を確定します。

検証

- メインメニューで **Podman containers** をクリックします。削除されたイメージは、**Images** テーブルにリストされません。

16.3. WEB コンソールでのコンテナイメージの削除

Web コンソールを使用して、以前にプルしたコンテナイメージを削除できます。

前提条件

- 少なくとも1つのコンテナイメージがプルされます。
- RHEL 8 Web コンソールをインストールし、アクセスできる。詳細は、[Web コンソールのインストール](#) および [Web コンソールへのログイン](#) を参照してください。
- **cockpit-podman** アドオンをインストールしている。

```
# dnf install cockpit-podman
```

手順

1. メインメニューで **Podman containers** をクリックします。
2. **Images** テーブルで、削除するイメージを選択し、オーバーフローメニューをクリックして **Delete** を選択します。
3. ウィンドウが表示されます。**Delete tagged images** をクリックして選択を確認します。

検証

- メインメニューで **Podman containers** をクリックします。削除されたコンテナは、**Images** テーブルにリストされません。

第17章 RHEL WEB コンソールを使用したコンテナの管理

Red Hat Enterprise Linux Web コンソールを使用して、コンテナと Pod を管理できます。Web コンソールを使用すると、非 root または root ユーザーとしてコンテナを作成できます。

- root ユーザーとして、追加の権限とオプションを備えたシステムコンテナを作成できます。
- 非 root ユーザーには2つのオプションがあります。
 - ユーザーコンテナのみを作成するには、Web コンソールをデフォルトモード (**Limited access**) で使用できます。
 - ユーザーコンテナとシステムコンテナの両方を作成するには、Web コンソールページの上部パネルで **Administrative access** をクリックします。

ルートコンテナとルートレスコンテナの違いの詳細については、[ルートレスコンテナに関する特別な考慮事項](#) を参照してください。

17.1. WEB コンソールでのコンテナの作成

コンテナを作成し、ポートマッピング、ボリューム、環境変数、ヘルスチェックなどを追加できます。

前提条件

- RHEL 8 Web コンソールをインストールし、アクセスできる。詳細は、[Web コンソールのインストール](#) および [Web コンソールへのログイン](#) を参照してください。
- **cockpit-podman** アドオンをインストールしている。

```
# dnf install cockpit-podman
```

手順

1. メインメニューで **Podman containers** をクリックします。
2. **Create container** をクリックします。
3. **Name** フィールドに、コンテナの名前を入力します。
4. **Details** タブに必要な情報を入力します。
 - **管理者アクセス権がある場合のみ利用可能**:コンテナの所有者を選択します。システムまたはユーザー。
 - **Image** ドロップダウンリストで、選択したレジストリー内のコンテナイメージを選択または検索します。
 - オプション: 最新のコンテナイメージをプルするには、**Pull latest image** チェックボックスをオンにします。
 - **Command** フィールドはコマンドを指定します。必要に応じて、デフォルトのコマンドを変更できます。
 - オプション: ターミナルを使用してコンテナを実行するには、**With terminal** チェックボックスをオンにします。

- **Memory limit** フィールドでは、コンテナのメモリー制限を指定します。デフォルトのメモリー制限を変更するには、チェックボックスをオンにして制限を指定します。
 - **システムコンテナでのみ利用可能:CPU shares** フィールドに、相対的な CPU 時間量を指定します。デフォルト値は 1024 です。デフォルト値を変更するには、チェックボックスをオンにします。
 - **システムコンテナでのみ利用可能:Restart policy** ドロップダウンメニューで、次のオプションのいずれかを選択します。
 - **No** (デフォルト値): 何もしません。
 - **On Failure**: 失敗時にコンテナを再起動します。
 - **Always**: コンテナの終了時、またはシステムの再起動後にコンテナを再起動します。
5. **Integration** タブに必要な情報を入力します。
- **Add port mapping** をクリックして、コンテナとホストシステムの間にポートマッピングを追加します。
 - IP アドレス、ホストポート、コンテナポート、および **プロトコル** を入力します。
 - **Add volume** をクリックしてボリュームを追加します。
 - **ホストパス**、**コンテナパス** を入力します。**Writable** オプションのチェックボックスをオンにして、書き込み可能なボリュームを作成できます。SELinux ドロップダウンリストで、**No Label**、**Shared**、または **Private** のいずれかのオプションを選択します。
 - **Add variable** をクリックして環境変数を追加します。
 - **Key** と **Value** を入力します。
6. **Health check** タブに必要な情報を入力します。
- **Command** フィールドに、'healthcheck' コマンドを入力します。
 - **ヘルスチェックオプション**を指定します。
 - **Interval** (デフォルトは 30 秒)
 - **Timeout** (デフォルトは 30 秒)
 - **Start period**
 - **Retries** (デフォルトは 3)
 - **When unhealthy**: 以下のオプションのいずれかを選択します。
 - **No action** (デフォルト): 何もしません。
 - **Restart**: コンテナを再起動します。
 - **Stop**: コンテナを停止します。
 - **Force stop**: コンテナを強制的に停止します。コンテナが終了するのを待ちません。

7. **Create and run** をクリックして、コンテナを作成して実行します。



注記

Create をクリックすると、コンテナのみを作成できます。

検証

- メインメニューで **Podman containers** をクリックします。新しく作成されたコンテナは **Containers** テーブルで確認できます。

17.2. WEB コンソールでのコンテナの検査

Web コンソールでコンテナの詳細情報を表示できます。

前提条件

- コンテナが作成されている。
- RHEL 8 Web コンソールをインストールし、アクセスできる。詳細は、[Web コンソールのインストール](#) および [Web コンソールへのログイン](#) を参照してください。
- **cockpit-podman** アドオンをインストールしている。

```
# dnf install cockpit-podman
```

手順

1. メインメニューで **Podman containers** をクリックします。
2. > 矢印アイコンをクリックすると、コンテナの詳細が表示されます。
 - **Details** タブでは、コンテナ ID、イメージ、コマンド、作成済み (コンテナ作成時のタイムスタンプ)、およびその状態を確認できます。
 - システムコンテナでのみ利用可能:IP アドレス、MAC アドレス、ゲートウェイアドレスも確認できます。
 - **Integration** タブでは、環境変数、ポートマッピング、およびボリュームを確認できます。
 - **Log** タブでは、コンテナのログを確認できます。
 - **Console** タブでは、コマンドラインを使用してコンテナを操作できます。

17.3. WEB コンソールでのコンテナの状態の変更

Red Hat Enterprise Linux Web コンソールでは、システム上のコンテナの起動、停止、再起動、一時停止、および名前変更が可能です。

前提条件

- コンテナが作成されている。

- RHEL 8 Web コンソールをインストールし、アクセスできる。詳細は、[Web コンソールのインストール](#) および [Web コンソールへのログイン](#) を参照してください。
- **cockpit-podman** アドオンをインストールしている。

```
# dnf install cockpit-podman
```

手順

1. メインメニューで **Podman containers** をクリックします。
2. **Containers** テーブルで、変更するコンテナを選択し、オーバーフローメニューをクリックして、実行するアクションを選択します。
 - Start
 - Stop
 - Force stop
 - Restart
 - Force restart
 - Pause
 - Rename

17.4. WEB コンソールでのコンテナのコミット

コンテナの現在の状態に基づいて新しいイメージを作成できます。

前提条件

- コンテナが作成されている。
- RHEL 8 Web コンソールをインストールし、アクセスできる。詳細は、[Web コンソールのインストール](#) および [Web コンソールへのログイン](#) を参照してください。
- **cockpit-podman** アドオンをインストールしている。

```
# dnf install cockpit-podman
```

手順

1. メインメニューで **Podman containers** をクリックします。
2. **Containers** テーブルで、変更するコンテナを選択し、オーバーフローメニューをクリックして **Commit** を選択します。
3. **Commit container** フォームに、次の詳細を追加します。
 - **New image name** フィールドにイメージ名を入力します。
 - オプション: **Tag** フィールドにタグを入力します。

- オプション: **Author** フィールドに名前を入力します。
- オプション: 必要に応じて、**Command** フィールドでコマンドを変更します。
- オプション: 必要な **オプション** を確認してください。
 - イメージの作成時にコンテナを一時停止します。イメージがコミットされている間、コンテナとそのプロセスは一時停止されます。
 - 従来の Docker 形式を使用する: Docker イメージ形式を使用しない場合は、OCI 形式が使用されます。

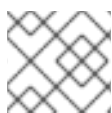
4. **Commit** をクリックします。

検証

- メインメニューで **Podman containers** をクリックします。新しく作成されたイメージは **Images** テーブルで確認できます。

17.5. WEB コンソールでのコンテナチェックポイントの作成

Web コンソールを使用すると、実行中のコンテナまたは個々のアプリケーションにチェックポイントを設定し、その状態をディスクに保存できます。



注記

チェックポイントの作成は、システムコンテナでのみ使用できます。

前提条件

- コンテナが実行されている。
- RHEL 8 Web コンソールをインストールし、アクセスできる。詳細は、[Web コンソールのインストール](#) および [Web コンソールへのログイン](#) を参照してください。
- **cockpit-podman** アドオンをインストールしている。

```
# dnf install cockpit-podman
```

手順

1. メインメニューで **Podman containers** をクリックします。
2. **Containers** テーブルで、変更するコンテナを選択し、オーバーフローアイコンメニューをクリックして **Containers** を選択します。
3. オプション: **Checkpoint container** フォームで、必要なオプションをチェックします。
 - すべての一時チェックポイントファイルを保持する: チェックポイント作成中に CRIU によって作成されたすべての一時ログおよび統計ファイルを保持します。さらなるデバッグのためのチェックポイント設定が失敗した場合でも、これらのファイルは削除されません。
 - チェックポイントをディスクに書き込んだ後も実行したままにする: チェックポイントを作成した後もコンテナを停止するのではなく、実行したままにします。

- 確立された TCP 接続の維持のサポート

4. **Checkpoint** をクリックします。

検証

- メインメニューで **Podman containers** をクリックします。チェックポイントを設定したコンテナを選択し、オーバーフローメニューアイコンをクリックして、**Restore** オプションがあることを確認します。

17.6. WEB コンソールでのコンテナチェックポイントの復元

保存したデータを使用して、再起動後に、チェックポイントの時点にコンテナを復元できます。



注記

チェックポイントの作成は、システムコンテナでのみ使用できます。

前提条件

- コンテナにチェックポイントが設定されている。
- RHEL 8 Web コンソールをインストールし、アクセスできる。詳細は、[Web コンソールのインストール](#) および [Web コンソールへのログイン](#) を参照してください。
- **cockpit-podman** アドオンをインストールしている。

```
# dnf install cockpit-podman
```

手順

1. メインメニューで **Podman containers** をクリックします。
2. **Containers** テーブルで、変更するコンテナを選択し、オーバーフローメニューをクリックして **Restore** を選択します。
3. オプション: **Restore container** フォームで、必要なオプションを確認します。
 - **Keep all temporary checkpoint files:** チェックポイント設定中に CRIU によって作成されたすべての一時ログおよび統計ファイルを保持します。さらなるデバッグのためのチェックポイント設定が失敗した場合でも、これらのファイルは削除されません。
 - **Restore with established TCP connections**
 - **Ignore IP address if set statically** コンテナが IP アドレスを使用して開始された場合、復元されたコンテナもその IP アドレスを使用しようとし、その IP アドレスがすでに使用されている場合は復元は失敗します。このオプションは、コンテナの作成時に Integration タブでポートマッピングを追加した場合に適用されます。
 - **Ignore MAC address if set statically** コンテナが MAC アドレスで開始された場合、復元されたコンテナもその MAC アドレスを使用しようとし、その MAC アドレスがすでに使用されている場合は復元は失敗します。
4. **Restore** をクリックします。

検証

- メインメニューで **Podman containers** クリックします。Containers テーブルで復元されたコンテナが実行されていることがわかります。

17.7. WEB コンソールでのコンテナの削除

Web コンソールを使用して既存のコンテナを削除できます。

前提条件

- コンテナがシステムに存在する。
- RHEL 8 Web コンソールをインストールし、アクセスできる。詳細は、[Web コンソールのインストール](#) および [Web コンソールへのログイン](#) を参照してください。
- **cockpit-podman** アドオンをインストールしている。

```
# dnf install cockpit-podman
```

手順

1. メインメニューで **Podman containers** をクリックします。
2. Containers テーブルで、削除するコンテナを選択し、オーバーフローメニューをクリックして **Delete** を選択します。
3. ポップアップウィンドウが表示されます。Delete をクリックして選択を確定します。

検証

- メインメニューで **Podman containers** クリックします。削除されたコンテナは、Containers テーブルにリストされません。

17.8. WEB コンソールでの POD の作成

RHEL Web コンソールインターフェイスで Pod を作成できます。

前提条件

- RHEL 8 Web コンソールをインストールし、アクセスできる。詳細は、[Web コンソールのインストール](#) および [Web コンソールへのログイン](#) を参照してください。
- **cockpit-podman** アドオンをインストールしている。

```
# dnf install cockpit-podman
```

手順

1. メインメニューで **Podman containers** をクリックします。
2. **Create pod** をクリックします。
3. Create pod フォームに必要な情報を入力します。

- **管理者アクセス権がある場合のみ利用可能:**コンテナの所有者を選択します。システムまたはユーザー。
 - **Name** フィールドに、コンテナの名前を入力します。
 - **Add port mapping** をクリックして、コンテナとホストシステムの間ポートマッピングを追加します。
 - IP アドレス、ホストポート、コンテナポート、プロトコルを入力します。
 - **Add volume** をクリックしてボリュームを追加します。
 - ホストパス、コンテナパスを入力します。書き込み可能チェックボックスをオンにして、書き込み可能なボリュームを作成できます。SELinux ドロップダウンリストで、No Label、Shared、または Private。
4. **Create** をクリックします。

検証

- メインメニューで **Podman containers** をクリックします。新しく作成された Pod は **Containers** テーブルで確認できます。

17.9. WEB コンソールの POD 内にコンテナを作成する

Pod 内にコンテナを作成できます。

前提条件

- RHEL 8 Web コンソールをインストールし、アクセスできる。詳細は、[Web コンソールのインストール](#) および [Web コンソールへのログイン](#) を参照してください。
- **cockpit-podman** アドオンをインストールしている。

```
# dnf install cockpit-podman
```

手順

1. メインメニューで **Podman containers** をクリックします。
2. **Create container in pod** をクリックします。
3. **Name** フィールドに、コンテナの名前を入力します。
4. **Details** タブに必要な情報を入力します。
 - **管理者アクセス権がある場合のみ利用可能:**コンテナの所有者を選択します。システムまたはユーザー。
 - **Image** ドロップダウンリストで、選択したレジストリー内のコンテナイメージを選択または検索します。
 - オプション: 最新のコンテナイメージをプルするには、**Pull latest image** チェックボックスをオンにします。

- **Command** フィールドはコマンドを指定します。必要に応じて、デフォルトのコマンドを変更できます。
 - オプション: ターミナルを使用してコンテナを実行するには、**With terminal** チェックボックスをオンにします。
- **Memory limit** フィールドでは、コンテナのメモリ制限を指定します。デフォルトのメモリ制限を変更するには、チェックボックスをオンにして制限を指定します。
- **システムコンテナでのみ利用可能:CPU shares** フィールドに、相対的な CPU 時間を指定します。デフォルト値は 1024 です。デフォルト値を変更するには、チェックボックスをオンにします。
- **システムコンテナでのみ利用可能:Restart policy** ドロップダウンメニューで、次のオプションのいずれかを選択します。
 - **No** (デフォルト値): 何もしません。
 - **On Failure**: 失敗時にコンテナを再起動します。
 - **Always**: コンテナの終了時またはシステム起動後にコンテナを再起動します。

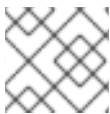
5. Integration タブに必要な情報を入力します。

- **Add port mapping** をクリックして、コンテナとホストシステムの間ポートマッピングを追加します。
 - IP アドレス、ホストポート、コンテナポート、および **プロトコル** を入力します。
- **Add volume** をクリックしてボリュームを追加します。
 - ホストパス、コンテナパス を入力します。 **Writable** オプションのチェックボックスをオンにして、書き込み可能なボリュームを作成できます。SELinux ドロップダウンリストで、**No Label**、**Shared**、または **Private** のいずれかのオプションを選択します。
- **Add variable** をクリックして環境変数を追加します。
 - **Key** と **Value** を入力します。

6. Health check タブに必要な情報を入力します。

- **Command** フィールドに、ヘルスチェックコマンドを入力します。
- ヘルスチェックオプションを指定します。
 - **Interval** (デフォルトは 30 秒)
 - **Timeout** (デフォルトは 30 秒)
 - **Start period**
 - **Retries** (デフォルトは 3)
 - **When unhealthy**: 以下のオプションのいずれかを選択します。
 - **No action** (デフォルト): 何もしません。
 - **Restart**: コンテナを再起動します。

- **Stop:** コンテナを停止します。
- **Force stop:** コンテナを強制的に停止します。コンテナが終了するのを待ちません。



注記

コンテナの所有者は Pod の所有者と同じです。



注記

Pod では、コンテナの検査、コンテナのステータスの変更、コンテナのコミット、またはコンテナの削除を行うことができます。

検証

- メインメニューで **Podman containers** をクリックします。Pod 内の **Containers** テーブルの下に、新しく作成されたコンテナが表示されます。

17.10. WEB コンソールでの POD の状態の変更

Pod のステータスを変更できます。

前提条件

- Pod が作成済みである。
- RHEL 8 Web コンソールをインストールし、アクセスできる。詳細は、[Web コンソールのインストール](#) および [Web コンソールへのログイン](#) を参照してください。
- **cockpit-podman** アドオンをインストールしている。

```
# dnf install cockpit-podman
```

手順

1. メインメニューで **Podman containers** をクリックします。
2. **Containers** テーブルで、変更する Pod を選択し、オーバーフローメニューをクリックして、実行するアクションを選択します。
 - **Start**
 - **Stop**
 - **Force stop**
 - **Restart**
 - **Force restart**
 - **Pause**

17.11. WEB コンソールでの POD の削除

Web コンソールを使用して既存の Pod を削除できます。

前提条件

- Pod がシステムに存在する。
- RHEL 8 Web コンソールをインストールし、アクセスできる。詳細は、[Web コンソールのインストール](#) および [Web コンソールへのログイン](#) を参照してください。
- **cockpit-podman** アドオンをインストールしている。

```
# dnf install cockpit-podman
```

手順

1. メインメニューで **Podman containers** をクリックします。
2. **Containers** テーブルで、削除する Pod を選択し、オーバーフローメニューをクリックして **Delete** を選択します。
3. 次のポップアップウィンドウで、**Delete** をクリックして選択を確定します。



警告

Pod 内のすべてのコンテナが削除されます。

検証

- メインメニューで **Podman containers** をクリックします。削除された Pod は、**Containers** テーブルにリストされません。

第18章 コンテナでの SKOPEO、BUILDDAH、および PODMAN の実行

コンテナで Skopeo、Buildah、および Podman を実行できます。

Skopeo を使用すると、すべてのレイヤーでイメージ全体をダウンロードすることなく、リモートレジストリーのイメージを検査できます。また、Skopeo を使用して、イメージのコピー、イメージの署名、イメージの同期、さまざまな形式およびレイヤー圧縮でのイメージ変換を行うこともできます。

Buildah を使用すると、OCI コンテナイメージのビルドが容易になります。Buildah を使用すると、作業コンテナをゼロから作成することも、イメージを開始点として使用して作成することも可能です。作業コンテナからか、**Containerfile** の指示を使用して、イメージを作成できます。作業コンテナの root ファイルシステムをマウントおよびアンマウントできます。

Podman を使用すると、コンテナおよびイメージ、それらのコンテナにマウントされたボリューム、およびコンテナのグループから作成された Pod を管理できます。Podman は、コンテナライフサイクル管理の **libpod** ライブラリーに基づいています。**libpod** ライブラリーは、コンテナ、Pod、コンテナイメージ、およびボリュームを管理するための API を提供します。

コンテナで Buildah、Skopeo、および Podman を実行する理由:

- **CI/CD システム:**
 - **Podman および Skopeo:** Kubernetes で CI/CD システムを実行するか、OpenShift を使用してコンテナイメージをビルドし、これらのイメージを異なるコンテナレジストリーに配布できます。Skopeo を Kubernetes ワークフローに統合するには、Skopeo をコンテナで実行する必要があります。
 - **Buildah:** 常にイメージをビルドする Kubernetes または OpenShift CI/CD システムに OCI/コンテナイメージをビルドします。以前は、Docker ソケットを使用してコンテナエンジンに接続し、**docker build** コマンドを実行していました。これは、パスワードなしにシステムに root アクセスを提供するのと同じで、安全ではありません。このため、Red Hat はコンテナで Buildah を使用することを推奨します。
- **各種バージョン:**
 - **All:** ホストで古いオペレーティングシステムを実行していても、最新バージョンの Skopeo、Buildah、または Podman を実行します。このソリューションは、コンテナでコンテナツールを実行します。たとえば、これは、最新版をネイティブで使用できない Red Hat Enterprise Linux 7 コンテナホストで、Red Hat Enterprise Linux 8 で提供される最新版のコンテナツールを実行するのに役立ちます。
- **HPC 環境:**
 - **All:** HPC 環境で一般的な制限は、root 以外のユーザーがホストにパッケージをインストールできないことです。コンテナで Skopeo、Buildah、または Podman を実行すると、root 以外のユーザーとしてこの特定のタスクを実行することができます。

18.1. コンテナでの SKOPEO の実行

Skopeo を使用してリモートコンテナイメージを検査できます。コンテナで Skopeo を実行すると、コンテナの root ファイルシステムがホストの root ファイルシステムから分離されることを意味します。ホストとコンテナ間でファイルを共有またはコピーするには、ファイルとディレクトリーをマウントする必要があります。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. registry.redhat.io レジストリーにログインします。

```
$ podman login registry.redhat.io
Username: myuser@mycompany.com
Password: <password>
Login Succeeded!
```

2. **registry.redhat.io/rhel9/skopeo** コンテナイメージを取得します。

```
$ podman pull registry.redhat.io/rhel9/skopeo
```

3. Skopeo を使用して、リモートコンテナイメージ **registry.access.redhat.com/ubi9/ubi** を検査します。

```
$ podman run --rm registry.redhat.io/rhel9/skopeo \
  skopeo inspect docker://registry.access.redhat.com/ubi9/ubi
{
  "Name": "registry.access.redhat.com/ubi9/ubi",
  ...
  "Labels": {
    "architecture": "x86_64",
    ...
    "name": "ubi9",
    ...
    "summary": "Provides the latest release of Red Hat Universal Base Image 9.",
    "url":
      "https://access.redhat.com/containers/#/registry.access.redhat.com/ubi9/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 オプションは、コンテナの終了後に **registry.redhat.io/rhel9/skopeo** イメージを削除します。

関連情報

- [How to run skopeo in a container](#)

18.2. 認証情報を使用したコンテナでの SKOPEO の実行

コンテナレジストリーを使用する場合には、データへのアクセスや変更に認証が必要です。skopeo は、さまざまな認証情報の指定方法に対応します。

このアプローチでは、**--cred USERNAME[:PASSWORD]** オプションを使用してコマンドラインで認証情報を指定できます。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

- ロックされたレジストリーに対して Skopeo を使用してリモートコンテナイメージを検証します。

```
$ podman run --rm registry.redhat.io/rhel9/skopeo inspect --creds
$USER:$PASSWORD docker://$IMAGE
```

関連情報

- [How to run skopeo in a container](#)

18.3. AUTHFILE を使用したコンテナでの SKOPEO の実行

認証ファイル (authfile) を使用して認証情報を指定できます。**skopeo login** コマンドは、特定のレジストリーにログインし、認証トークンを authfile に保存します。authfiles を使用する利点は、認証情報を繰り返し入力する必要がないことです。

同じホストで実行する場合、Skopeo、Buildah、Podman などのすべてのコンテナツールで同じ authfile を共有します。コンテナで Skopeo を実行する場合は、コンテナに authfile をボリュームマウントしてホスト上で authfile を共有するか、コンテナ内で再認証する必要があります。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

- ロックされたレジストリーに対して Skopeo を使用してリモートコンテナイメージを検証します。

```
$ podman run --rm -v $AUTHFILE:/auth.json registry.redhat.io/rhel9/skopeo inspect
docker://$IMAGE
```

-v \$AUTHFILE:/auth.json オプションの volume-mounts は、コンテナ内の /auth.json に authfile をマウントします。Skopeo は、ホストの authfile の認証トークンにアクセスでき、レジストリーにセキュアにアクセスできるようになります。

その他の Skopeo コマンドは、以下の例のように機能します。

- **skopeo-copy** コマンドで **--source-creds** および **--dest-creds** オプションを使用して、ソースおよび宛先のイメージに、コマンドラインで認証情報を指定します。また、**/auth.json** の authfile も読み取ります。

- ソースイメージおよび宛先イメージに個別の authfile を指定する場合は、**--source-authfile** オプションおよび **--dest-authfile** オプションを使用し、ホストからコンテナにこれらの authfile をボリュームマウントします。

関連情報

- [How to run skopeo in a container](#)

18.4. ホストに対するコンテナイメージのコピー

skopeo、Buildah、Podman は同じローカルコンテナイメージストレージを共有します。ホストコンテナストレージとの間でコンテナをコピーする場合は、これを Skopeo コンテナにマウントする必要があります。



注記

ホストコンテナストレージへのパスは、root (**/var/lib/containers/storage**) と root 以外のユーザー (**\$(HOME)/.local/share/containers/storage**) の間で異なります。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. **registry.access.redhat.com/ubi9/ubi** イメージをローカルストレージにコピーします。

```
$ podman run --privileged --rm -v
$(HOME)/.local/share/containers/storage:/var/lib/containers/storage \
registry.redhat.io/rhel9/skopeo skopeo copy \
docker://registry.access.redhat.com/ubi9/ubi containers-
storage:registry.access.redhat.com/ubi9/ubi
```

- **--privileged** オプションは、すべてのセキュリティーメカニズムを無効にします。Red Hat では、このオプションは信頼できる環境でのみ使用することを推奨します。
- セキュリティーメカニズムを無効にするには、イメージを tarball またはその他のパスベースのイメージトランスポートにエクスポートして Skopeo コンテナにマウントします。
 - **\$ podman save --format oci-archive -o oci.tar \$IMAGE**
 - **\$ podman run --rm -v oci.tar:/oci.tar registry.redhat.io/rhel9/skopeo copy oci-archive:/oci.tar \$DESTINATION**

2. オプション: ローカルストレージ内のイメージをリスト表示する

```
$ podman images
REPOSITORY                                TAG  IMAGE ID  CREATED  SIZE
registry.access.redhat.com/ubi9/ubi        latest ecbc6f53bba0 8 weeks ago 211 MB
```

関連情報

- [How to run skopeo in a container](#)

18.5. コンテナでの BUILDDAH の実行

この手順では、コンテナで Buildah を実行し、イメージを基に作業コンテナを作成する方法を説明します。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. registry.redhat.io レジストリーにログインします。

```
$ podman login registry.redhat.io
Username: myuser@mycompany.com
Password: <password>
Login Succeeded!
```

2. registry.redhat.io/rhel9/buildah イメージをプルして実行します。

```
# podman run --rm --device /dev/fuse -it \
registry.redhat.io/rhel9/buildah /bin/bash
```

- **--rm** オプションは、コンテナの終了後に **registry.redhat.io/rhel9/buildah** イメージを削除します。
- **--device** オプションは、ホストデバイスをコンテナに追加します。
- **sys_chroot** - 別のルートディレクトリーに変更する機能。コンテナのデフォルト機能には含まれていません。

3. registry.access.redhat.com/ubi9 イメージを使用して、新しいコンテナを作成します。

```
# buildah from registry.access.redhat.com/ubi9
...
ubi9-working-container
```

4. ubi9-working-container コンテナ内で ls / コマンドを実行します。

```
# buildah run --isolation=chroot ubi9-working-container ls /
bin boot dev etc home lib lib64 lost+found media mnt opt proc root run sbin srv
```

5. オプション: ローカルストレージ内のイメージをリスト表示します。

```
# buildah images
REPOSITORY          TAG      IMAGE ID      CREATED      SIZE
registry.access.redhat.com/ubi9 latest   ecbc6f53bba0 5 weeks ago 211 MB
```

6. オプション: 作業用のコンテナおよびそれらのベースイメージをリスト表示します。

```
# buildah containers
CONTAINER ID  BUILDER  IMAGE ID      IMAGE NAME          CONTAINER NAME
0aaba7192762  *       ecbc6f53bba0 registry.access.redhat.com/ub... ubi9-working-container
```


- オプション: `registry.access.redhat.com/ubi9` イメージを `registry.example.com` にあるローカルレジストリーにプッシュします。

```
# buildah push ecbc6f53bba0 registry.example.com:5000/ubi9/ubi
```

関連情報

- [Best practices for running Buildah in a container](#)

18.6. 特権および非特権 PODMAN コンテナ

デフォルトでは、Podman コンテナは権限がないため、たとえばホスト上のオペレーティングシステムの一部を変更することはできません。これは、デフォルトでは、コンテナはデバイスへの限定的なアクセスしか許可されないためです。

以下は、特権コンテナの重要なプロパティのリストです。特権コンテナは、`podman run --privileged <image_name>` コマンドを使用して実行できます。

- 特権コンテナには、コンテナを起動するユーザーと同じデバイスへのアクセス権限が付与されます。
- 特権コンテナは、コンテナをホストから分離するセキュリティ機能を無効にします。削除された機能、制限されたデバイス、読み取り専用マウントポイント、Apparmor/SELinux の分離、および Seccomp フィルターは、すべて無効にされます。
- 特権コンテナには、それらを起動したアカウント以上の特権を割り当てることはできません。

関連情報

- [How to use the --privileged flag with container engines](#)
- `podman-run` の man ページ

18.7. 拡張された権限での PODMAN の実行

ルートレス環境でワークロードを実行できない場合は、これらのワークロードを root ユーザーとして実行する必要があります。拡張された権限でのコンテナの実行は、慎重に実施する必要があります。すべてのセキュリティ機能が無効になるためです。

前提条件

- `container-tools` メタパッケージがインストールされている。

手順

- Podman コンテナ内で Podman コンテナを実行します。

```
$ podman run --privileged --name=privileged_podman \
  registry.access.redhat.com//podman podman run ubi9 echo hello
Resolved "ubi9" as an alias (/etc/containers/registries.conf.d/001-rhel-shortnames.conf)
Trying to pull registry.access.redhat.com/ubi9:latest...
```

```
...
Storing signatures
hello
```

- **registry.access.redhat.com/ubi9/podman** イメージに基づいて、**privileged_podman** という名前の外部コンテナを実行します。
- **--privileged** オプションは、コンテナをホストから分離するセキュリティー機能を無効にします。
- **podman run ubi9 echo hello** コマンドを実行して、**ubi9** イメージに基づいて内部コンテナを作成します。
- **ubi9** の短縮イメージ名がエイリアスとして解決されていることに注意してください。これにより、**registry.access.redhat.com/ubi9:latest** イメージがプルされます。

検証

- すべてのコンテナをリスト表示します。

```
$ podman ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS
PORTS NAMES
52537876caf4 registry.access.redhat.com/ubi9/podman podman run ubi9 e... 30
seconds ago Exited (0) 13 seconds ago privileged_podman
```

関連情報

- [How to use Podman inside of a container](#)
- **podman-run** の man ページ

18.8. 少ない権限での PODMAN の実行

--privileged オプションを指定せずに、ネストされた2つの Podman コンテナを実行できます。 **--privileged** オプションを指定せずにコンテナを実行することは、より安全なオプションです。

これは、可能な限り安全な方法で異なるバージョンの Podman を試す場合に役立ちます。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

- ネストされた2つのコンテナを実行します。

```
$ podman run --name=unprivileged_podman --security-opt label=disable \
--user podman --device /dev/fuse \
registry.access.redhat.com/ubi9/podman \
podman run ubi9 echo hello
```

- **registry.access.redhat.com/ubi9/podman** イメージに基づいて、**unprivileged_podman** という名前の外部コンテナを実行します。

- **--security-opt label=disable** オプションは、ホスト Podman での SELinux の分離を無効にします。SELinux では、コンテナ化されたプロセスは、コンテナ内で実行するために必要なすべてのファイルシステムをマウントすることはできません。
- **--user podman** オプションを指定すると、自動的に、外部コンテナ内の Podman がユーザーの名前空間内で実行されるようになります。
- **--device /dev/fuse** オプションは、コンテナ内の **fuse-overlayfs** パッケージを使用します。このオプションは **/dev/fuse** を外部コンテナに追加し、コンテナ内の Podman がこれを使用できるようにします。
- **podman run ubi9 echo hello** コマンドを実行して、**ubi9** イメージに基づいて内部コンテナを作成します。
- **ubi9** の短縮イメージ名がエイリアスとして解決されていることに注意してください。これにより、**registry.access.redhat.com/ubi9:latest** イメージがプルされます。

検証

- すべてのコンテナをリスト表示します。

```
$ podman ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS
PORTS NAMES
a47b26290f43 podman run ubi9 e... 30 seconds ago Exited (0) 13 seconds ago
unprivileged_podman
```

18.9. PODMAN コンテナ内でのコンテナのビルド

Podman を使用してコンテナ内でコンテナを実行できます。この例では、Podman を使用してコンテナから別のコンテナをビルドし、実行する方法を示しています。コンテナは、簡単なテキストベースのゲーム Moon-buggy を実行します。

前提条件

- **container-tools** メタパッケージがインストールされている。
- **registry.redhat.io** レジストリーにログインしている。

```
# podman login registry.redhat.io
```

手順

1. **registry.redhat.io/rhel9/podman** イメージをベースとするコンテナを実行します。

```
# podman run --privileged --name podman_container -it \
registry.redhat.io/rhel9/podman /bin/bash
```

- **registry.redhat.io/rhel9/podman** イメージに基づいて、**podman_container** という名前の外部コンテナを実行します。
- **--it** オプションは、コンテナ内で対話式 bash シェルを実行します。

- **--privileged** オプションは、コンテナをホストから分離するセキュリティー機能を無効にします。

2. **podman_container** コンテナ内に **Containerfile** を作成します。

```
# vi Containerfile
FROM registry.access.redhat.com/ubi9/ubi
RUN dnf install -y https://dl.fedoraproject.org/pub/epel/epel-release-latest-8.noarch.rpm
RUN dnf -y install moon-buggy && dnf clean all
CMD ["/usr/bin/moon-buggy"]
```

Containerfile のコマンドで以下の build コマンドが実行されます。

- **registry.access.redhat.com/ubi9/ubi** イメージからコンテナをビルドします。
 - **epel-release-latest-8.noarch.rpm** パッケージをインストールします。
 - **moon-buggy** パッケージをインストールします。
 - コンテナコマンドを設定します。
3. **Containerfile** を使用して **moon-buggy** という名前の新しいコンテナイメージをビルドします。

```
# podman build -t moon-buggy .
```

4. オプション: すべてのイメージをリスト表示します。

```
# podman images
REPOSITORY          TAG      IMAGE ID      CREATED      SIZE
localhost/moon-buggy latest  c97c58abb564  13 seconds ago  1.67 GB
registry.access.redhat.com/ubi9/ubi latest  4199acc83c6a  132seconds ago  213 MB
```

5. **moon-buggy** コンテナに基づいて新しいコンテナを実行します。

```
# podman run -it --name moon moon-buggy
```

6. オプション: **moon-buggy** イメージにタグを付けます。

```
# podman tag moon-buggy registry.example.com/moon-buggy
```

7. オプション: **moon-buggy** イメージをレジストリーにプッシュします。

```
# podman push registry.example.com/moon-buggy
```

関連情報

- [テクノロジーレビュー:コンテナ内でコンテナの実行](#)

第19章 BUILDDAH でコンテナイメージの構築

Buildah は、[OCI Runtime Specification](#) を満たす OCI コンテナイメージの構築を容易にします。Buildah を使用すると、作業コンテナをゼロから作成することも、イメージを開始点として使用して作成することも可能です。イメージは、**Containerfile** 内の指示を使用して作業コンテナから作成するか、**Containerfile** 内のコマンドをエミュレートする一連の Buildah コマンドを使用して作成できます。

19.1. BUILDDAH ツール

Buildah を使用することは、以下の点で、docker コマンドでイメージを作成するのと異なります。

デーモンなし

Buildah にはコンテナランタイムは必要ありません。

ベースイメージまたは scratch

別のコンテナに基づいてイメージをビルドしたり、空のイメージで始めることができます。

ビルドツールが外部のもの

Buildah では、イメージ自体にはビルドツールが含まれません。その結果、Buildah は以下のようになります。

- ビルドイメージのサイズを縮小します。
- 作成されたイメージからソフトウェア (gcc、make、dnf など) を除外して、イメージのセキュリティを強化します。
- イメージサイズの縮小により、少ないリソースを使用してイメージを転送できます。

互換性

Buildah は、Dockerfile を使用したコンテナイメージの構築をサポートしており、Docker から Buildah への移行を容易にします。



注記

Buildah がコンテナストレージにデフォルトで使用する場所は、CRI-O コンテナエンジンがイメージのローカルコピーを保存するために使用する場所と同じです。そのため、CRI-O または Buildah で、もしくは buildah コマンドでレジストリーから取得したイメージは、同じディレクトリー構造に保存されます。ただし、CRI-O および Buildah が現在イメージを共有できても、コンテナを共有することはできません。

関連情報

- [Buildah - a tool that facilitates building Open Container Initiative \(OCI\) container images](#)
- [Buildah Tutorial 1:Building OCI container images](#)
- [Buildah Tutorial 2:Using Buildah with container registries](#)
- [Buildah を使用した構築:Dockerfile、コマンドライン、またはスクリプト](#)
- [ルートレス Buildah の仕組み:非特権環境でのコンテナの構築](#)

19.2. BUILDDAH のインストール

dnf コマンドを使用して Buildah ツールをインストールします。

手順

- Buildah ツールをインストールします。

```
# dnf -y install buildah
```

検証

- ヘルプメッセージを表示します。

```
# buildah -h
```

19.3. BUILDDAH でイメージの取得

buildah from コマンドを使用して、新規作業コンテナをゼロから作成するか、指定したイメージを開始点として作成します。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

- **registry.redhat.io/ubi9/ubi** イメージをもとに、新しい作業コンテナを作成します。

```
# buildah from registry.access.redhat.com/ubi9/ubi
Getting image source signatures
Copying blob...
Writing manifest to image destination
Storing signatures
ubi-working-container
```

検証

1. ローカルストレージ内のイメージをリスト表示します。

```
# buildah images
REPOSITORY                                TAG    IMAGE ID    CREATED    SIZE
registry.access.redhat.com/ubi9/ubi       latest 272209ff0ae5 2 weeks ago 234 MB
```

2. 作業用のコンテナおよびそれらのベースイメージをリスト表示します。

```
# buildah containers
CONTAINER ID  BUILDER  IMAGE ID    IMAGE NAME                                CONTAINER NAME
01eab9588ae1  *       272209ff0ae5 registry.access.redhat.com/ub... ubi-working-container
```

関連情報

- **buildah-from** の man ページ

- `buildah-images` の man ページ
- `buildah.containers` の man ページ

19.4. BUILDDAH を使用した CONTAINERFILE からのイメージのビルド

`buildah bud` コマンドを使用して、**Containerfile** からのステップを使用してイメージをビルドします。



注記

`buildah bud` コマンドは、コンテキストディレクトリーにある場合に、**Containerfile** を使用します。コンテキストディレクトリーにない場合、`buildah bud` コマンドは **Dockerfile** を使用します。それ以外の場合は、`--file` オプションでファイルを指定できません。**Containerfile** および **Dockerfile** 内で使用できる利用可能なコマンドは同じです。

前提条件

- `container-tools` メタパッケージがインストールされている。

手順

1. **Containerfile** を作成します。

```
# cat Containerfile
FROM registry.access.redhat.com/ubi9/ubi
ADD myecho /usr/local/bin
ENTRYPOINT "/usr/local/bin/myecho"
```

2. **myecho** スクリプトを作成します。

```
# cat myecho
echo "This container works!"
```

3. **myecho** スクリプトのアクセスパーミッションを変更します。

```
# chmod 755 myecho
```

4. 現在のディレクトリーの **Containerfile** を使用して **myecho** イメージをビルドします。

```
# buildah bud -t myecho .
STEP 1: FROM registry.access.redhat.com/ubi9/ubi
STEP 2: ADD myecho /usr/local/bin
STEP 3: ENTRYPOINT "/usr/local/bin/myecho"
STEP 4: COMMIT myecho
...
Storing signatures
```

検証

1. すべてのイメージをリスト表示します。

```
# buildah images
```

```
REPOSITORY          TAG    IMAGE ID    CREATED    SIZE
localhost/myecho    latest b28cd00741b3 About a minute ago 234 MB
```

2. **localhost/myecho** イメージに基づいて **myecho** コンテナを実行します。

```
# podman run --name=myecho localhost/myecho
```

```
This container works!
```

3. すべてのコンテナをリスト表示します。

```
# podman ps -a
```

```
0d97517428d localhost/myecho    12 seconds ago Exited (0) 13
seconds ago    myecho
```



注記

podman history コマンドを使用して、イメージで使用された各レイヤーに関する情報を表示できます。

関連情報

- **buildah-bud** の man ページ

19.5. BUILDDAH でゼロからイメージを新規作成

ベースイメージから開始する代わりに、最低限のコンテナメタデータのみを保持する新しいコンテナを作成できます。

スクラッチコンテナからイメージを作成するときは、次のことを考慮してください。

- **scratch** イメージに依存関係のない実行ファイルをコピーして、**minimal** コンテナを機能させるため、いくつかの設定を行うことができます。
- RPM データベースを初期化し、**dnf** や **rpm** などのツールを使用するために、コンテナにリリースパッケージを追加する必要があります。
- パッケージを多数追加する場合は、**scratch** イメージではなく、標準の **UBI** イメージまたは **UBI minimal** イメージの使用を検討してください。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

Web サービス **httpd** をコンテナに追加し、実行するように設定できます。

1. 空のコンテナを作成します。

```
# buildah from scratch
```

```
working-container
```


2. **working-container** コンテナをマウントし、マウントポイントパスを **scratchmnt** 変数に保存します。

```
# scratchmnt=$(buildah mount working-container)

# echo $scratchmnt
/var/lib/containers/storage/overlay/be2eaecf9f74b6acfe4d0017dd5534fde06b2fa8de9ed875691f6ccc791c1836/merged
```

3. scratch イメージで RPM データベースを初期化し、**redhat-release** パッケージを追加します。

```
# dnf install -y --releasever=8 --installroot=$scratchmnt redhat-release
```

4. httpd サービスを **scratch** ディレクトリーにインストールします。

```
# dnf install -y --setopt=reposdir=/etc/yum.repos.d \
  --installroot=$scratchmnt \
  --setopt=cachedir=/var/cache/dnf httpd
```

5. **\$scratchmnt/var/www/html/index.html** ファイルを作成します。

```
# mkdir -p $scratchmnt/var/www/html
# echo "Your httpd container from scratch works!" >
  $scratchmnt/var/www/html/index.html
```

6. コンテナから直接 **httpd** デーモンを実行するように **working-container** を設定します。

```
# buildah config --cmd "/usr/sbin/httpd -DFOREGROUND" working-container
# buildah config --port 80/tcp working-container
# buildah commit working-container localhost/myhttpd:latest
```

検証

1. ローカルストレージ内のイメージをリスト表示します。

```
# podman images
REPOSITORY          TAG   IMAGE ID   CREATED   SIZE
localhost/myhttpd   latest 08da72792f60 2 minutes ago 121 MB
```

2. **localhost/myhttpd** イメージを実行し、コンテナとホストシステム間のポートマッピングを設定します。

```
# podman run -p 8080:80 -d --name myhttpd 08da72792f60
```

3. Web サーバーをテストします。

```
# curl localhost:8080
Your httpd container from scratch works!
```

関連情報

- **buildah-config** の man ページ
- **buildah-commit** の man ページ

19.6. BUILDDAH でイメージの削除

buildah rmi コマンドを使用して、ローカルに保存されたコンテナイメージを削除します。ID または名前を使用して、イメージを削除できます。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. ローカルシステムにある全イメージのリストを表示します。

```
# 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/ubi9/ubi        latest 272209ff0ae5 3 weeks ago 234 MB
```

2. **localhost/myecho** イメージを削除します。

```
# buildah rmi localhost/myecho
```

- 複数のイメージを削除するには、以下のコマンドを実行します。

```
# buildah rmi docker.io/library/mynewecho docker.io/library/myecho2
```

- システムからすべてのイメージを削除するには、以下のコマンドを実行します。

```
# buildah rmi -a
```

- 複数の名前 (タグ) が関連付けられているイメージを削除するには、**-f** オプションを追加して削除します。

```
# buildah rmi -f localhost/ubi-micro-httpd
```

検証

- イメージが削除されていることを確認します。

```
# buildah images
```

関連情報

- **buildah-rmi** の man ページ

第20章 BUILDDAH を使用したコンテナの操作

Buildah を使用すると、コマンドラインからコンテナイメージまたはコンテナに対していくつかの操作を実行できます。操作の例は次のとおりです。作業コンテナを最初から作成するか開始点としてコンテナイメージから作成する、作業コンテナからまたは **Containerfile** を使用してイメージを作成する、コンテナのエントリーポイント、ラベル、ポート、シェル、作業ディレクトリーを設定する。ファイルシステム操作のために作業コンテナディレクトリーをマウントしたり、作業コンテナやコンテナイメージを削除したりすることができます。

その後、作業コンテナからイメージを作成し、そのイメージをレジストリーにプッシュできます。

20.1. コンテナ内でのコマンドの実行

buildah run コマンドを使用して、コンテナからコマンドを実行します。

前提条件

- **container-tools** メタパッケージがインストールされている。
- プルしたイメージがローカルシステムで利用できる。

手順

- オペレーティングシステムのバージョンを表示します。

```
# buildah run ubi-working-container cat /etc/redhat-release  
Red Hat Enterprise Linux release 8.4 (Ootpa)
```

関連情報

- **buildah-run** の man ページ

20.2. BUILDDAH でコンテナおよびイメージの検証

buildah inspect コマンドを使用して、コンテナまたはイメージに関する情報を表示します。

前提条件

- **container-tools** メタパッケージがインストールされている。
- イメージが Containerfile の命令を使用してビルドされている。詳細は、[Buildah を使用した Containerfile からのイメージのビルド](#)を参照してください。

手順

- イメージを検証します。
 - myecho イメージを検証するには、以下を入力します。

```
# buildah inspect localhost/myecho  
{  
  "Type": "buildah 0.0.1",  
  "FromImage": "localhost/myecho:latest",
```

```

    "FromImageID":
    "b28cd00741b38c92382ee806e1653eae0a56402bcd2c8d31bdcd36521bc267a4",
    "FromImageDigest":
    "sha256:0f5b06cbd51b464fabe93ce4fe852a9038cdd7c7b7661cd7efef8f9ae8a59585",
    "Config":
    ...
    "Entrypoint": [
        "/bin/sh",
        "-c",
        "\"/usr/local/bin/myecho\""
    ],
    ...
}

```

- **myecho** イメージから作業コンテナを検証するには、以下を実行します。
 - i. **localhost/myecho** イメージをもとに作業コンテナを作成します。

```
# buildah from localhost/myecho
```

- ii. **myecho-working-container** コンテナを検査します。

```

# buildah inspect ubi-working-container
{
  "Type": "buildah 0.0.1",
  "FromImage": "registry.access.redhat.com/ubi8/ubi:latest",
  "FromImageID":
  "272209ff0ae5fe54c119b9c32a25887e13625c9035a1599feba654aa7638262d",
  "FromImageDigest":
  "sha256:77623387101abefbf83161c7d5a0378379d0424b2244009282acb39d42f1fe13",
  "Config":
  ...
  "Container": "ubi-working-container",
  "ContainerID":
  "01eab9588ae1523746bb706479063ba103f6281ebaeccc5dc42b70e450d5ad0",
  "ProcessLabel": "system_u:system_r:container_t:s0:c162,c1000",
  "MountLabel": "system_u:object_r:container_file_t:s0:c162,c1000",
  ...
}

```

関連情報

- **buildah-inspect** の man ページ

20.3. BUILDDAH MOUNT を使用したコンテナの変更

buildah mount コマンドを使用して、コンテナまたはイメージに関する情報を表示します。

前提条件

- **container-tools** メタパッケージがインストールされている。

- Containerfile の指示を使用してビルドされたイメージ。詳細は、[Buildah を使用した Containerfile からのイメージのビルド](#)を参照してください。

手順

1. **registry.access.redhat.com/ubi8/ubi** イメージをもとに作業コンテナを作成し、コンテナの名前を **mycontainer** 変数に保存します。

```
# mycontainer=$(buildah from localhost/myecho)

# echo $mycontainer
myecho-working-container
```

2. **myecho-working-container** コンテナをマウントし、マウントポイントパスを **mymount** 変数に保存します。

```
# mymount=$(buildah mount $mycontainer)

# echo $mymount
/var/lib/containers/storage/overlay/c1709df40031dda7c49e93575d9c8eebcaa5d8129033a58e5b6a95019684cc25/merged
```

3. **myecho** スクリプトを変更し、実行可能にします。

```
# echo 'echo "We modified this container."' >> $mymount/usr/local/bin/myecho
# chmod +x $mymount/usr/local/bin/myecho
```

4. **myecho-working-container** コンテナから **myecho2** イメージを作成します。

```
# buildah commit $mycontainer containers-storage:myecho2
```

検証

1. ローカルストレージ内のイメージをリスト表示します。

```
# 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/myecho2** イメージに基づいて **myecho2** コンテナを実行します。

```
# podman run --name=myecho2 docker.io/library/myecho2
This container works!
We even modified it.
```

関連情報

- **buildah-mount** の man ページ
- **buildah-commit** の man ページ

20.4. BUILDAH COPY および BUILDAH CONFIG を使用したコンテナの変更

buildah copy コマンドを使用して、マウントせずにファイルをコンテナにコピーします。次に、**buildah config** コマンドを使用して、デフォルトで作成したスクリプトを実行するコンテナを設定できます。

前提条件

- **container-tools** メタパッケージがインストールされている。
- Containerfile の指示を使用してビルドされたイメージ。詳細は、[Buildah を使用した Containerfile からのイメージのビルド](#)を参照してください。

手順

1. **newecho** という名前のスクリプトを作成し、実行可能にします。

```
# cat newecho
echo "I changed this container"
# chmod 755 newecho
```

2. 新しい作業コンテナを作成します。

```
# buildah from myecho:latest
myecho-working-container-2
```

3. **newecho** スクリプトを、コンテナ内の **/usr/local/bin** ディレクトリーにコピーします。

```
# buildah copy myecho-working-container-2 newecho /usr/local/bin
```

4. **newecho** スクリプトを、新しいエントリーポイントとして使用するよう設定を変更します。

```
# buildah config --entrypoint "/bin/sh -c /usr/local/bin/newecho" myecho-working-
container-2
```

5. オプション: 実行する **newecho** スクリプトのトリガーとなる **myecho-working-container-2** コンテナを実行します。

```
# buildah run myecho-working-container-2 -- sh -c '/usr/local/bin/newecho'
I changed this container
```

6. **myecho-working-container-2** コンテナを、**mynewecho** という新しいイメージにコミットします。

```
# buildah commit myecho-working-container-2 containers-storage:mynewecho
```

検証

- ローカルストレージ内のイメージをリスト表示します。

buildah images

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
docker.io/library/mynewecho	latest	fa2091a7d8b6	8 seconds ago	234 MB

関連情報

- **buildah-copy** の man ページ
- **buildah-config** の man ページ
- **buildah-commit** の man ページ
- **buildah-run** の man ページ

20.5. プライベートレジストリーへのコンテナのプッシュ

buildah push コマンドを使用して、イメージをローカルストレージからパブリックリポジトリーまたはプライベートリポジトリーにプッシュします。

前提条件

- **container-tools** メタパッケージがインストールされている。
- イメージが Containerfile の命令を使用してビルドされている。詳細は、[Buildah を使用した Containerfile からのイメージのビルド](#)を参照してください。

手順

1. マシンにローカルレジストリーを作成します。

```
# podman run -d -p 5000:5000 registry:2
```

2. **myecho:latest** イメージを **localhost** レジストリーにプッシュします。

```
# 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
```

検証

1. **localhost** リポジトリー内のイメージをリスト表示します。

```
# 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** イメージを調査します。

```
# 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** イメージをプルします。

```
# podman pull --tls-verify=false localhost:5000/myecho2
# podman run localhost:5000/myecho2
This container works!
```

関連情報

- **buildah-push** の man ページ

20.6. DOCKER HUB へのコンテナのプッシュ

buildah コマンドで、Docker Hub の認証情報を使用して、Docker Hub からイメージをプッシュおよびプルします。

前提条件

- **container-tools** メタパッケージがインストールされている。
- Containerfile の指示を使用してビルドされたイメージ。詳細は、[Buildah を使用した Containerfile からのイメージのビルド](#)を参照してください。

手順

1. **docker.io/library/myecho:latest** を Docker Hub にプッシュします。 **username** および **password** を、Docker Hub 認証情報に置き換えます。

```
# buildah push --creds username:password \
  docker.io/library/myecho:latest docker://testaccountXX/myecho:latest
```

検証

- **docker.io/testaccountXX/myecho:latest** イメージを取得して実行します。
 - Podman ツールの使用:

```
# podman run docker.io/testaccountXX/myecho:latest
This container works!
```


- Buildah および Podman ツールの使用:

```
# buildah from docker.io/testaccountXX/myecho:latest
myecho2-working-container-2
# podman run myecho-working-container-2
```

関連情報

- **buildah-push** の man ページ

20.7. BUILDDAH でコンテナの削除

buildah rm コマンドを使用して、コンテナを削除します。コンテナ ID または名前で、削除するコンテナを指定できます。

前提条件

- **container-tools** メタパッケージがインストールされている。
- 1つ以上のコンテナが停止されている。

手順

1. すべてのコンテナをリスト表示します。

```
# 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 コンテナを削除します。

```
# buildah rm myecho-working-container
05387e29ab93151cf52e9c85c573f3e8ab64af1592b1ff9315db8a10a77d7c22
```

検証

- コンテナが削除されていることを確認します。

```
# buildah containers
```

関連情報

- **buildah-rm** の man ページ

第21章 コンテナの監視

Podman コマンドを使用して Podman 環境を管理します。これにより、システムおよび Pod 情報を表示し、Podman イベントをモニターすることで、コンテナの正常性を判別できます。

21.1. コンテナでヘルスチェックを使用する

ヘルスチェックを使用して、コンテナ内で実行されているプロセスの状態または準備ができているかどうかを判断できます。

ヘルスチェックが成功すると、コンテナは正常とマークされます。そうでなければ、それは正常ではありません。**podman exec** コマンドを実行して終了コードを調べることで、ヘルスチェックを比較できます。ゼロの終了値は、コンテナが正常であることを意味します。

Containerfile で **HEALTHCHECK** 命令を使用してイメージをビルドするとき、またはコマンドラインでコンテナを作成するときに、ヘルスチェックを設定できます。**podman inspect** または **podman ps** コマンドを使用して、コンテナのヘルスチェックのステータスを表示できます。

ヘルスチェックは、次の 6 つの基本コンポーネントで構成されます。

- コマンド
- Retries
- Interval
- Start-period
- Timeout
- コンテナ回収

ヘルスチェックコンポーネントの説明は次のとおりです。

コマンド (--health-cmd オプション)

Podman は、ターゲットコンテナ内でコマンドを実行して終了コードを待ちます。

他の 5 つのコンポーネントは、ヘルスチェックのスケジューリングに関連しており、オプションです。

再試行 (--health-retries オプション)

正常でないでマークするまでに、ヘルスチェックで連続して不合格とならなければならないか、その回数を定義します。Healthcheck に成功すると、再試行数がリセットされます。

間隔 (--health-interval オプション)

次に healthcheck コマンドを実行するまでの時間を指定します。間隔が短いと、システムでの healthcheck の実行時間が長くなる点に注意してください。間隔が長いと、タイムアウトの検出が困難になります。

開始期間 (--health-start-period オプション)

コンテナを起動してから healthcheck の不合格を無視するまでの時間を指定します。

タイムアウト (--health-timeout オプション)

healthcheck の完了前に不合格とみなされるまでの期間を指定します。



注記

Retries、Interval、および Start-period コンポーネントの値は、30s や 1h15m などの期間です。有効な時間単位は、ns、us、または μ s、ms、s、m、および h です。

コンテナの復旧 (--health-on-failure オプション)

コンテナのステータスが異常な場合に実行するアクションを決定します。アプリケーションに障害が発生すると、Podman は自動的にアプリケーションを再起動して堅牢性を提供します。--health-on-failure オプションは、次の 4 つのアクションをサポートします。

- **none**: 何もしないでください。これがデフォルトのアクションです。
- **kill**: コンテナを強制終了します。
- **restart**: コンテナを再起動します。
- **stop**: コンテナを停止します。



注記

--health-on-failure オプションは、Podman バージョン 4.2 以降で使用できません。



警告

restart アクションを --restart オプションと組み合わせないでください。systemd ユニット内で実行する場合は、代わりに **kill** または **stop** アクションを使用して、systemd 再起動ポリシーを利用することを検討してください。

コンテナ内でヘルスチェックが実行されます。ヘルスチェックは、サービスのヘルス状態がどのようなものを把握しており、ヘルスチェックの成功と失敗を区別できる場合にのみ意味があります。

関連情報

- [podman-healthcheck man ページ](#)
- [podman-run の man ページ](#)
- [Podman at the edge: Keeping services alive with custom healthcheck actions](#)
- [Monitoring container vitality and availability with Podman](#)

21.2. コマンドラインを使用してヘルスチェックを実行する

コマンドラインでコンテナを作成するときに、ヘルスチェックを設定できます。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. ヘルスチェックを定義します。

```
$ podman run -dt --name=hc-container -p 8080:8080 --health-cmd='curl
http://localhost:8080 || exit 1' --health-interval=0
registry.access.redhat.com/ubi8/httpd-24
```

- **--health-cmd** オプションは、コンテナの healthcheck コマンドを設定します。
 - healthcheck を手動で実行するには、**--health-interval=0** オプションで0の値を指定します。
2. **hc-container** コンテナのヘルスステータスを確認します。

- **podman inspect** コマンドの使用:

```
$ podman inspect --format='{{json .State.Health.Status}}' hc-container
healthy
```

- **podman ps** コマンドの使用:

```
$ podman ps
CONTAINER ID IMAGE          COMMAND                                CREATED   STATUS
PORTS      NAMES
a680c6919fe localhost/hc-container:latest /usr/bin/run-http... 2 minutes ago Up 2
minutes (healthy) hc-container
```

- **podman healthcheck run** コマンドを使用します。

```
$ podman healthcheck run hc-container
healthy
```

関連情報

- **podman-healthcheck** man ページ
- **podman-run** の man ページ
- [Podman at the edge: Keeping services alive with custom healthcheck actions](#)
- [Monitoring container vitality and availability with Podman](#)

21.3. CONTAINERFILE を使用してヘルスチェックを実行する

Containerfile で **HEALTHCHECK** 命令を使用してヘルスチェックを設定できます。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. **Containerfile** を作成します。

```
$ cat Containerfile
FROM registry.access.redhat.com/ubi8/httpd-24
EXPOSE 8080
HEALTHCHECK CMD curl http://localhost:8080 || exit 1
```



注記

HEALTHCHECK 命令は、**docker** イメージ形式でのみサポートされています。**oci** イメージ形式の場合、命令は無視されます。

2. コンテナをビルドし、イメージ名を追加します。

```
$ podman build --format=docker -t hc-container .
STEP 1/3: FROM registry.access.redhat.com/ubi8/httpd-24
STEP 2/3: EXPOSE 8080
--> 5aea97430fd
STEP 3/3: HEALTHCHECK CMD curl http://localhost:8080 || exit 1
COMMIT health-check
Successfully tagged localhost/health-check:latest
a680c6919fe6bf1a79219a1b3d6216550d5a8f83570c36d0dadfee1bb74b924e
```

3. コンテナを実行します。

```
$ podman run -dt --name=hc-container localhost/hc-container
```

4. **hc-container** コンテナのヘルスステータスを確認します。

- **podman inspect** コマンドの使用:

```
$ podman inspect --format='{{json .State.Health.Status}}' hc-container
healthy
```

- **podman ps** コマンドの使用:

```
$ podman ps
CONTAINER ID IMAGE COMMAND CREATED STATUS
PORTS NAMES
a680c6919fe localhost/hc-container:latest /usr/bin/run-http... 2 minutes ago Up 2
minutes (healthy) hc-container
```

- **podman healthcheck run** コマンドを使用します。

```
$ podman healthcheck run hc-container
healthy
```

関連情報

- **podman-healthcheck** man ページ

- [podman-run の man ページ](#)
- [Podman at the edge: Keeping services alive with custom healthcheck actions](#)
- [Monitoring container vitality and availability with Podman](#)

21.4. PODMAN システム情報の表示

podman system コマンドを使用すると、システム情報を表示することで Podman システムを管理できます。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

- Podman システム情報を表示します。
 - Podman のディスク使用量を表示するには、以下を入力します。

```
$ 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%)
```

- 領域の使用状況に関する詳細情報を表示するには、次のコマンドを実行します。

```
$ podman system df -v
Images space usage:

REPOSITORY          TAG    IMAGE ID    CREATED    SIZE
SHARED SIZE UNIQUE SIZE CONTAINERS
registry.access.redhat.com/ubi9    latest    b1e63aaae5cf    13 days    233.4MB
233.4MB  0B      0
registry.access.redhat.com/ubi9/httpd-24    latest    0d04740850e8    13 days    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 ubi9 echo hello 0      0B      11 hours
configured dazzling_tu

Local Volumes space usage:

VOLUME NAME                                LINKS    SIZE
76de0efa83a3dae1a388b9e9e67161d28187e093955df185ea228ad0b3e435d0 0
0B
```

```
8a1b4658aecc9ff38711a2c7f2da6de192c5b1e753bb7e3b25e9bf3bb7da8b13 0
0B
d9cab4f6ccbcbf2ac3cd750d2efff9d2b0f29411d430a119210dd242e8be20e26 0      0B
```

- Podman のホスト、現在のストレージ統計、およびビルドについての情報を表示するには、以下を入力します。

```
$ 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,C
AP_KILL,CAP_NET_BIND_SERVICE,CAP_SETFCAP,CAP_SETGID,CAP_SETPCAP,CA
P_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

```

- 未使用のコンテナ、イメージ、およびボリュームデータをすべて削除するには、次のコマンドを実行します。

```

$ 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** コマンドは、未使用のコンテナ (関連付けられていないコンテナおよび参照されていないコンテナ両方)、Pod、ローカルストレージのボリューム (任意) をすべて削除します。
- **--all** オプションを使用して、未使用のイメージをすべて削除します。未使用のイメージとは、ベースとするコンテナのないイメージや、関連付けられていないイメージを指します。
- ボリュームをプルーニングするには、**--volume** オプションを使用します。デフォルトでは、現在ボリュームを使用するコンテナがない場合に、重要なデータが削除されないようにボリュームは削除されません。

関連情報

- **podman-system-df** の man ページ
- **podman-system-info** の man ページ
- **podman-system-prune** の man ページ

21.5. PODMAN イベントタイプ

Podman で発生するイベントをモニターできます。イベントタイプが複数存在し、イベントタイプごとに異なるステータスを報告します。

コンテナ イベントタイプは以下のステータスを報告します。

- attach
- checkpoint
- cleanup
- commit
- create

- exec
- export
- import
- init
- kill
- mount
- pause
- prune
- remove
- restart
- restore
- start
- stop
- sync
- unmount
- unpause

Pod イベントタイプは以下のステータスを報告します。

- create
- kill
- pause
- remove
- start
- stop
- unpause

image イベントタイプは以下のステータスを報告します。

- prune
- push
- pull
- save

- remove
- tag
- untag

system タイプは、以下のステータスを報告します。

- refresh
- renumber

volume タイプは、以下のステータスを報告します。

- create
- prune
- remove

関連情報

- **podman-events** の man ページ

21.6. PODMAN イベントのモニタリング

podman events コマンドを使用して、Podman で発生するイベントを監視および印刷できます。各イベントには、タイムスタンプ、タイプ、ステータス、名前 (該当する場合)、およびイメージ (該当する場合) が含まれます。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. **myubi** コンテナを実行します。

```
$ podman run -q --rm --name=myubi registry.access.redhat.com/ubi8/ubi:latest
```

2. Podman イベントを表示します。

- すべての Podman イベントを表示するには、次のように入力します。

```
$ now=$(date --iso-8601=seconds)
$ podman events --since=now --stream=false
2023-03-08 14:27:20.696167362 +0100 CET container create
d4748226a2bcd271b1bc4b9f88b54e8271c13ffea9b30529968291c62d72fe09
(image=registry.access.redhat.com/ubi8/ubi:latest, name=myubi,...)
2023-03-08 14:27:20.652325082 +0100 CET image pull
registry.access.redhat.com/ubi8/ubi:latest
2023-03-08 14:27:20.795695396 +0100 CET container init
d4748226a2bcd271b1bc4b9f88b54e8271c13ffea9b30529968291c62d72fe09
(image=registry.access.redhat.com/ubi8/ubi:latest, name=myubi...)
2023-03-08 14:27:20.809205161 +0100 CET container start
```

```
d4748226a2bcd271b1bc4b9f88b54e8271c13ffea9b30529968291c62d72fe09
(image=registry.access.redhat.com/ubi8/ubi:latest, name=myubi...)
2023-03-08 14:27:20.809903022 +0100 CET container attach
d4748226a2bcd271b1bc4b9f88b54e8271c13ffea9b30529968291c62d72fe09
(image=registry.access.redhat.com/ubi8/ubi:latest, name=myubi...)
2023-03-08 14:27:20.831710446 +0100 CET container died
d4748226a2bcd271b1bc4b9f88b54e8271c13ffea9b30529968291c62d72fe09
(image=registry.access.redhat.com/ubi8/ubi:latest, name=myubi...)
2023-03-08 14:27:20.913786892 +0100 CET container remove
d4748226a2bcd271b1bc4b9f88b54e8271c13ffea9b30529968291c62d72fe09
(image=registry.access.redhat.com/ubi8/ubi:latest, name=myubi...)
```

--stream=false オプションを使用すると、最後の既知のイベントを読み取るときに **podman events** コマンドが確実に終了します。

podman run コマンドを入力すると、発生したいくつかのイベントが表示されます。

- 新しいコンテナを作成する際は、**container create**。
 - コンテナイメージがローカルストレージに存在しない場合、イメージをプルする際は、**image pull**。
 - ランタイムでコンテナを初期化し、ネットワークを設定する際は、**container init**。
 - コンテナを起動する際は、**container start**。
 - コンテナのターミナルにアタッチする際は、**container attach**。これは、コンテナがフォアグラウンドで実行されるためです。
 - コンテナが終了すると、**container died** が発行されます。
 - **--rm** フラグは、コンテナの終了後、コンテナの削除に使用されたため、**container remove**。
- また、**journalctl** コマンドを使用して Podman イベントを表示することもできます。

```
$ journalctl --user -r SYSLOG_IDENTIFIER=podman
Mar 08 14:27:20 fedora podman[129324]: 2023-03-08 14:27:20.913786892 +0100 CET
m=+0.066920979 container remove
...
Mar 08 14:27:20 fedora podman[129289]: 2023-03-08 14:27:20.696167362 +0100 CET
m=+0.079089208 container create
d4748226a2bcd271b1bc4b9f88b54e8271c13ffea9b30529968291c62d72f>
```

- Podman の作成イベントのみを表示するには、以下を入力します。

```
$ podman events --filter event=create
2023-03-08 14:27:20.696167362 +0100 CET container create
d4748226a2bcd271b1bc4b9f88b54e8271c13ffea9b30529968291c62d72fe09
(image=registry.access.redhat.com/ubi8/ubi:latest, name=myubi,...)
```

- また、**journalctl** コマンドを使用して Podman 作成イベントを表示することもできます。

```
$ journalctl --user -r PODMAN_EVENT=create
Mar 08 14:27:20 fedora podman[129289]: 2023-03-08 14:27:20.696167362 +0100 CET
m=+0.079089208 container create
```

```
d4748226a2bcd271b1bc4b9f88b54e8271c13ffea9b30529968291c62d72f>
```

関連情報

- `podman-events` の man ページ
- [コンテナイベントと監査](#)

21.7. PODMAN イベントを使用した監査

以前は、イベントを正しく解釈するには、イベントをイベントに接続する必要がありました。たとえば、どのイメージが使用されたかを知るには、`container-create` イベントを `image-pull` イベントにリンクする必要がありました。また、`container-create` イベントには、セキュリティー設定、ボリューム、マウントなどのすべてのデータが含まれているわけではありません。

Podman v4.4 以降、コンテナに関するすべての関連情報を1つのイベントと `journald` エントリーから直接収集できるようになりました。データは `podman container inspect` コマンドからのものと同じ JSON 形式であり、コンテナのすべての設定とセキュリティー設定が含まれます。監査目的でコンテナ検査データを添付するように Podman を設定できます。

前提条件

- `container-tools` メタパッケージがインストールされている。

手順

1. `~/.config/containers/containers.conf` ファイルを変更し、`events_container_create_inspect_data=true` オプションを `[engine]` セクションに追加します。

```
$ cat ~/.config/containers/containers.conf
[engine]
events_container_create_inspect_data=true
```

システム全体の設定の場合は、`/etc/containers/containers.conf` または `/usr/share/container/containers.conf` ファイルを変更します。

2. コンテナを作成します。

```
$ podman create registry.access.redhat.com/ubi8/ubi:latest
19524fe3c145df32d4f0c9af83e7964e4fb79fc4c397c514192d9d7620a36cd3
```

3. Podman イベントを表示します。

- `podman events` コマンドを使用します。

```
$ now=$(date --iso-8601=seconds)
$ podman events --since $now --stream=false --format "{{.ContainerInspectData}}"
| jq ".Config.CreateCommand"
[
  "/usr/bin/podman",
  "create",
  "registry.access.redhat.com/ubi8"
]
```

- `--format "{{.ContainerInspectData}}"` オプションは検査データを表示します。
- `jq ".Config.CreateCommand"` は、JSON データをより読みやすい形式に変換し、`podman create` コマンドのパラメーターを表示します。
- `journalctl` コマンドを使用します。

```
$ journalctl --user -r PODMAN_EVENT=create --all -o json | jq  
".PODMAN_CONTAINER_INSPECT_DATA | fromjson" | jq  
".Config.CreateCommand"  
[  
  "/usr/bin/podman",  
  "create",  
  "registry.access.redhat.com/ubi8"  
]
```

`podman events` と `journalctl` コマンドの出力データは同じです。

関連情報

- `podman-events` の man ページ
- `containers.conf` の man ページ
- [コンテナイベントと監査](#)

第22章 コンテナチェックポイントの作成および復元

CRIU (Checkpoint/Restore In Userspace) は、実行中のコンテナまたは個々のアプリケーションでチェックポイントを設定して、その状態をディスクに保存するソフトウェアです。保存したデータを使用して、再起動後に、チェックポイントの時点にコンテナを復元できます。



警告

カーネルは、AArch64 でのコピー前のチェックポイント設定をサポートしていません。

22.1. ローカルでのコンテナチェックポイントの作成および復元

以下の例は、要求ごとにインクリメントする整数を1つ返す Python ベースの Web サーバーをベースとしています。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. Python ベースのサーバーを作成します。

```
# 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. 以下の定義でコンテナを作成します。

```
# cat Containerfile
FROM registry.access.redhat.com/ubi9/ubi
```

```
COPY counter.py /home/counter.py

RUN useradd -ms /bin/bash counter

RUN dnf -y install python3 && chmod 755 /home/counter.py

USER counter
ENTRYPOINT /home/counter.py
```

コンテナは Universal Base Image (UBI 8) をベースとしており、Python ベースのサーバーを使用します。

3. コンテナをビルドします。

```
# podman build . --tag counter
```

counter.py ファイルおよび **Containerfile** ファイルは、コンテナビルドプロセス (**podman build**) の入力情報です。ビルドされたイメージはローカルに保存され、**counter** でタグ付けされます。

4. root でコンテナを起動します。

```
# podman run --name criu-test --detach counter
```

5. 実行中のコンテナのリストを表示するには、次のコマンドを実行します。

```
# podman ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
e4f82fd84d48 localhost/counter:latest 5 seconds ago Up 4 seconds ago criu-test
```

6. コンテナの IP アドレスを表示します。

```
# podman inspect criu-test --format "{{.NetworkSettings.IPAddress}}"
10.88.0.247
```

7. 要求をコンテナに送信します。

```
# curl 10.88.0.247:8088
0
# curl 10.88.0.247:8088
1
```

8. コンテナのチェックポイントを作成します。

```
# podman container checkpoint criu-test
```

9. システムを再起動します。

10. コンテナを復元します。

```
# podman container restore --keep criu-test
```

11. 要求をコンテナに送信します。


```
# curl 10.88.0.247:8080
2
# curl 10.88.0.247:8080
3
# curl 10.88.0.247:8080
4
```

結果は **0** で開始されるのではなく、以前の値で継続されます。

これにより、再起動後に完全なコンテナの状態を簡単に保存できます。

関連情報

- [Podman checkpoint](#)

22.2. コンテナ復元を使用した起動時間の短縮

コンテナマイグレーションを使用して、初期化に特定の時間を必要とするコンテナの起動時間を短縮できます。チェックポイントを使用すると、同じホストまたは別のホストでコンテナを複数回復元できます。この例は、[コンテナチェックポイントのローカルでの作成および復元](#)のコンテナを基にしています。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. コンテナのチェックポイントを作成し、チェックポイントイメージを **tar.gz** ファイルにエクスポートします。

```
# podman container checkpoint criu-test --export /tmp/chkpt.tar.gz
```

2. **tar.gz** ファイルからコンテナを復元します。

```
# 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) オプションは、エクスポートしたチェックポイントを基に復元したコンテナに新しい名前を指定します。

3. 各コンテナの ID と名前を表示します。

```
# podman ps -a --format "{{.ID}} {{.Names}}"
a8b2e50d463c counter3
faabc5c27362 counter2
2ce648af11e5 counter1
```

4. 各コンテナの IP アドレスを表示します。

```
# 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. 各コンテナに要求を送信します。

```
# curl 10.88.0.248:8080
4
# curl 10.88.0.249:8080
4
# curl 10.88.0.250:8080
4
```

同じチェックポイントから異なるコンテナを使用して復元しているため、結果は、どの場合も **4** になる点に注意してください。

この方法では、最初にチェックポイントを作成したコンテナのステートフルレプリカを迅速に起動できます。

関連情報

- [Container migration with Podman on RHEL](#)

22.3. システム間のコンテナの移行

コンテナで実行しているアプリケーションの状態を失うことなく、実行中のコンテナを別のシステムに移行できます。この例は、**counter** とタグ付けされた [コンテナのチェックポイントの作成および復元のコンテナ](#) セクションのコンテナを基にしています。

重要

podman container checkpoint コマンドと **podman container restore** コマンドを使用したシステム間でのコンテナの移行は、以下に示すように、システムの設定が完全に一致する場合にのみサポートされます。

- Podman のバージョン
- OCI ランタイム (runc/crun)
- ネットワークスタック (CNI/Netavark)
- Cgroups のバージョン
- カーネルのバージョン
- CPU の機能

より多くの機能を備えた CPU に移行することはできますが、使用している特定の機能を備えていない CPU には移行できません。チェックポイントを実行する低レベルツール (CRIU) には、CPU 機能の互換性をチェックする機能があります (<https://criu.org/Cpuinfo>)。

前提条件

- **container-tools** メタパッケージがインストールされている。
- 以下の手順は、コンテナがレジストリーにプッシュされている場合には不要です。理由は、Podman がローカルでコンテナを利用できない場合に自動的にコンテナをレジストリーからダウンロードするためです。この例ではレジストリーを使用しません。以前に構築されタグ付けされたコンテナをエクスポートする必要があります ([ローカルでのコンテナチェックポイントの作成と復元](#) を参照)。
 - 以前にビルドしたコンテナをエクスポートします。

```
# podman save --output counter.tar counter
```

- エクスポートしたコンテナイメージを移行先システム (**other_host**) にコピーします。

```
# scp counter.tar other_host:
```

- エクスポートしたコンテナを移行先システムにインポートします。

```
# ssh other_host podman load --input counter.tar
```

このコンテナの移行先のシステムには、ローカルコンテナストレージに保存されているのと同じコンテナイメージがあります。

手順

1. root でコンテナを起動します。

```
# podman run --name criu-test --detach counter
```

2. コンテナの IP アドレスを表示します。

```
# podman inspect criu-test --format "{{.NetworkSettings.IPAddress}}"  
10.88.0.247
```

3. 要求をコンテナに送信します。

```
# curl 10.88.0.247:8080  
0  
# curl 10.88.0.247:8080  
1
```

4. コンテナのチェックポイントを作成し、チェックポイントイメージを **tar.gz** ファイルにエクスポートします。

```
# podman container checkpoint criu-test --export /tmp/chkpt.tar.gz
```

5. チェックポイントアーカイブを移行先ホストにコピーします。

```
# scp /tmp/chkpt.tar.gz other_host:/tmp/
```

6. 移行先ホスト (**other_host**) のチェックポイントを復元します。

```
# podman container restore --import /tmp/chkpt.tar.gz
```

7. 宛先ホスト (**other_host**) のコンテナに要求を送信します。

```
# *curl 10.88.0.247:8080*  
2
```

これで、ステートフルコンテナが、状態を失うことなく、別のシステムへ移行されました。

関連情報

- [Container migration with Podman on RHEL](#)

第23章 開発とトラブルシューティングに TOOLBX を使用する

システムにソフトウェアをインストールすると、システムの動作が変更したり、不要になったファイルやディレクトリが残されたりするなど、一定のリスクが生じます。ベースオペレーティングシステムに影響を与えることなく、お気に入りの開発およびデバッグツール、エディター、ソフトウェア開発キット (SDK) を、完全に変更可能な Toolbox コンテナにインストールすることで、これらのリスクを回避できます。**less**、**lsuf**、**rsync**、**ssh**、**sudo**、**unzip** などのコマンドを使用して、ホストシステム上で変更を実行できます。

Toolbox ユーティリティーは次のアクションを実行します。

1. **registry.access.redhat.com/ubi9/toolbox:latest** イメージをローカルシステムにプルする
2. イメージからコンテナを起動する
3. コンテナ内でシェルを実行し、そこからホストシステムにアクセスできる



注記

Toolbox は、Toolbox コンテナを作成したユーザーの権限に応じて、ルートコンテナまたはルートレスコンテナを実行できます。ホストシステムでルート権限を必要とするユーティリティーも、ルートコンテナで実行する必要があります。

デフォルトのコンテナ名は **rhel-toolbox** です。

23.1. TOOLBX コンテナの起動

toolbox create コマンドを使用して Toolbox コンテナを作成できます。その後、**toolbox enter** コマンドを使用してコンテナに入ることができます。

手順

1. Toolbox コンテナを作成します。

- ルートレスユーザーとして:

```
$ toolbox create <mytoolbox>
```

- ルートユーザーとして:

```
$ sudo toolbox create <mytoolbox>
Created container: <mytoolbox>
Enter with: toolbox enter
```

- 正しいイメージを取得したことを確認します。

```
[user@toolbox ~]$ toolbox list
IMAGE ID  IMAGE NAME  CREATED
fe0ae375f149  registry.access.redhat.com/ubi{ProductVersion}/toolbox 5 weeks ago

CONTAINER ID  CONTAINER NAME  CREATED  STATUS  IMAGE NAME
5245b924c2cb  <mytoolbox>    7 minutes ago  created
registry.access.redhat.com/ubi{ProductVersion}/toolbox:8.9-6
```

2. Toolbox コンテナに入ります。

```
[user@toolbox ~]$ toolbox enter <mytoolbox>
```

検証

- **<mytoolbox>** コンテナ内でコマンドを実行し、コンテナの名前とイメージを表示します。

```
● [user@toolbox ~]$ cat /run/.containerenv
engine="podman-4.8.2"
name="<mytoolbox>"
id="5245b924c2cb..."
image="registry.access.redhat.com/ubi{ProductVersion}/toolbox"
imageid="fe0ae375f14919cbc0596142e3aff22a70973a36e5a165c75a86ea7ec5d8d65c"
```

23.2. 開発に TOOLBX を使用する

エディター、コンパイラー、ソフトウェア開発キット (SDK) などの開発ツールをインストールするために、Toolbox コンテナをルートレスユーザーとして使用できます。インストール後、ルートレスユーザーとしてこれらのツールを引き続き使用できます。

前提条件

- Toolbox コンテナを作成し、実行している。Toolbox コンテナに入っている。Toolbox コンテナをルート権限で作成する必要はありません。[Toolbox コンテナの起動](#) を参照してください。

手順

- Emacs テキストエディター、GCC コンパイラー、GNU デバッガー (GDB) など、選択したツールをインストールします。

```
● [user@toolbox ~]$ sudo dnf install emacs gcc gdb
```

検証

- ツールがインストールされていることを確認します。

```
● [user@toolbox ~]$ dnf repoquery --info --installed <package_name>
```

23.3. ホストシステムのトラブルシューティングに TOOLBX を使用する

ルート権限を持つ Toolbox コンテナを使用すると、**systemd**、**journalctl**、**nmap** などのツールをホストシステムにインストールせずに使用して、ホストシステムのさまざまな問題の根本原因を見つけることができます。Toolbox コンテナ内では、たとえば次のアクションを実行できます。

前提条件

- Toolbox コンテナを作成し、実行している。Toolbox コンテナに入っている。Toolbox コンテナはルート権限で作成する必要があります。[Toolbox コンテナの起動](#) を参照してください。

手順

1. **journalctl** コマンドを実行できるようにするには、**systemd** スイートをインストールします。

```
●[root@toolbox ~]# dnf install systemd
```

2. ホスト上で実行しているすべてのプロセスのログメッセージを表示します。

```
●[root@toolbox ~]# j journalctl --boot -0
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: microcode: updated ear>
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: Linux version 6.6.8-10>
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: Command line: BOOT_IMA>
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: x86/split lock detecti>
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: BIOS-provided physical>
```

3. カーネルのログメッセージを表示します。

```
●[root@toolbox ~]# journalctl --boot -0 --dmesg
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: microcode: updated ear>
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: Linux version 6.6.8-10>
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: Command line: BOOT_IMA>
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: x86/split lock detecti>
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: BIOS-provided physical>
Jan 02 09:06:48 user-thinkpadp1gen4i.brq.csb kernel: BIOS-e820: [mem 0x0000>
```

4. **nmap** ネットワークスキャンツールをインストールします。

```
●[root@toolbox ~]# dnf install nmap
```

5. ネットワーク内の IP アドレスとポートをスキャンします。

```
●[root@toolbox ~]# nmap -sS scanme.nmap.org
Starting Nmap 7.93 ( https://nmap.org ) at 2024-01-02 10:39 CET
Stats: 0:01:01 elapsed; 0 hosts completed (0 up), 256 undergoing Ping Scan
Ping Scan Timing: About 29.79% done; ETC: 10:43 (0:02:24 remaining)
Nmap done: 256 IP addresses (0 hosts up) scanned in 206.45 seconds
```

- **-sS** オプションは TCP SYN スキャンを実行します。Nmap のスキャンタイプのほとんどは、生のパケットを送受信するため、特権ユーザーのみが使用できます。UNIX システムではルートアクセスが必要です。

23.4. TOOLBX コンテナの停止

Toolbox コンテナを終了するには **exit** コマンドを使用し、コンテナを停止するには **podman stop** コマンドを使用します。

手順

1. コンテナを離れてホストに戻ります。

```
● [user@toolbox ~]$ exit
```

2. ツールボックスコンテナを停止します。

```
● [user@toolbox ~]$ podman stop <mytoolbox>
```

3. オプション: ツールボックスコンテナを削除します。

```
● [user@toolbox ~]$ toolbox rm <mytoolbox>
```

あるいは、**podman rm** コマンドを使用してコンテナを削除することもできます。

第24章 HPC 環境での PODMAN の使用

Podman を Open MPI (Message Passing Interface) と共に使用して、HPC (High Performance Computing) 環境でコンテナを実行できます。

24.1. PODMAN と MPI の使用

この例は、Open MPI から取得した `ring.c` プログラムを基にしています。この例では、全プロセスで値はリングのように渡されます。メッセージがランク 0 を渡すと必ず、値は1つ下がります。各プロセスが 0 メッセージを受信すると、次のプロセスに渡して終了します。0 を最初に渡すと、すべてのプロセスが 0 メッセージを取得し、通常通りに終了できます。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. Open MPI をインストールします。

```
# dnf install openmpi
```

2. 環境モジュールをアクティベートするには、以下を入力します。

```
$. /etc/profile.d/modules.sh
```

3. **mpi/openmpi-x86_64** モジュールを読み込みます。

```
$ module load mpi/openmpi-x86_64
```

必要に応じて、**mpi/openmpi-x86_64** モジュールを自動的に読み込むには、以下の行を **.bashrc** ファイルに追加します。

```
$ echo "module load mpi/openmpi-x86_64" >> .bashrc
```

4. **mpirun** と **podman** を統合するには、以下の定義でコンテナを作成します。

```
$ cat Containerfile
FROM registry.access.redhat.com/ubi9/ubi

RUN dnf -y install openmpi-devel wget && \
    dnf 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. コンテナをビルドします。

```
$ podman build --tag=mpi-ring .
```

6. コンテナを起動します。このコマンドは、CPU が 4 つあるシステムではコンテナを 4 つ起動します。

```
$ mpirun \
  --mca orte_tmpdir_base /tmp/podman-mpirun \
  podman run --env-host \
  -v /tmp/podman-mpirun:/tmp/podman-mpirun \
  --users=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
```

これにより、**mpirun** は 4 つの Podman コンテナを開始し、コンテナごとに **ring** バイナリーのインスタンスを 1 台実行します。4 つプロセスはすべて、MPI 経由で相互に通信しています。

関連情報

- [Podman in HPC environments](#)

24.2. MPIRUN オプション

以下の **mpirun** オプションは、コンテナの起動に使用します。

- **--mca orte_tmpdir_base /tmp/podman-mpirun** 行は、Open MPI に対し、**/tmp** ではなく、**/tmp/podman-mpirun** にその一時ファイルをすべて作成するように指示します。複数のノードを使用する場合には、別のノードではこのディレクトリーの名前は異なります。このような場合には、**/tmp** ディレクトリー全体をコンテナにマウントする必要があり、操作がより複雑です。

mpirun コマンドは、**podman** コマンドを起動するコマンドを指定します。以下の **podman** オプションは、コンテナの起動に使用されます。

- **run** コマンドはコンテナを実行します。
- **--env-host** オプションは、ホストからコンテナにすべての環境変数をコピーします。
- **-v /tmp/podman-mpirun:/tmp/podman-mpirun** は、Open MPI が一時ディレクトリーとファイルをコンテナで利用できるように、Podman にディレクトリーのマウントを指示します。
- **--users=keep-id** 行を使用すると、コンテナ内外でのユーザー ID マッピングを保証します。
- **--net=host --pid=host --ipc=host** 行では、同じネットワーク、PID、および IPC 名前空間が設定されます。

- **mpi-ring** はコンテナの名前です。
- **/home/ring** は、コンテナ内の MPI プログラムです。

関連情報

- [Podman in HPC environments](#)

第25章 特殊なコンテナイメージの実行

特殊なタイプのコンテナイメージを実行できます。一部のコンテナイメージには、事前に設定されたオプションおよび引数を使用してコンテナを実行できる `runlabels` と呼ばれる組み込みラベルがあります。`podman container runlabel <label>` コマンドでは、コンテナイメージの `<label>` で定義されたコマンドを実行できます。サポートされるラベルは、`install`、`run`、および `uninstall` です。

25.1. ホストへの権限の付与

特権コンテナと非特権コンテナにはいくつかの違いがあります。たとえば、`toolbox` コンテナは特権コンテナです。以下は、コンテナがホストに対して許可する、または許可しない権限の例です。

- **権限:**特権コンテナは、コンテナをホストから分離するセキュリティー機能を無効にします。特権コンテナは、`podman run --privileged <image_name>` コマンドを使用して実行できます。たとえば、`root` ユーザーが所有するホストからマウントされたファイルおよびディレクトリーを削除できます。
- **プロセステーブル:**`podman run --privileged --pid=host <image_name>` コマンドを使用して、コンテナでホストの PID 名前空間を使用できます。特権コンテナ内で `ps -e` コマンドを使用して、ホストで実行しているすべてのプロセスをリスト表示できます。ホストから特権コンテナで実行するコマンドにプロセス ID を渡すことができます (例: `kill <PID>`)。
- **ネットワークインターフェイス:**デフォルトでは、コンテナには外部のネットワークインターフェイスとループバックネットワークインターフェイスが1つずつだけあります。`podman run --net=host <image_name>` コマンドを使用して、コンテナ内からホストのネットワークインターフェイスに直接アクセスできます。
- **プロセス間の通信:**ホストの IPC 機能は、特権コンテナからアクセスできます。`ipcs` などのコマンドを実行して、ホスト上のアクティブなメッセージキュー、共有メモリーセグメント、およびセマフォセットに関する情報を表示できます。

25.2. RUNLABEL が組み込まれたコンテナイメージ

Red Hat イメージには、そのイメージと連携するために事前に設定されたコマンドラインを提供するラベルが含まれているものがあります。`podman container runlabel <label>` コマンドを使用すると、`podman` コマンドを使用してイメージの `<label>` で定義されたコマンドを実行できます。

既存の `runlabel` には次のものが含まれます。

- **install:**イメージを実行する前にホストシステムを設定します。通常、これにより、後で実行するときにコンテナがアクセスできるホストにファイルとディレクトリーが作成されます。
- **run:**コンテナの実行時に使用する `podman` コマンドラインオプションを特定します。通常、オプションはホストで権限を開き、コンテナがホストで永続的に維持する必要があるホストのコンテンツをマウントします。
- **uninstall:**コンテナの実行を終了した後にホストシステムをクリーンアップします。

25.3. RUNLABEL での RSYSLOG の実行

`rhel9/rsyslog` コンテナイメージは、`rsyslogd` デーモンのコンテナ化されたバージョンを実行するように設計されています。`rsyslog` イメージには、`install`、`run` および `uninstall` の `runlabel` が含まれます。以下の手順に従って、`rsyslog` イメージのインストール、実行、アンインストールを行います。

前提条件

- **container-tools** メタパッケージがインストールされている。

手順

1. **rsyslog** イメージをプルします。

```
# podman pull registry.redhat.io/rhel9/rsyslog
```

2. **rsyslog** の **install** runlabel を表示します。

```
# podman container runlabel install --display rhel9/rsyslog
command: podman run --rm --privileged -v /:/host -e HOST=/host -e
IMAGE=registry.redhat.io/rhel9/rsyslog:latest -e NAME=rsyslog
registry.redhat.io/rhel9/rsyslog:latest /bin/install.sh
```

これは、コマンドがホストに特権を開き、コンテナ内の **/host** にホストの root ファイルシステムをマウントし、**install.sh** スクリプトを実行します。

3. **rsyslog** の **install** runlabel を実行します。

```
# podman container runlabel install rhel9/rsyslog
command: podman run --rm --privileged -v /:/host -e HOST=/host -e
IMAGE=registry.redhat.io/rhel9/rsyslog:latest -e NAME=rsyslog
registry.redhat.io/rhel9/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
```

これにより、**rsyslog** イメージが後で使用するホストシステムにファイルが作成されます。

4. **rsyslog** の **run** runlabel を表示します。

```
# podman container runlabel run --display rhel9/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/rhel9/rsyslog:latest -e
NAME=rsyslog --restart=always registry.redhat.io/rhel9/rsyslog:latest /bin/rsyslog.sh
```

これは、コマンドがホストへの特権を開き、**rsyslog** コンテナを起動して **rsyslogd** デーモンを実行するときに、コンテナ内のホストから多数のファイルとディレクトリーをマウントすることを示しています。

5. **rsyslog** の **run** runlabel を実行します。

```
# podman container runlabel run rhel9/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/rhel9/rsyslog:latest -e  
NAME=rsyslog --restart=always registry.redhat.io/rhel9/rsyslog:latest /bin/rsyslog.sh  
28a0d719ff179adcea81eb63cc90fcd09f1755d5edb121399068a4ea59bd0f53
```

rsyslog コンテナは権限を開き、ホストから必要なものをマウントし、バックグラウンドで **rsyslogd** デーモンを実行します (**-d**)。 **rsyslogd** デーモンは、ログメッセージを収集し、メッセージを **/var/log** ディレクトリー内のファイルに送信します。

6. **rsyslog** の **uninstall** runlabel を表示します。

```
# podman container runlabel uninstall --display rhel9/rsyslog  
command: podman run --rm --privileged -v /:/host -e HOST=/host -e  
IMAGE=registry.redhat.io/rhel9/rsyslog:latest -e NAME=rsyslog  
registry.redhat.io/rhel9/rsyslog:latest /bin/uninstall.sh
```

7. **rsyslog** の **uninstall** runlabel を実行します。

```
# podman container runlabel uninstall rhel9/rsyslog  
command: podman run --rm --privileged -v /:/host -e HOST=/host -e  
IMAGE=registry.redhat.io/rhel9/rsyslog:latest -e NAME=rsyslog  
registry.redhat.io/rhel9/rsyslog:latest /bin/uninstall.sh
```



注記

この場合、**uninstall.sh** スクリプトは、**/etc/logrotate.d/syslog** ファイルを削除します。設定ファイルはクリーンアップされません。

第26章 CONTAINER-TOOLS API の使用

varlink ライブラリーを使用する Podman リモート API に代わって、新しい REST ベースの Podman 2.0 API が導入されました。新規 API は、ルートフル環境およびルートレス環境の両方で動作します。

Podman v2.0 RESTful API は、Podman のサポートを提供する Libpod API と、Docker 互換 API で構成されます。この新しい REST API を使用すると、cURL、Postman、Google の Advanced REST クライアントなどのプラットフォームから Podman を呼び出すことができます。



注記

podman サービスはソケットのアクティブ化をサポートしているため、ソケット上の接続がアクティブでない限り、podman サービスは実行されません。したがって、ソケットのアクティブ化機能を有効にするには、**podman.socket** サービスを手動で起動する必要があります。ソケット上で接続がアクティブになると、podman サービスが起動し、要求された API アクションが実行されます。アクションが完了すると、podman プロセスが終了し、podman サービスは非アクティブな状態に戻ります。

26.1. ROOT モードで SYSTEMD を使用した PODMAN API の有効化

以下を実行できます。

1. **systemd** を使用して Podman API ソケットをアクティベートします。
2. Podman クライアントを使用して基本的なコマンドを実行します。

前提条件

- **podman-remote** パッケージがインストールされている。

```
# dnf install podman-remote
```

手順

1. サービスをすぐに起動します。

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

2. **docker-podman** パッケージを使用して **var/lib/docker.sock** へのリンクを有効にするには以下を実行します。

```
# dnf install podman-docker
```

検証手順

1. Podman のシステム情報を表示します。

```
# podman-remote info
```

2. リンクを確認します。

```
# ls -al /var/run/docker.sock
```

```
lrwxrwxrwx. 1 root root 23 Nov  4 10:19 /var/run/docker.sock -> /run/podman/podman.sock
```

関連情報

- [Podman v2.0 RESTful API](#)
- [A First Look At Podman 2.0 API](#)
- [スニークピーク:Podman の新しい REST API](#)

26.2. ルートレスモードで SYSTEMD を使用した PODMAN API の有効化

systemd を使用して、Podman API ソケットと Podman API サービスをアクティベートできます。

前提条件

- **podman-remote** パッケージがインストールされている。

```
# dnf install podman-remote
```

手順

1. サービスをすぐに有効にして起動します。

```
$ systemctl --user enable --now podman.socket
```

2. オプション: Docker を使用してプログラムがルートレス Podman ソケットを操作できるようにするには、以下を実行します。

```
$ export DOCKER_HOST=unix:///run/user/<uid>/podman//podman.sock
```

検証手順

1. ソケットのステータスを確認します。

```
$ 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 はアクティブで、**/run/user/<uid>/podman.podman.sock** をリッスンしています。<uid> はユーザーの ID です。

2. Podman のシステム情報を表示します。

```
$ podman-remote info
```

関連情報

- [Podman v2.0 RESTful API](#)
- [A First Look At Podman 2.0 API](#)
- [スニークピーク:Podman の新しい REST API](#)
- [Exploring Podman RESTful API using Python and Bash](#)

26.3. PODMAN API の手動実行

Podman API を実行できます。手動での実行は、特に Docker の互換性レイヤーを使用する場合など、API 呼び出しのデバッグに便利です。

前提条件

- **podman-remote** パッケージがインストールされている。

```
# dnf install podman-remote
```

手順

1. REST API のサービスを実行します。

```
# podman system service -t 0 --log-level=debug
```

- 値が 0 の場合はタイムアウトなしを意味します。ルートフルサービスのデフォルトのエンドポイントは **unix:/run/podman/podman.sock** です。
 - **--log-level <level>** オプションは、ロギングレベルを設定します。標準のロギングレベルは **debug**、**info**、**warn**、**error**、**fatal**、および **panic** です。
2. 別のターミナルで、Podman のシステム情報を表示します。**podman-remote** コマンドは、通常の **podman** コマンドとは異なり、Podman ソケットを介して通信します。

```
# podman-remote info
```

3. Podman API をトラブルシューティングし、要求および応答を表示するには、**curl** コマンドを使用します。JSON 形式で Linux サーバーの Podman インストールの情報を取得するには、以下を実行します。

```
# 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 コーティリティーは、コマンドライン JSON プロセッサです。

4. **registry.access.redhat.com/ubi8/ubi** コンテナイメージをプルします。

```

# 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. プルしたイメージを表示します。

```

# 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  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"  
    ],  
    ...  
  ]  
}  
]
```

関連情報

- [Podman v2.0 RESTful API](#)
- [スニークピーク:Podman の新しい REST API](#)
- [Exploring Podman RESTful API using Python and Bash](#)
- [podman-system-service の man ページ](#)