Red Hat Enterprise Linux Atomic Host 7

コンテナー開発における推奨プラクティス

コンテナー開発の推奨プラクティスガイド

Last Updated: 2023-03-25

Red Hat Enterprise Linux Atomic Host 7 コンテナー開発における推奨プラクティス

コンテナー開発の推奨プラクティスガイド

Enter your first name here. Enter your surname here.
Enter your organisation's name here. Enter your organisational division here.
Enter your email address here.

法律上の通知

概要

基本的なコンテナー関連のプラクティス。

目次

1. 非推奨の通知

重要

2020 年 8 月 6 日の時点で、Red Hat Enterprise Linux Atomic Host は廃止され、アクティブなサポートは提供されなくなりました。したがって、本書は非推奨となり、更新を受信しなくなります。

第1章 概要

本書では、Red Hat がサポートするコンテナー開発の推奨事項を提供します。現在の Docker 駆動型の実装において、コンテナーは新規かつ迅速に開発されるテクノロジーですが、本書では Red Hat 内のコンテナーサポートの状態をキャプチャーします。コンテナーには多くのユースケースがあるため、本書では Red Hat が役立ち、サポートする基本的なコンテナー関連のプラクティスについての一般的な推奨事項を説明します。

1.1. コンテナーの伝搬

コンテナーの出所どのように取得すればよいでしょうか ?安全な方法で確実に入手するにはどうすれば良いですか ?

コンテナーはイメージからビルドされ、イメージはレジストリーのリポジトリーに保存されます。このトピックでは、**docker pull** コマンドがアクセスできる 2 つのレジストリーについて説明します。

- **docker.io** レジストリー
- [The Red Hat Registry](http://registry.access.redhat.com/)

1.1.1. docker pull のリスク

プル元のレジストリーなしで使用される **docker pull** コマンドは危険なコマンドです。Docker はソフトウェアの取得とソフトウェアのインストールを区別しません。この動作は RPM の場合とは異なります。RPM をインストールしない限り、**wget** を使用してマルウェアを含む RPM を取得することが推奨されます。ただし、が安全ではない場合、**docker pull** を使用してマルウェアをプルする場合は、イメージの取得はそのインストールと機能的に同等であるためです。

たとえば、取得時にコンテナーが特権として実行されるソフトウェアであると仮定します。コンテナーは、その設定を操作した後にのみ特権を再アクティブ化します。コンテナーの分離は通常自発的であり、デフォルトでは分離されていません。

警告

docker pull コマンドを使用する場合は注意が必要です。

コンテナーが Red Hat レジストリーにない場合、**docker pull** は **docker.io** レジストリーにフェイルオーバーします。Red Hat は、**docker.io** などのサードパーティーソースからのコンテナーのセキュリティーまたは信頼性を検証しません。Image-Naming Conventions の章も参照してください。

Red Hat レジストリー以外の場所からイメージを取得する場合は、**docker pull** を使用しないでください。可能な場合は、**docker load** および **docker save** を使用します。tarball で **docker load** および **docker save** を使用してから、イメージを確認することができます。

docker pull の代わりに **docker load** および **docker save** を使用する理由。Docker **負荷** および **docker save** は、システムをサードパーティーレジストリーに公開することで導入されたセキュリティー脆弱性を回避する方法を提供します。これは、**docker pull** を実行すると発生する可能性があります。

docker pull を使用する場合のコンテナーの証明および注意上の注意に関する詳しい情報は、**docker pull** を 開始する前に Red Hat のセキュリティーブログ を参照してください 。

- **docker pull**

docker pull の基本的な形式は **$ sudo docker pull repo/image:tag** で、**repo** と **tag** は任意です。**リポジトリー** および **タグ** が指定されていない場合、docker は **docker.io** レジストリーでイメージの検索を試行します。このため、常にイメージをプルするレジストリーに名前を付けることが推奨されます。レジストリーが指定されていない場合、docker は **docker.io** レジストリーでイメージの検索を試行します。タグが指定されていない場合、docker はデフォルトで最新のイメージをプルしようとします。

- **Docker の負荷**

docker 負荷の基本的な形式は、**$ sudo docker load -i input.tar** です。**input.tar** は、ローカルコンテナーレジストリーに読み込まれる tar イメージです。**-i** はオプションであり、**input.tar** ファイル名は任意です。**-i** もファイル名も指定されていない場合、**docker load** は STDIN の tar データを想定します。

- **docker save**

docker save の基本的な形式は **$ sudo docker save -o output.tar** です。**output.tar** は、ローカルコンテナーレジストリーに読み込まれる tar イメージです。**-o** はオプションであり、**output.tar** ファイル名は任意です。**-o** もファイル名も指定されていない場合、**docker save** はコンテナーデータを STDOUT に出力します。

第2章 コンテナーのビルド

コンテナーの更新を可能にするメカニズムを維持し、それらのコンテナーに適用されるセキュリティー脆弱性が検出されたときにパッチを適用できるようにする必要があります。コンテナーの構築方法を理解することで、コンテナーを適切に維持するのに役立ちます。

コンテナーにセキュリティー脆弱性が発生するリスクを最小限に抑えるには、コンテナーを更新してセキュリティーパッチを適用できるようにする必要があります。コンテナーを確実に更新できるようにするには、既存のパッケージインフラストラクチャーおよびツールをできるだけ多く活用する必要があります。つまり、Yum を使用する必要があります。そのため、実行中のコンテナーは他のディストリビューションの実行とは異なります。適用に精通しているすべてのルール：可能な場合は RPM を使用する必要があります。

第3章 ラベル

ラベルを使用すると、コマンドをオプションとしてコマンドラインに追加する代わりに、コンテナーイメージにコマンドを埋め込むことができます。これらは通常、特定のコンテナーの実行に必要な複数のオプションを持つ docker コマンドであり、特権としてコンテナーを実行することが含まれます。オプションはコンテナーごとに処理され、コマンドが非常に長く、覚えにくくなる可能性があります。ラベルは、この情報を docker に提供する別の方法です。

ラベルは、Docker ですでに利用可能なメタデータ上に構築されます。このメタデータは、コンテナーを定義する **.json** ファイルに保存されます。Docker には、名前と値のペアをコンテナーに挿入できる機能のコンテナーやラベルビルドに名前と値のペアを配置することができる機能があります。

3.1. ラベルの例

ラベルを使用すると、コマンドラインでの長い docker コマンドの実行が容易になります。以下の例は、ラベルを理解するのに役立ちます。

ここでは、セマンティック意味を持つ "run labels" を作成します。

GPU アクセラレーションおよびピルスオーディオで Chrome ブラウザーを実行したいとします。これを実行するには、コンテナーが GPU にアクセスする必要があります。これは十分ですが、設定を検討する必要があります。X11 と Window Manager は連携しています。これは設定になります。これら 2 つのプログラムが連携するために最も効率的な方法は、プログラムが記述される大きなフレームバッファーを 1 つ作成することです。これは、以下のような非常に長いコマンドで処理されます。

```
$ sudo docker run -v /dev/dri:/dev/dri \
-v /dev/snd:/dev/snd \
-v /dev/shm:/shm \
-ipc=container:foo_bar \
-privileged -e 'DISPLAY=:0' \
-u username rhel_chrome google-chrome
```

これをダウンさせてみましょう。

-v /dev/dri:/dev/dri

ビデオカードコンポーネント

-v /dev/snd:/dev/snd

サウンドカードコンポーネント

-v /dev/shm:/dev/shm

共有メモリー

shm

共有メモリー

-ipc

プロセス間の通信

container:foo_bar は、このラベルが **foo_bar** と呼ばれる別の実行中のコンテナーを参照することを意味します。これにより、実行中のコンテナー **foo_bar** が実行中のコンテナー **fedora_chrome** にリンクされます。この引数は、2 つのコンテナーのプロセス間通信名前空間を共有し、2 つのコンテナーが同じオブジェクトを参照し、2 つのコンテナーが同じ名前を使用してこれらのオブジェクトを参照するようにします。これは、**/dev/shm** で通信されるために重要になります。

共有メモリーは、プロセス間の通信を提供する 1 つの方法です。

-e 'DISPLAY=:0'

出力がどの X11 に表示されるか

-u username

これはユーザー名を指定します。

rhel_chrome

コンテナーの名前

google-chrome

コンテナーを実行するコマンド

第4章 イメージ命名規則

4.1. Red Hat のネーミングポリシー

Red Hat は、ユーザーに予測性を提供するために、Docker 形式イメージの一貫した命名ポリシーを提供しています。

V1 バージョンのプロトコルおよびレジストリー形式の Docker URL は GitHub リポジトリー名と同様に機能します。それらの構造は次のとおりです。

```
REGISTRY[:PORT]/USER/REPO[:TAG]
```

暗黙的なデフォルトレジストリーは **docker.io** です。これは、redhat/rhel などの相対 URL が **docker.io/redhat/rhel** に解決することを意味します。

特別な名前 "library/*" は、ダイレクトかつ接頭辞なしのイメージにマッピングされます。
例："docker.io/rhel"

4.2. 要求のリダイレクト

Red Hat コンテンツに対する要求は、**docker.io** から **registry.access.redhat.com** にリダイレクトされます。以下のマッピングは、このリダイレクトについて説明しています。

- docker.io/rhel → registry.access.redhat.com/rhel (rhel7)のエイリアス
- docker.io/rhel7 → registry.access.redhat.com/rhel7
- docker.io/rhel6 → registry.access.redhat.com/rhel6
- docker.io/redhat/* → registry.access.redhat.com/redhat/*

Red Hat Enterprise Linux 7.2 に同梱される Docker のバージョンは、デフォルトで内部 Red Hat レジストリーと通信します。これにより、Red Hat はレジストリーごとに namespace をセグメント化できます。

レジストリーの内容はタグ付けされた名前のマッピングであり、コンテンツのコピーは含まれません。このマッピングは、複数のシンボリックリンクを使用して実現されます。Red Hat は、データの整合性を確保するために、これらのリンクを最新の状態に保ちます。

REPO は、明示的にタグ付けされている多数のレイヤーと、非表示レイヤーを多数含むリポジトリーです。IMAGE は、リポジトリー内の特定のレイヤー補完ブランチです。多くの場合、イメージとリポジトリーは同義語で、同じとは見なされません。たとえば、異なるイメージを 1 つのリポジトリーにタグ付けすることができます。

第5章 プロセス管理

プロセス管理に関する本セクションでは、以下の 2 つの状況について説明します。

1. コンテナーがジョブを実行するために必要なファイルを格納する単一プロセスコンテナー
2. 同時に実行される複数の異なるプロセスで設定される managed-multiprocess コンテナー

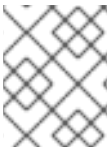
Apache は、単一プロセスコンテナーの例です。Apache は、すべて独自のロギングを行います。つまり、外部コンテナーがロギングを行う必要はありません。つまり、この場合は **systemd** を使用しても利点はありません。

GNOME 3 は、managed-multiprocess コンテナーの例です。この種のコンテナーを手動で管理しようとすると（例：upower と DBUS とログが連携しようとする場合など）、単に **systemd** が機能するようにするよりも有益です。

5.1. コンテナーでの systemd の実行

managed-multiprocess コンテナーの場合は、systemd を実装し、コンテナー内で実行中のプロセスを管理します。コンテナーで systemd を実行するには、以下のコマンドと同様のコマンドを実行します。

```
$ sudo docker run -it IMAGE /bin/bash
```



注記

上記のコマンド(**-it**)は、**systemd** を実行しているインタラクティブな tty コンテナーを起動します。

コンテナーで **systemd** を実行する際の過去のコンテキストについては、Dan Walsh's May 2014 blog post on the matter: Running systemd within a Docker Container を参照してください。

5.1.1. コンテナー内の journald および systemd

コンテナーでジャーナル監査を無効にします。docker ホストのみが **journald** の実行を許可します。ジャーナル監査はカーネル機能であり、一度に使用できる journald のインスタンスは 1 つだけです。(docker ホストが journald を実行している場合は問題ありません。)

5.1.2. サービスおよびコンテナー

サービスは、コンテナーで追加設定なしで機能します。コンテナーでサービス のみ の作業を行います。サービスがコンテナーで機能するには、特別な設定は必要ありません。

5.1.3. コンテナー外にあるハードウェアの管理

systemd を使用して外部ハードウェアを管理するコンテナーを作成することは可能ですが、コンテナーを設定すると、コンテナーが管理に使用できるよう設定された特定のハードウェア設定を想定するため、移植できません。

5.1.4. Systemd および Zombies

systemd は、動機の問題を解決します。プロセスが停止して異常になると、**init 1** はそれらを取得します。これはコンテナーの外部にあるため、コンテナー内で true に似ています。コンテナー内の **systemd** が、コンテナー外で異常を取り除くことを期待したように、異常な状態を取得することを想定

しています。

systemd （または **sysv**）なしでコンテナーを起動すると、そのコンテナーで失われたプロセスは取得されません。

第6章 LINTER FOR DOCKERFILE

Dockerfile の **Linter は、有効な Dockerfile** の構築に役立つエラーがないか Dockerfile をチェックするユーティリティーです。

Dockerfile の Linter にアクセスするには、Red Hat ログインと有効なサブスクリプションが必要です。

Dockerfile ユーティリティーの Linter には、https://access.redhat.com/labs/linterfordockerfile/でアクセスしてください。

Dockerfile を検証するには、以下を実行します。

1. Dockerfile に URL を指定するか、Dockerfile の内容を手動で貼り付けて Dockerfile をアップロードします。
2. 特定のプロファイル(Red Hat ISV など)を選択します。
3. **Check** を選択してください。