



OpenShift コンテナプラットフォーム 3.10

イメージの作成

OpenShift Container Platform 3.10 イメージ作成ガイド

OpenShift コンテナプラットフォーム 3.10 イメージの作成

OpenShift Container Platform 3.10 イメージ作成ガイド

法律上の通知

Copyright © 2018 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, 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 Software Collections 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

以下のトピックでは、OpenShift Container Platform 3.10 で使用可能な Docker イメージの記述およびテストにおけるベストプラクティスを説明しています。

目次

第1章 概要	3
第2章 ガイドライン	4
2.1. 概要	4
2.2. コンテナイメージの一般的なガイドライン	4
イメージの再利用	4
タグ内の互換性の維持	4
複数プロセスの回避	4
ラッパースクリプトでの exec の使用	4
一時ファイルの消去	5
正しい順序での命令の指定	5
重要なポートのマーク付け	6
環境変数の設定	6
デフォルトのパスワードの回避	6
SSHD の回避	6
永続データ向けのボリュームの使用	6
外部のガイドライン	7
2.3. OPENSIFT CONTAINER PLATFORM 固有のガイドライン	7
Source-To-Image (S2I) 向けのイメージの有効化	7
任意のユーザー ID のサポート	7
イメージ内の通信でのサービスの使用	8
共通のライブラリーの提供	8
設定での環境変数の使用	8
イメージのメタデータ設定	9
クラスタリング	9
ロギング	9
Liveness および Readiness プローブ	10
Templates (テンプレート)	10
2.4. 外部の参考資料	10
第3章 イメージのメタデータ	11
3.1. 概要	11
3.2. イメージメタデータの定義	11
第4章 S2I の要件	13
4.1. 概要	13
4.2. ビルドプロセス	13
4.3. S2I スクリプト	14
4.4. ONBUILD 命令でのイメージの使用	17
4.5. 外部の参考資料	18
第5章 S2I イメージのテスト	19
5.1. 概要	19
5.2. テストの要件	19
5.3. スクリプトおよびツールの生成	19
5.4. ローカルでのテスト	19
5.5. テストの基本的なワークフロー	20
5.6. イメージのビルドでの OPENSIFT CONTAINER PLATFORM の使用	21
第6章 カスタムのビルダー	22
6.1. 概要	22
6.2. カスタムビルダーイメージ	22
6.3. カスタムビルダーのワークフロー	23

第1章 概要

本ガイドは、OpenShift Container Platform で使用可能なコンテナイメージの記述およびテストに関するベストプラクティスを説明します。イメージを作成したら、「[内部のレジストリー](#)」にイメージをプッシュすることができます。

第2章 ガイドライン

2.1. 概要

OpenShift Container Platform で実行するためにコンテナイメージを作成する場合には、使用者がイメージを快適に使用できるように、イメージの作成者として考慮すべきベストプラクティスが複数あります。イメージは、変更できず、そのままの状態で使用するように作成されているので、以下のガイドラインでは、イメージを OpenShift Container Platform で簡単に使用でき、消費しやすくなるように、サポートします。

2.2. コンテナイメージの一般的なガイドライン

以下のガイドラインは、一般的なコンテナイメージの作成時に適用され、イメージが OpenShift Container Platform で使用されるかどうかは関係ありません。

イメージの再利用

できる限り、**FROM** ステートメントを使用して、適切なアップストリームイメージをベースに、お使いのイメージを設定することを推奨します。こうすることで、依存関係を直接更新する必要なく、更新があるとイメージが簡単にセキュリティー修正を取得できるようになります。

さらに、**FROM** 命令 (例: `rhel:rhel7`) のタグを使用して、お使いのイメージがどのバージョンのイメージを使用しているのかを明確にします。アップストリームの **latest** バージョンに壊れた変更が組み込まれている可能性があり、**latest** 以外のタグを使用することで、そのような変更の影響を受けないようにします。

タグ内の互換性の維持

独自のイメージにタグを付ける場合には、タグ内で後方互換性が維持されるようにすることを推奨します。たとえば、**foo** という名前のイメージがあり、現在バージョン 1.0 が含まれている場合には、タグに **foo:v1** を指定します。イメージの更新時には、オリジナルのイメージと互換性があれば、新しいイメージに **foo:v1** のタグを付けて、このタグのダウストリームのコンシューマーは、イメージを破壊することなく更新を取得できるようになります。

互換性のない更新を後にリリースした場合には、**foo:v2** などの新しいタグに切り替える必要があります。これにより、ダウストリームのコンシューマーはいつでも新しいバージョンに移行できますが、不注意で互換性のない新規イメージで破壊してしまうことはありません。**foo:latest** を使用するダウストリームコンシューマーの方には、互換性のない変更が導入される危険性があります。

複数プロセスの回避

データベースや **SSHD** など複数のサービスを 1 つのコンテナ内で起動しないようにしてください。コンテナは軽量で、簡単にリンクして複数のプロセスをオーケストレーションできるので、これは必要ありません。OpenShift Container Platform では、関連のあるイメージを 1 つの pod にグループ化して、簡単に共存、共同管理できるようになります。

このように共存させることで、コンテナがネットワークの namespace とストレージを通信用に共有できるようになります。また、イメージの更新頻度が低く、個別に更新されるので、更新の破壊レベルが低くなります。シグナル処理のフローは、起動したプロセスへのルーティングシグナルを管理する必要がないので、単一プロセスのほうが明確です。

ラッパースクリプトでの **exec** の使用

詳細は、『[Project Atomic documentation](#)』の「Always **exec** in Wrapper Scripts」のセクションを参照してください。

また、コンテナ内で実行すると、プロセスは PID 1 として実行されている点に留意してください。つまり、主なプロセスが中断された場合には、コンテナ全体が停止され、PID 1 プロセスから起動した子プロセスが終了します。

その他関連のある内容は、「[Docker and the PID 1 zombie reaping problem](#)」のブログ記事を参照してください。また、PID 1 や **init** システムの詳細は、「[Demystifying the init system \(PID 1\)](#)」のブログ記事を参照してください。

一時ファイルの消去

ビルドプロセスで作成される一時ファイルはすべて削除する必要があります。これには、**ADD** コマンドで追加したファイルも含まれます。たとえば、**yum install** の操作を実行してから、**yum clean** コマンドを実行することを強く推奨します。

yum キャッシュがイメージ層に残らないように、以下のように、**RUN** ステートメントを作成します。

```
RUN yum -y install mypackage && yum -y install myotherpackage && yum clean all -y
```

以下のように記述した場合には注意してください。

```
RUN yum -y install mypackage  
RUN yum -y install myotherpackage && yum clean all -y
```

上記のように記述すると、最初の **yum** 呼び出しにより、対象の層に追加のファイルが残り、**yum clean** 操作を後に実行してもこれらのファイルは削除できません。これらの追加ファイルは最終イメージでは確認できませんが、下層には存在します。

現在の Docker ビルドプロセスでは、前の層で何かが削除された場合でも、後の層でコマンドを実行してイメージが使用する容量を縮小できません。ただし、これについては今後変更される可能性はあります。後の層で **rm** コマンドを実行しても、ファイルが表示されていなくても、ダウンロードする全体のイメージサイズは減少しません。そのため、**yum clean** の例のように、層に書き込まれないように、できるだけ同じコマンドでファイルを削除することが一番です。

また、単一の **RUN** ステートメントで複数のコマンドを実行すると、イメージの階層数が減り、ダウンロードと実行時間が短縮されます。

正しい順序での命令の指定

Docker は **Dockerfile** を読み取り、トップダウンで命令を実行します。命令が正常に実行されると、同じイメージが次回ビルドされるときや、別のイメージがビルドされる時に再利用することができる階層が作成されます。**Dockerfile** の上部でほぼ変更されない命令を配置することが非常に重要です。こうすることで、上層で加えられた変更により、キャッシュが無効にならないので、同じイメージが次回、非常にすばやくビルドされるようになります。

たとえば、反復するファイルをインストールするための **ADD** コマンドと、パッケージを **yum install** する **RUN** コマンドが含まれる **Dockerfile** で作業を行う場合には、**ADD** コマンドを最後に配置することがベストです。

```
FROM foo  
RUN yum -y install mypackage && yum clean all -y  
ADD myfile /test/myfile
```

こうすることで、**myfile** を編集して **docker build** を返すたびに、システムは **yum** コマンドのキャッシュ階層を再利用し、**ADD** 操作に対してのみ、新規階層を生成します。

以下のように **Dockerfile** を記述した場合:

```
FROM foo
ADD myfile /test/myfile
RUN yum -y install mypackage && yum clean all -y
```

次に、**myfile** を変更して、**docker build** を再実行するたびに、**ADD** 操作は **RUN** 階層キャッシュを無効にするので、**yum** 操作も再実行する必要があります。

重要なポートのマーク付け

詳細は、『[Project Atomic documentation](#)』の「Always **EXPOSE** Important Ports」のセクションを参照してください。

環境変数の設定

ENV 命令で環境変数を設定することが適切です。一例として、プロジェクトのバージョンを設定するなどが挙げられます。バージョンを設定して、**Dockerfile** を確認せずにバージョンを簡単に見つけ出すことができます。別の例としては、**JAVA_HOME** など、別のプロセスで使用可能なシステムでパスを広告する場合などです。

デフォルトのパスワードの回避

デフォルトのパスワードは設定しないことが一番です。イメージを拡張して、デフォルトのパスワードを削除または変更するのを忘れることが多く、実稼働環境で使用するユーザーに誰でも知っているパスワードが割り当てられると、セキュリティの問題につながります。パスワードは、環境変数を使用して設定できるようにする必要があります。詳細は、『[設定での環境変数の使用](#)』のトピックを参照してください。

デフォルトのパスワードを設定することにした場合には、コンテナの起動時に適切な警告メッセージが表示されるようにしてください。メッセージで、デフォルトパスワードの値をユーザーに通知し、環境変数の設定など、パスワードの変更方法を説明する必要があります。

SSHD の回避

イメージで **SSHD** の実行を回避するようにしてください。ローカルホストで実行中のコンテナにアクセスするには、**docker exec** コマンドを使用できます。または、『[oc exec](#)』コマンドまたは『[oc rsh](#)』コマンドを使用して、OpenShift Container Platform クラスタで実行中のコンテナにアクセスできます。イメージで **SSHD** をインストール、実行すると、攻撃の経路が増え、セキュリティ修正が必要になります。

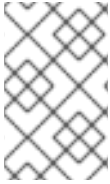
永続データ向けのボリュームの使用

イメージは、永続データ用に [Docker ボリューム](#) を使用する必要があります。こうすることで、OpenShift Container Platform により、コンテナを実行するノードにネットワークストレージがマウントされ、コンテナが新しいノードに移動した場合に、ストレージはそのノードにアタッチしなおされます。永続ストレージの全要件にボリュームを使用することで、コンテナが再起動されたり、移動されたりしても、コンテンツは保存されます。イメージがコンテナ内の任意の場所にデータを書き込む場合には、コンテンツは保存されない可能性があります。

コンテナが破棄された後でさえも保存する必要のあるデータはすべて、ボリュームに書き込む必要があります。Docker 1.5 では、コンテナに **readonly** フラグがあり、このフラグを使用して、コンテナの一時ストレージにデータが絶対に記述されないように強制することができます。その機能をもとにイメージを設計すると、後でこの機能をより簡単に活用できるようになります。

さらに、『[Dockerfile](#)』でボリュームを明示的に定義すると、イメージの消費者が簡単に、イメージの実行時に定義する必要のあるボリュームがどれかを理解できるようになります。

OpenShift Container Platform でボリュームがどのように使用されるかについては、『[Kubernetes ドキュメント](#)』を参照してください。



注記

永続ボリュームでも、イメージの各インストールには独自のボリュームがあり、ファイルシステムはインスタンス間で共有されません。つまり、ボリュームを使用してクラスターの状態を共有できません。

外部のガイドライン

他のガイドラインについては、以下の資料を参照してください。

- Docker ドキュメント: [「Best practices for writing Dockerfiles」](#)
- Project Atomic ドキュメント: [「Guidance for Container Image Authors」](#)

2.3. OPENSIFT CONTAINER PLATFORM 固有のガイドライン

以下は、OpenShift Container Platform で使用するためのコンテナイメージを作成時に適用されるガイドラインです。

Source-To-Image (S2I) 向けのイメージの有効化

開発者が提供した Ruby コードを実行するように設計された Ruby イメージなど、サードパーティー提供のアプリケーションコードを実行することが目的のイメージの場合には、イメージを有効化して [Source-to-Image \(S2I\)](#) ビルドツールと連携できるようにします。S2I は、インプットとして、アプリケーションのソースコードを受け入れるイメージを簡単に記述でき、アウトプットとして、組み立てられたアプリケーションを実行する新規イメージを簡単に生成することができるフレームワークです。

たとえば、この [Python イメージ](#) は S2I スクリプトを定義して、Python アプリケーションのさまざまなバージョンをビルドします。

イメージ用に S2I スクリプトを記述する方法については、[「S2I 要件」](#) のトピックを参照してください。

任意のユーザー ID のサポート

デフォルトでは OpenShift Container Platform は、任意に割り当てられたユーザー ID を使用してコンテナを実行します。こうすることで、コンテナエンジンの脆弱性が原因でコンテナから出ていくプロセスに対して追加のセキュリティーを設定できるので、ホストノードでパーミッションのエスカレーションが可能になります。

イメージが任意ユーザーとしての実行をサポートできるように、イメージ内のプロセスで記述される可能性のあるディレクトリーやファイルは、root グループが所有し、このグループに対して読み取り/書き込みの権限を割り当てる必要があります。実行予定のファイルには、グループの実行権限も必要です。

以下を Dockerfile に追加すると、root グループのユーザーがビルドイメージにアクセスできるように、ディレクトリーおよびファイルのパーミッションが設定されます。

```
RUN chgrp -R 0 /some/directory && \
    chmod -R g=u /some/directory
```

コンテナユーザーは常に、root グループのメンバーとして所属しているので、コンテナユーザーはこれらのファイルに対する読み取り、書き込みが可能です。root グループには、(root ユーザーのように) 特別なパーミッションがないので、この設定ではセキュリティーに関する懸念点はありません。さらに、コンテナで実行中のプロセスは、特権のあるユーザーとして実行されていないので、特権のあるポート (1024 未満のポート) をリッスンしてはいけません。

コンテナのユーザー ID が動的に生成されるので、`/etc/passwd` に関連のあるエントリーがありません。これが原因で、ユーザー ID を検索する必要のあるアプリケーションで問題が発生する可能性があります。

ります。この問題に対応するには、イメージの起動スクリプトの一部として、コンテナのユーザー ID を指定した **passwd** のファイルエントリを動的に作成します。Dockerfile の内容は、以下のようになります。

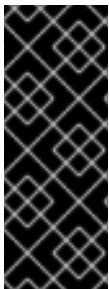
```
RUN chmod g=u /etc/passwd
ENTRYPOINT [ "uid_entrpoint" ]
USER 1001
```

uid_entrpoint には、以下を含めてください。

```
if ! whoami &> /dev/null; then
  if [ -w /etc/passwd ]; then
    echo "${USER_NAME:-default}:x:$(id -u):0:${USER_NAME:-default}
user:${HOME}:/sbin/nologin" >> /etc/passwd
  fi
fi
```

上記の完全な例は、[この Dockerfile](#) を参照してください。

最後に、**Dockerfile** の最後の **USER** 宣言は、ユーザー名ではなく、ユーザー ID (数値) を指定するようにしてください。こうすることで、OpenShift Container Platform が、イメージが実行しようとしている認証情報を検証して、root でイメージを実行できないようにします。理由は、特権のあるユーザーにより「[セキュリティ上の欠陥](#)」が生み出される可能性があるためです。イメージで **USER** が指定されていない場合は、親イメージの **USER** が継承されます。



重要

S2I イメージに、ユーザーを数値で指定した **USER** 宣言が含まれない場合には、デフォルトで、ビルドが失敗するようになっていました。名前が指定されたユーザーや root (**0**) ユーザーを使用するイメージを使用できるようにするには、「[プロジェクトのサービスアカウント](#)」 (**system:serviceaccount:<your-project>:builder**) を「特権付き」SCC (security context constraint) に追加してください。または、すべてのイメージを「[どのユーザーでも実行できるようにしてください](#)」。

イメージ内の通信でのサービスの使用

データの保存や取得にデータベースイメージにアクセスする必要のある Web フロントエンドイメージなど、別のイメージが提供するサービスとイメージが通信する場合には、イメージは OpenShift Container Platform 「[サービス](#)」を使用する必要があります。サービスは、コンテナが停止、開始、または移動されても、変更しない静的アクセスエンドポイントを提供します。さらに、サービスにより、要求が負荷分散されます。

共通のライブラリーの提供

サードパーティーが提供するアプリケーションコードの実行を目的とするイメージの場合は、プラットフォーム用に共通で使用されるライブラリーをイメージに含めるようにしてください。特に、プラットフォームで使用する共通のデータベース用のデータベースドライバーを設定してください。たとえば、Java フレームワークイメージを作成する場合に、MySQL や PostgreSQL には JDBC ドライバーを設定します。このように設定することで、アプリケーションの組み立て時に共通の依存関係をダウンロードする必要がありません。また、全依存関係が提供され、アプリケーション開発者の作業が簡素化されます。

設定での環境変数の使用

イメージのユーザーは、イメージをもとにダウストリームイメージを作成する必要なしに、設定が行えるようにしてください。つまり、ランタイム設定は環境変数を使用して処理してください。単純な設定の場合、実行中のプロセスは直接環境変数を使用します。より複雑な設定や、これをサポートしない

ランタイムの場合は、起動時に処理されるテンプレート設定ファイルを定義してランタイムを設定します。このプロセス時に、環境変数を使用して渡される値を、設定ファイルに置き換えることも、この値を使用して、設定ファイルに指定するオプションを決定することもできます。

環境変数を使用して、コンテナに証明書や鍵などのシークレットを渡すこともでき、これを推奨しています。環境変数を使用することで、シークレット値がイメージにコミットされたり、Docker レジストリーに漏洩してしまったりしないようにします。

環境変数を指定して、イメージのコンシューマーは、イメージの上に新しい階層を作成することなく、データベースの設定、パスワード、パフォーマンスチューニングなどの動作をカスタマイズできます。代わりに、pod の定義時に環境変数の値を定義するだけで、イメージのリビルドなしに設定が変更されます。

非常に複雑なシナリオでは、ランタイム時にコンテナにマウントされるボリュームを使用して設定を提供することも可能です。ただし、この方法を使用する場合には、必要なボリュームや設定が存在しない時に明確なエラーメッセージが起動時に表示されるように、イメージが設定されている必要があります。

このトピックは、サービスエンドポイントの情報を渡す環境変数としてデータソースなどの設定は定義する必要がある点で、「[イメージ間の通信でのサービスの使用](#)」のトピックに関連しています。環境変数で設定を定義することで、アプリケーションは、アプリケーションイメージを変更せずに、OpenShift Container Platform 環境に定義されているデータソースサービスを動的に使用できます。

さらに、コンテナの **cgroups** 設定を確認して、調整を行う必要があります。これにより、イメージは利用可能なメモリー、CPU、他のリソースに合わせてチューニングが可能になります。たとえば、Java ベースのイメージは、制限を超えずに、メモリー不足のエラーが表示されないように、**cgroup** の最大メモリーパラメーターをもとにヒープをチューニングする必要があります。

Docker コンテナの **cgroup** クォータを管理する方法については、以下の資料を参照してください。

- ブログ記事: [「Resource management in Docker」](#)
- Docker ドキュメント: [「Runtime Metrics」](#)
- ブログ記事: [「Memory inside Linux containers」](#)

イメージのメタデータ設定

イメージのメタデータを定義することで、OpenShift Container Platform によるコンテナイメージの消費が改善され、イメージを使用する開発者が OpenShift Container Platform をより快適に使用できるようになります。たとえば、メタデータを追加して、イメージに関する役立つ情報を提供したり、必要とされる可能性のある他のイメージを提案したりできます。

サポートされるメタデータや、定義の方法に関する詳細は、「[イメージのメタデータ](#)」のトピックを参照してください。

クラスタリング

イメージのインスタンスを複数実行する意味を完全に理解する必要があります。最もシンプルな例では、サービスの負荷分散機能は、イメージの全インスタンスにトラフィックをルーティングします。ただし、セッションの複製などで、リーダーの選択やフェイルオーバーの状態を実行するには、多くのフレームワークが情報を共有する必要があります。

OpenShift Container Platform で実行時に、インスタンスでこのような通信を実現する方法を検討します。pod 同士で直接通信できますが、pod が起動、停止、移動するたびに、IP アドレスが変更されるので、クラスタリングスキームを動的にしておくことが重要です。

ロギング

すべてのロギングを標準出力に送信することが最適です。OpenShift Container Platform はコンテナ

から標準出力を収集し、表示が可能な中央ロギングサービスに送信します。別個のログコンテンツが必要な場合には、出力のプレフィックスに適切なキーワードを指定して、メッセージをフィルタリングできるようにしてください。

お使いのイメージがファイルにロギングをする場合には、手動で実行中のコンテナに入り、ログファイルを取得または表示する必要があります。

Liveness および **Readiness** プローブ

イメージ使用可能な「[liveness および readiness プローブ](#)」の例をまとめます。これらのプローブで、処理の準備ができるまでトラフィックがルーティングされず、コンテナのヘルス状態が良くない場合にコンテナが再起動されるので、ユーザーは安心してイメージをデプロイできます。

Templates (テンプレート)

イメージの「[テンプレート](#)」例を提供することも検討してください。テンプレートがあると、ユーザーは、正しく機能する設定を指定してイメージをすばやく簡単にデプロイできるようになります。完全を期するため、テンプレートには、イメージにドキュメントとして追加した「[liveness および readiness プローブ](#)」を含めるようにしてください。

2.4. 外部の参考資料

- [「Docker basics」](#)
- [「Dockerfile reference」](#)
- [「Project Atomic Guidance for Container Image Authors」](#)

第3章 イメージのメタデータ

3.1. 概要

イメージのメタデータを定義することで、OpenShift Container Platform によるコンテナイメージの消費が改善され、イメージを使用する開発者が OpenShift Container Platform をより快適に使用できるようになります。たとえば、メタデータを追加して、イメージに関する役立つ情報を提供したり、必要とされる可能性のある他のイメージを提案したりできます。

このトピックでは、現在のユースケースに必要なメタデータのみを定義します。他のメタデータまたはユースケースは、今後追加される可能性があります。

3.2. イメージメタデータの定義

Dockerfile で **LABEL** 命令を使用して、イメージのメタデータを定義することができます。ラベルは、イメージやコンテナにアタッチする鍵と値のペアである点が環境変数に似ています。ただし、ラベルは、実行中のアプリケーションに表示されず、イメージやコンテナをすばやく検索する場合にも使用可能な点が、環境変数とは異なります。

LABEL 命令に関する詳細は、[Docker ドキュメント](#) を参照してください。

ラベル名は、通常 namespace を指定する必要があります。namespace は、対象のラベルを選択して使用するプロジェクトを反映するように設定してください。OpenShift Container Platform の場合は、namespace は **io.openshift** に、Kubernetes の場合は、namespace は **io.k8s** に設定してください。

形式に関する詳細は、「[Docker のカスタムメタデータ](#)」に関するドキュメントを参照してください。

表3.1 サポートされるメタデータ

変数	説明
io.openshift.tags	このラベルには、コンマ区切りの文字列値の一覧として表現されているタグの一覧が含まれます。タグを使用して、コンテナイメージを幅広い機能エリアに分類します。タグがあれば、UI および生成ツールがアプリケーションの作成プロセス時に適切なコンテナイメージを提案しやすくなります。 <pre>LABEL io.openshift.tags mongodb,mongodb24,nosql</pre>
io.openshift.wants	コンテナイメージにすでにタグが指定されていない場合に、生成ツールと UI が適切な提案を行うのに使用するタグを指定します。たとえば、コンテナイメージが mysql と redis が必要で、コンテナイメージに redis タグが指定されていない場合には、UI はこのイメージをデプロイメントに追加するように提案します。 <pre>LABEL io.openshift.wants mongodb,redis</pre>

変数	説明
io.k8s.description	<p>このラベルは、コンテナイメージのコンシューマーに、このイメージが提供するサービスや機能に関する詳細情報を提供するのに使用できます。UI は、この説明とコンテナイメージ名を使用して、人間が解読しやすい情報として、エンドユーザーに提供します。</p> <pre data-bbox="518 392 1412 481">LABEL io.k8s.description The MySQL 5.5 Server with master-slave replication support</pre>
io.openshift.non-scalable	<p>イメージは、この変数を使用して、スケーリングがサポートされていない旨を示し、その内容を UI でこのイメージを使用するユーザーに、通知します。スケーリング不可に指定した場合には、replicas の値を最初に 1 よりも大きい値に設定する必要があります。</p> <pre data-bbox="518 772 1236 817">LABEL io.openshift.non-scalable true</pre>
io.openshift.min-memory および io.openshift.min-cpu	<p>このラベルは、コンテナイメージが正しく機能するにはどの程度リソースが必要かを提案します。UI 経由でユーザーに対して、このコンテナイメージをデプロイすると、ユーザークォータを超過する可能性があることを警告します。この値は、Kubernetes の数量と互換性がある必要があります。</p> <pre data-bbox="518 1086 1125 1164">LABEL io.openshift.min-memory 8Gi LABEL io.openshift.min-cpu 4</pre>

第4章 S2I の要件

4.1. 概要

「Source-to-Image (S2I)」は、インプットとして、アプリケーションのソースコードを受け入れるイメージを簡単に記述でき、アウトプットとして、組み立てられたアプリケーションを実行する新規イメージを簡単に生成することができるフレームワークです。

再生成可能なコンテナイメージのビルドに S2I を使用する主な利点として、開発者の使い勝手の良さが挙げられます。ビルダーイメージの作成者は、イメージが最適な S2I パフォーマンスを実現できるように、**ビルドプロセス**と**S2I スクリプト**の基本的なコンセプト 2 点を理解する必要があります。

4.2. ビルドプロセス

ビルドプロセスは、以下の 3 つの要素で構成されており、これら 3 つを組み合わせることで最終的なコンテナイメージが作成されます。

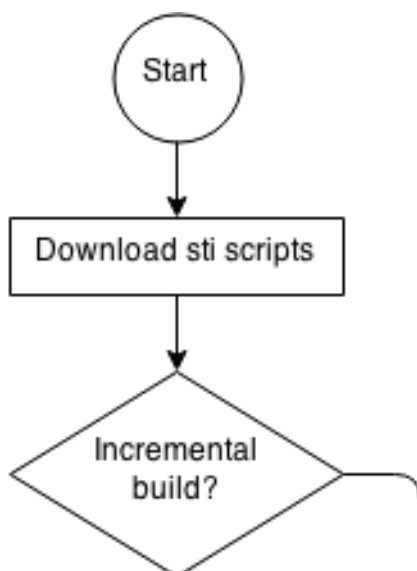
- ソース
- S2I スクリプト
- ビルダーイメージ

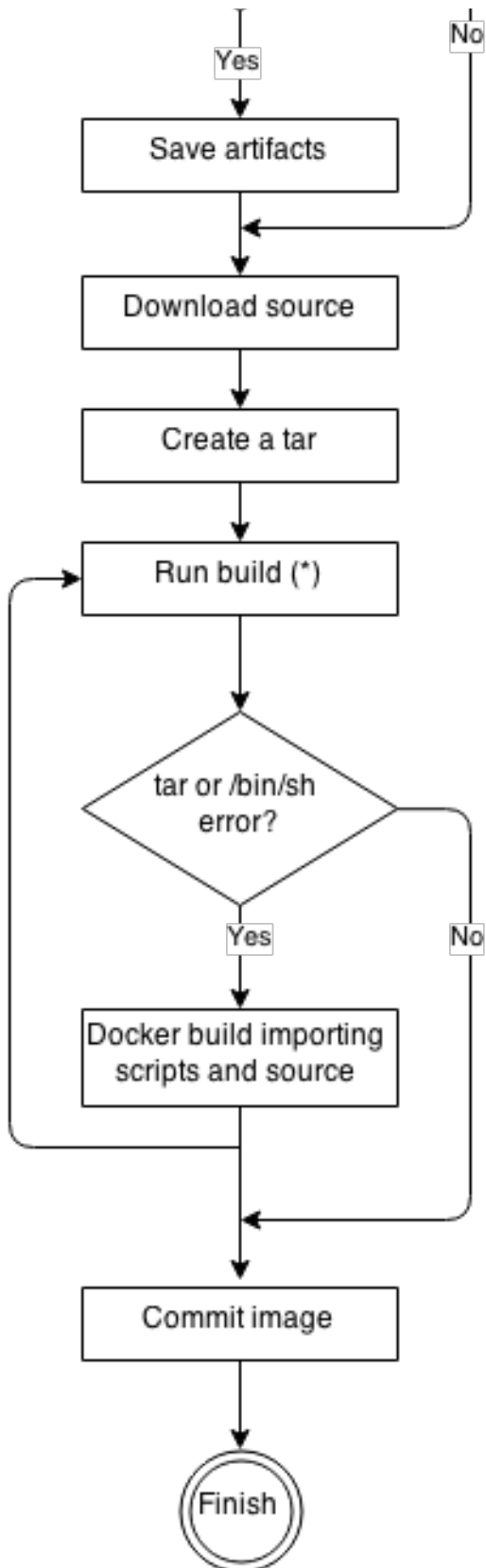
ビルドプロセス中に、S2I はソースとスクリプトをビルダーイメージ内に配置する必要があります。ビルダーイメージ内にソースとスクリプトを配置するために、S2I はソースとスクリプトを含む **tar** ファイルを作成してから、このファイルをビルダーイメージにストリーミングします。**assemble** スクリプトを実行する前に、S2I はファイルを展開して、ビルダーイメージからコンテンツを、デフォルトの **/tmp** ディレクトリーではなく、**io.openshift.s2i.destination** ラベルが指定する場所に配置します。

このプロセスを行うには、イメージに **tar** アーカイブユーティリティー (**tar** コマンドは **\$PATH** にあります) とコマンドラインインタープリター (**/bin/sh** コマンド) が必要です。これにより、イメージが最速のビルドパスを使用できるようになります。**tar** または **/bin/sh** コマンドが利用できない場合には **s2i** ビルド プロセスにより、イメージ内にソースとスクリプトが配置されるように、追加のコンテナが自動で強制実行されて初めて、通常のビルドが実行されます。

基本的な S2I ビルドワークフローについては、以下の図を参照してください。

図4.1 ビルドのワークフロー





- ビルドの実行では、ソース、スクリプト、アーティファクト (ある場合) を展開して、**assemble** スクリプトを実行します。2 回目の実行の場合には (**tar** または **/bin/sh** を検出できないエラーが発生した後など)、スクリプトとソースの両方があるので、**assemble** スクリプトの呼び出しのみを行います。

4.3. S2I スクリプト

S2I スクリプトは、ビルダーイメージ内でスクリプトを実行できる限り、どのプログラム言語でも記述できます。S2I は **assemble/run/save-artifacts** スクリプトを提供する複数のオプションをサポートします。ビルドごとに、これらの場所はすべて、以下の順番にチェックされます。


1. 「BuildConfig で指定する」スクリプト
2. アプリケーションソースの **.s2i/bin** ディレクトリーにあるスクリプト
3. デフォルトの URL (**io.openshift.s2i.scripts-url** ラベル) にあるスクリプト

イメージで指定した **io.openshift.s2i.scripts-url** ラベルも、BuildConfig で指定したスクリプトも、以下の形式のいずれかを使用します。

- **image:///path_to_scripts_dir**: S2I スクリプトが配置されているディレクトリーへのイメージ内の絶対パス
- **file:///path_to_scripts_dir**: S2I スクリプトが配置されているディレクトリーへのホスト上の相対パスまたは絶対パス
- **http(s)://path_to_scripts_dir**: S2I スクリプトが配置されているディレクトリーの URL

表4.1 S2I スクリプト

スクリプト	説明
assemble (必須)	<p>assemble スクリプトは、ソースからアプリケーションアーティファクトをビルドし、イメージ内の適切なディレクトリーに配置します。このスクリプトのワークフローは以下のとおりです。</p> <ol style="list-style-type: none"> 1. ビルドのアーティファクトを復元します。save-artifacts も定義するようにしてください (オプション)。 2. 任意の場所に、アプリケーションソースを配置します。 3. アプリケーションのアーティファクトをビルドします。 4. 実行に適した場所に、アーティファクトをインストールします。
run (必須)	<p>run スクリプトはアプリケーションを実行します。</p>
save-artifacts (オプション)	<p>save-artifacts スクリプトは、次に続くビルドプロセスを加速できるように全依存関係を集めます。以下に例を示します。</p> <ul style="list-style-type: none"> ● Ruby の場合は、Bundler でインストールされる gems ● Java の場合は、.m2 のコンテンツ <p>これらの依存関係は、tar ファイルに集められ、標準出力としてストリーミングされます。</p>
usage (オプション)	<p>usage スクリプトでは、ユーザーに、イメージの正しい使用方法を通知します。</p>

スクリプト	説明
test/run (オプション)	<p>test/run スクリプトでは、イメージが正しく機能しているかどうかを確認するための、単純なプロセスを作成できます。このプロセスの推奨フローは以下のとおりです。</p> <ol style="list-style-type: none"> 1. イメージをビルドします。 2. イメージを実行して usage スクリプトを検証します。 3. s2i build を実行して assemble スクリプトを検証します。 4. 再度、s2i build を実行して、save-artifacts と assemble スクリプトの保存、復元アーティファクト機能を検証します (オプション)。 5. イメージを実行して、テストアプリケーションが機能していることを確認します。 <p>詳細は、「S2I イメージのテスト」 のトピックを参照してください。</p> <div style="display: flex; align-items: flex-start;"> <div style="flex: 1;">  </div> <div style="flex: 2;"> <p>注記</p> <p>test/run スクリプトでビルドしたテストアプリケーションを配置する推奨の場所は、イメージリポジトリの test/test-app ディレクトリーです。詳しい情報は、S2I ドキュメント を参照してください。</p> </div> </div>

S2I スクリプトの例



注記

以下の例は Bash で記述されており、tar の内容はすべて **/tmp/s2i** ディレクトリーに展開されることが前提です。

例4.1 assemble スクリプト:

```
#!/bin/bash

# restore build artifacts
if [ "$(ls /tmp/s2i/artifacts/ 2>/dev/null)" ]; then
  mv /tmp/s2i/artifacts/* $HOME/.
fi

# move the application source
mv /tmp/s2i/src $HOME/src

# build application artifacts
pushd ${HOME}
make all
```

```
# install the artifacts
make install
popd
```

例4.2 run スクリプト:

```
#!/bin/bash

# run the application
/opt/application/run.sh
```

例4.3 save-artifacts スクリプト:

```
#!/bin/bash

pushd ${HOME}
if [ -d deps ]; then
    # all deps contents to tar stream
    tar cf - deps
fi
popd
```

例4.4 usage スクリプト:

```
#!/bin/bash

# inform the user how to use the image
cat <<EOF
This is a S2I sample builder image, to use it, install
https://github.com/openshift/source-to-image
EOF
```

4.4. ONBUILD 命令でのイメージの使用

ONBUILD 命令は、以下など、Docker の公式イメージの多くに含まれています。

- Ruby
- Node.js
- Python

ONBUILD に関する詳しい情報は、[Docker ドキュメント](#) を参照してください。

S2I は、開始されると、S2I プロセスでビルドインプットの注入に必要とされる **sh** と **tar** バイナリーが、ビルダーイメージに含まれているかどうかを検出します。ビルダーイメージでこれらの要件が満たされていない場合には、代わりにインプットを階層化するコンテナビルドの実行を試みます。ビル

ダーイメージに **ONBUILD** 命令が含まれる場合には、**ONBUILD** 命令が階層化プロセス中に実行され、S2I ビルドよりセキュリティが低い一般的なコンテナが実行されるのと同等で、明示的なパーミッションが必要となるので、S2I ではビルドが失敗します。

このように、S2I ビルダイメージに **ONBUILD** 命令が含まれていないことを確認するか、**sh** と **tar** バイナリーの要件が満たされていることを確認してください。

4.5. 外部の参考資料

- [S2I イメージ作成のチュートリアル](#)
- [S2I プロジェクトのリポジトリ](#)

第5章 S2I イメージのテスト

5.1. 概要

Source-to-Image (S2I) ビルダーイメージの作成者は、S2I イメージをローカルでテストして、自動テストや継続的な統合に OpenShift Container Platform ビルドシステムを使用できます。



注記

続行する前に、S2I アーキテクチャーの詳細については、「[S2I 要件](#)」のトピックを参照してください。

「[S2I 要件](#)」のトピックに説明されているように、S2I ビルドを正常に実行するには、S2I に **assemble** と **run** スクリプトが必要です。S2I 以外のコンテナイメージを実行した場合に、**save-artifacts** スクリプトがあると、ビルドのアーティファクトが再利用され、**usage** スクリプトがあると、用途の情報がコンソールに出力されるようになります。

S2I イメージのテストは、ベースのコンテナイメージを変更したり、コマンドが使用するツールが更新されたりした場合でも、上記のコマンドが正しく機能することを確認するのが目的です。

5.2. テストの要件

test スクリプトは、基本的に **test/run** に配置されます。このスクリプトは、OpenShift Container Platform S2I イメージビルダーが呼び出し、単純な Bash スクリプトか静的な Go バイナリーのいずれかの形式を取ることができます。

test/run スクリプトは、S2I ビルドを実行するので、S2I バイナリーを **\$PATH** に配置しておく必要があります。必要に応じて、[S2I README](#) のインストールの手順に従うようにしてください。

S2I は、アプリケーションのソースコードおよびビルダーイメージを統合します。これをテストするには、ソースが実行可能なコンテナイメージに変換されたことを検証するためのサンプルアプリケーションのソースが必要です。サンプルアプリケーションはシンプルであるはずですが、**assemble** および **run** スクリプトの極めて重要な手順を実行する必要があります。

5.3. スクリプトおよびツールの生成

S2I ツールには、強力な生成ツールが含まれており、新しい S2I イメージの作成プロセスを加速化します。**s2i create** コマンドでは、**Makefile** 以外に、必要とされる S2I スクリプトとテストツールすべてが生成されます。

```
$ s2i create <image name> <destination directory>
```

生成された **test/run** スクリプトは、使用性を高めるために調整する必要がありますが、このスクリプトは開発のスタート地点としては、最適です。



注記

s2i create コマンドで生成した **test/run** スクリプトでは、サンプルアプリケーションのソースを、**test/test-app** ディレクトリーに配置しておく必要があります。

5.4. ローカルでのテスト

S2I イメージのテストをローカルでテストする方法として、生成した **Makefile** を使用するのが最も簡単です。

s2i create コマンドを使用しない場合には、以下の **Makefile** テンプレートをコピーして、**IMAGE_NAME** パラメーターをお使いのイメージ名に置き換えます。

例5.1 Makefile の例

```
IMAGE_NAME = openshift/ruby-20-centos7

build:
  docker build -t $(IMAGE_NAME) .

.PHONY: test
test:
  docker build -t $(IMAGE_NAME)-candidate .
  IMAGE_NAME=$(IMAGE_NAME)-candidate test/run
```

5.5. テストの基本的なワークフロー

test スクリプトは、テストするイメージをすでにビルドしていることが前提です。必要に応じて、以下のコマンドで S2I イメージを先にビルドしてください。

```
$ docker build -t <BUILDER_IMAGE_NAME>
```

以下の手順では、S2I イメージビルダーをテストするデフォルトのワークフローを記述しています。

1. **usage** スクリプトが機能していることを確認します。

```
$ docker run <BUILDER_IMAGE_NAME> .
```

2. イメージをビルドします。

```
$ s2i build file:///path-to-sample-app <BUILDER_IMAGE_NAME>
<OUTPUT_APPLICATION_IMAGE_NAME>
```

3. オプションで、**save-artifacts** をサポートする場合には、再度、手順 2 を実行して、保存して復元したアーティファクトが正しく機能することを確認します。
4. コンテナを実行します。

```
$ docker run <OUTPUT_APPLICATION_IMAGE_NAME>
```

5. コンテナが実行され、アプリケーションが応答していることを確認します。

これらの手順を実行すると、通常、ビルダーイメージが予想通りに機能しているかどうか分かります。

5.6. イメージのビルドでの **OPENSIFT CONTAINER PLATFORM** の使用

新しい S2I ビルダージェイメージを構成する **Dockerfile** と他のアーティファクトが準備できたら、git リポジトリに配置して、OpenShift Container Platform を使用し、イメージをビルドしてプッシュします。後は、お使いのリポジトリを参照する「[Docker ビルド](#)」を定義するだけです。

OpenShift Container Platform インスタンスが公開 IP アドレスでホストされている場合には、ビルドは、S2I ビルダージェイメージの GitHub リポジトリにプッシュするたびに、トリガーされます。詳細は、「[webhook トリガー](#)」を参照してください。

ImageChangeTrigger を使用して、更新した S2I イメージをもとにアプリケーションのリビルドをトリガーすることも可能です。詳しい情報は、「[イメージ変更トリガー](#)」を参照してください。

第6章 カスタムのビルダー

6.1. 概要

ビルドプロセス全体を行う固有のビルダーイメージを定義できるように、OpenShift Container Platform の「[カスタムビルドストラテジー](#)」は、コンテナイメージの作成がよく使用されるようになり、できた差を埋めるために設計されました。ビルドが個別のアーティファクト (パッケージ、JAR、WAR、インストール可能な ZIP およびベースイメージなど) を生成する必要がある場合には、カスタムビルドストラテジーを使用する [カスタムビルダーイメージ](#) がこの要件を満たすのに最適です。

カスタムビルダーイメージは、RPM またはベースのコンテナイメージの構築など、ビルドプロセスのロジックに組み込まれるプレーンなコンテナイメージです。[openshift/origin-custom-docker-builder](#) イメージは、カスタムのビルダーイメージの実装例として [Docker Hub](#) で入手できます。

さらに、カスタムビルダーは、単体または統合テストを実行する CI/CD フローなどの拡張ビルドプロセスを実装できます。

カスタムのビルドストラテジーの利点を完全に活用するには、任意のオブジェクトをビルド可能な、カスタムビルダーイメージの作成方法を理解する必要があります。

6.2. カスタムビルダーイメージ

呼び出し時に、カスタムのビルダーイメージは、ビルドの続行に必要な情報が含まれる以下の環境変数を受け取ります。

表6.1 カスタムビルダーの環境変数

変数名	説明
BUILD	Build オブジェクト定義のシリアル化された JSON すべて。シリアル化した中で固有の API バージョンを使用する必要がある場合は、ビルド設定の「 カスタムストラテジーの仕様 」で、 buildAPIVersion パラメーターを設定できます。
SOURCE_REPOSITORY	ビルドするソースが含まれる Git リポジトリの URL
SOURCE_URI	SOURCE_REPOSITORY と同じ値を仕様します。どちらでも使用できます。
SOURCE_CONTEXT_DIR	ビルドに使用する Git リポジトリのサブディレクトリーを指定します。定義された場合のみ表示されます。
SOURCE_REF	ビルドする git 参照
ORIGIN_VERSION	ビルドオブジェクトを作成した OpenShift Container Platform のマスターのバージョン
OUTPUT_REGISTRY	イメージをプッシュする先の Docker レジストリー
OUTPUT_IMAGE	ビルドするイメージの Docker タグ名

変数名	説明
PUSH_DOCKERCFG_P ATH	docker push の操作実行に使用する Docker の認証情報へのパス
DOCKER_SOCKET	Docker ソケットの公開がビルド設定で有効化されている場合に Docker ソケットへのパスを指定します (exposeDockerSocket が true に設定されている場合)。

6.3. カスタムビルダーのワークフロー

カスタムビルダーイメージの作者は、ビルドプロセスを柔軟に定義できますが、ビルダーイメージは、OpenShift Container Platform 内でビルドがシームレスに実行されるように必要とされる以下の手順に従う必要があります。

1. **Build** オブジェクト定義には、ビルドの入力パラメーターの必要情報がすべて含まれています。
2. ビルドプロセスを実行します。
3. ビルドでイメージが生成される場合には、ビルドの出力場所が定義されていれば、その場所にプッシュします。他の出力場所には、環境変数を使用して渡すことができます。