



# OpenShift Container Platform 3.11

## イメージの作成

OpenShift Container Platform 3.11 イメージ作成ガイド



# OpenShift Container Platform 3.11 イメージの作成

---

## OpenShift Container Platform 3.11 イメージ作成ガイド

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

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

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

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

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

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

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

Linux<sup>®</sup> is the registered trademark of Linus Torvalds in the United States and other countries.

Java<sup>®</sup> is a registered trademark of Oracle and/or its affiliates.

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

MySQL<sup>®</sup> is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js<sup>®</sup> 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<sup>®</sup> 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 3.11 で使用可能な Docker イメージの作成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

<b>第1章 概要</b> .....	<b>4</b>
<b>第2章 ガイドライン</b> .....	<b>5</b>
2.1. 概要	5
2.2. コンテナイメージについての一般的なガイドライン	5
イメージの再利用	5
タグ内の互換性の維持	5
複数プロセスの回避	5
ラッパースクリプトでの exec の使用	5
一時ファイルの消去	6
適切な順序での命令の指定	6
重要なポートへのマーキング	7
環境変数の設定	7
デフォルトパスワードの回避	7
SSHD の回避	7
永続データ向けのボリュームの使用	7
外部のガイドライン	8
2.3. OPENSIFT CONTAINER PLATFORM 固有のガイドライン	8
Source-To-Image (S2I) 向けのイメージの有効化	8
任意のユーザー ID のサポート	8
イメージ内の通信でのサービスの使用	10
共通ライブラリーの提供	10
設定での環境変数の使用	10
イメージメタデータの設定	11
クラスターリング	11
ロギング	11
Liveness および Readiness プローブ	11
テンプレート	11
2.4. 外部の参考資料	11
<b>第3章 イメージのメタデータ</b> .....	<b>13</b>
3.1. 概要	13
3.2. イメージメタデータの定義	13
<b>第4章 S2I 要件</b> .....	<b>15</b>
4.1. 概要	15
4.2. ビルドプロセス	15
4.3. S2I スクリプト	16
4.4. ONBUILD 命令でのイメージの使用	19
4.5. 外部の参考資料	20
<b>第5章 S2I イメージのテスト</b> .....	<b>21</b>
5.1. 概要	21
5.2. テストの要件	21
5.3. スクリプトおよびツールの生成	21
5.4. ローカルでのテスト	21
5.5. テストの基本的なワークフロー	22
5.6. イメージのビルドでの OPENSIFT CONTAINER PLATFORM の使用	23
<b>第6章 カスタムのビルダー</b> .....	<b>24</b>
6.1. 概要	24
6.2. カスタムビルダーイメージ	24





## 第1章 概要

本書では、OpenShift Container Platform で使用可能なコンテナイメージの作成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 で実行するコンテナイメージを作成する場合には、イメージの作成者は、イメージの使いやすさの点で数多くのベストプラクティスを考慮する必要があります。イメージは変更不可で、そのままの状態で使用されることが意図されているため、以下のガイドラインは、イメージを使用しやすく、OpenShift Container Platform で簡単に使用できるようにするのに役立ちます。

### 2.2. コンテナイメージについての一般的なガイドライン

以下のガイドラインは、通常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とは関係がありません。

#### イメージの再利用

可能な場合は、**FROM** ステートメントを使用し、適切なアップストリームイメージをベースとしてイメージを設定することを推奨します。これにより、依存関係を直接更新する必要なく、イメージが更新時にアップストリームイメージからセキュリティ修正を簡単に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ベースとしているかを明確にします。アップストリームイメージの **latest** バージョンを使用すると互換性に影響のある変更が組み込まれる可能性があるため、**latest** 以外のタグを使用することができません。

#### 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、現時点でバージョン 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、元のイメージとの互換性がある限り、新しいイメージに `foo:v1` のタグを付けることができ、このタグのダウストリームのコンシューマーは、互換性に関する影響を被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、意図せずにこの互換性のない新規イメージによる影響を受けることはありません。`foo:latest` を使用するダウストリームコンシューマーには、互換性のない変更が導入されるリスクがあります。

#### 複数プロセスの回避

データベースや **SSH** 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、複数のプロセスをオーケストレーションするために簡単にリンクできるので、複数プロセスの実行は不要です。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、簡単に共存させ、共同管理することができます。

この共存により、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、更新による中断の可能性が低くなります。シグナル処理のフローは、生成され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ため、単一プロセスの場合により明確になります。

#### ラッパースクリプトでの `exec` の使用

詳細は、[Project Atomic ドキュメント](#) の Always `exec` in Wrapper Scripts 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。

その他の影響については、[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 および `init` システムの詳細については、[Demystifying the init system \(PID 1\)](#) のブログ記事も参照してください。

### 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レイヤーに残らないように、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、対象のレイヤー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、下位レイヤーには存在します。

現在のコンテナビルドプロセスでは、前のレイヤーで何かが削除された場合でも、後のレイヤー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。後のレイヤーでファイルが表示されていなくても **rm** コマンドを実行したとしても、ダウンロードするイメージの全体のサイズを縮小することになりません。そのため、**yum clean** の場合のように、可能な場合は後にレイヤーに書き込まれないように、ファイルの作成に使用したのと同じコマンドでファイルを削除することが最も適切と言えます。

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、イメージのレイヤー数が減り、ダウンロードと実行時間が短縮されます。

### 適切な順序での命令の指定

コンテナビルダーは **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層が作成されます。**Dockerfile** の上部にほとんど変更されない命令を配置することは非常に重要です。こうすることで、上位レイヤーで加えられた変更によってキャッシュが無効にならないので、同じイメージの次のビルドをすばやく実行できます。

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最善の方法です。

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

これにより、**myfile** を編集して **podman build** または **docker build** を返すたびに、システムは **yum** コマンドのキャッシュされたレイヤーを再利用し、**ADD** 操作に対してのみ新規レイヤーを生成します。

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、**podman build** または **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。

### 重要なポートへのマーキング

詳細は、**Project Atomic** ドキュメント 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ことなどが挙げられます。これにより、**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公開する場合などです。

### デフォルトパスワードの回避

デフォルトのパスワードは設定しないことが最善の策です。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あります。これは、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、セキュリティーの問題に発展する可能性があります。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](#) について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。メッセージは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ものである必要があります。

### SSHD の回避

イメージでの **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、**podman exec** または **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。イメージを使用して **SSHD** をインストール、実行すると、攻撃の経路が増え、セキュリティー修正が必要になります。

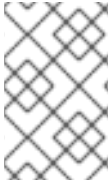
### 永続データ向けのボリュームの使用

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再度割り当てられます。永続ストレージのすべての要件に対応するよう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。コンテナエンジンはコンテナの **readonly** フラグをサポートしており、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決して記述されないようにすることができます。イメージをこの機能に基づいて設計すると、この機能を後に利用することがより簡単になります。

さらに、**Dockerfile** でボリュームを明示的に定義すると、イメージの利用者が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簡単に理解できるようになります。

OpenShift Container Platform でのボリュームの使用方法についての詳細は、[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タンス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 [Source-to-Image \(S2I\)](#) ビルドツールと連携できるようにすることができます。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、アセンブルされたアプリケーションをアウトプットとして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、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されたイメージで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+rwX /some/directory
```

コンテナユーザーは常に root グループのメンバーであるため、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。



## 警告

コンテナの慎重に扱うべき分野のディレクトリーおよびファイルパーミッションを変更する場合には注意が必要です (通常のシステムの扱い方と同様です)。

`/etc/passwd` などの慎重に扱うべき分野に適用されると、意図しないユーザーによるこのようなファイルの変更が可能となり、コンテナやホストにセキュリティ上のリスクが生じる可能性があります。CRI-O は、ランダムユーザー ID のコンテナの `/etc/passwd` への挿入をサポートするため、そのパーミッションを変更する必要はありません。

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) をリッスンできません。

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。この問題に対処する 1 つの方法として、`nss wrapper` を使用し、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 で `passwd` ファイルを動的に作成する方法があります。

```
export USER_ID=$(id -u)
export GROUP_ID=$(id -g)
envsubst < ${HOME}/passwd.template > /tmp/passwd
export LD_PRELOAD=/usr/lib64/libnss_wrapper.so
export NSS_WRAPPER_PASSWD=/tmp/passwd
export NSS_WRAPPER_GROUP=/etc/group
```

`passwd.template` には以下が含まれます。

```
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
operator:x:11:0:operator:/root:/sbin/nologin
games:x:12:100:games:/usr/games:/sbin/nologin
ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin
nobody:x:99:99:Nobody:./sbin/nologin
postgres:x:${USER_ID}:${GROUP_ID}:PostgreSQL Server:${HOME}:/bin/bash
```

さらに、これを機能させるには、`nss_wrapper` パッケージおよび `gettext` パッケージをイメージにインストールする必要があります。後者には `envsubst` コマンドが含まれます。たとえば、yum ベースのイメージに対して、以下の行を `Dockerfile` に追加できます。

```
RUN yum -y install nss_wrapper gettext
```

最後に、`Dockerfile` の最後の `USER` 宣言では、ユーザー名ではなく、ユーザー ID (数値) を指定してく

ださい。こうすることで、OpenShift Container Platform が、イメージが実行時に使用しようとしている認証を検証して、root でのイメージの実行を防ぐことができます。特権のあるユーザーがコンテナを実行すると、[セキュリティ上の欠陥が発生する](#) 可能性があるためです。特権のあるユーザーがコンテナを実行すると、セキュリティ上の欠陥が発生す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きます。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 OpenShift Container Platform でビルドできるようにするには、[プロジェクトのビルダーサービスアカウント \(system:serviceaccount:<your-project>:builder\)](#) を **特権付き SCC (security context constraint)** に追加できます。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** できます。

## 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のため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、または移動しても変更され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として共通に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。とくに、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アセンブリー時に共通の依存関係をダウンロードする必要がなくなり、アプリケーションイメージのビルドがスピードアップします。また、すべての依存関係の要件を満たすための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、ダウンストリームイメージをイメージに基づいて作成しなくても、イメージの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される必要があります。単純な設定の場合、実行中のプロセスは環境変数を直接使用できます。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、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は設定ファイルで置き換えることも、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キーなどのシークレットを渡すこともでき、これは推奨されています。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、コンテナイメージレジストリーに漏洩されることはありません。

環境変数を指定することで、イメージの利用者は、イメージ上に新しいレイヤー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。Pod の定義時に環境変数の値を定義するだけで、イメージの再ビルドなしに設定を変更できます。

非常に複雑なシナリオの場合、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指定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場合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において、[イメージ間の通信でのサービスの使用](#) についてのトピックと関連しています。これにより、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、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基にヒープをチューニングする必要があります。

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](#)

### イメージメタデータの設定

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使用が改善され、開発者が OpenShift Container Platform でイメージを使用しやすくなります。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。

### クラスターリング

イメージの複数のインスタンスを実行するとはどういうことかを十分に理解しておく必要があります。最も単純な例では、サービスの負荷分散機能は、イメージのすべての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 での実行時に、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 同士は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ます。そのため、クラスター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推奨されます。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。ログコンテンツを分離する必要がある場合には、出力の接頭辞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

イメージで使用可能な [liveness](#) および [readiness](#) プローブ の例をまとめます。これらのプローブにより、処理の準備ができるまでトラフィックがコンテナにルーティングされず、プロセスが正常でない状態になる場合にコンテナが再起動されるので、ユーザーはイメージを安全にデプロイできます。

### テンプレート

イメージと共に [テンプレート](#) サンプル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、イメージに関連して記述した [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使用が改善され、開発者が OpenShift Container Platform でイメージを使用しやすくなります。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、現在の一連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割り当てるキー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できる点で、環境変数とは異なります。

ラベル値にスペースを含めるには、値を引用符 (") で囲みます。コンマ区切りの値の間にはスペースを入れなくてください。LABEL 命令の詳細は、[Docker ドキュメント](#) を参照してください。

ラベル名には、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
<b>io.openshift.tags</b>	<p>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。タグを使用すると、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で適切なコンテナイメージを提案しやすくなります。</p> <pre> <b>LABEL</b> io.openshift.tags  mongodb,mongodb24,nosql </pre>
<b>io.openshift.wants</b>	<p>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の一覧を指定します。たとえば、コンテナイメージに <b>mysql</b> と <b>redis</b> が必要で、コンテナイメージに <b>redis</b> 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する可能性があります。</p> <pre> <b>LABEL</b> io.openshift.wants  mongodb,redis </pre>

変数	説明
<b>io.k8s.description</b>	<p>このラベルは、コンテナイメージの利用者に、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をエンドユーザーに提供します。</p> <pre data-bbox="518 392 1428 481">LABEL io.k8s.description "The MySQL 5.5 Server with master-slave replication support"</pre>
<b>io.openshift.non-scalable</b>	<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ます。その後、UI はこれをそのイメージのコンシューマーに通知します。スケーリング不可にした場合は基本的に <b>replicas</b> の値を最初に1よりも大きい値に設定することはできません。</p> <pre data-bbox="518 739 1061 795">LABEL io.openshift.non-scalable true</pre>
<b>io.openshift.min-memory</b> および <b>io.openshift.min-cpu</b>	<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 でユーザーに対し、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 1052 1021 1131">LABEL io.openshift.min-memory 8Gi LABEL io.openshift.min-cpu 4</pre>

## 第4章 S2I 要件

### 4.1. 概要

Source-to-Image (S2I) は、アプリケーションのソースコードを入力として取り、アセンブルされたアプリケーションを出力として実行する新規イメージを生成するイメージを簡単に作成できるようにす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。1 つは、[ビルドプロセス](#) で、もう 1 つは [S2I スクリプト](#) です。

### 4.2. ビルドプロセス

ビルドプロセスは、以下の 3 つの要素で設定されており、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。

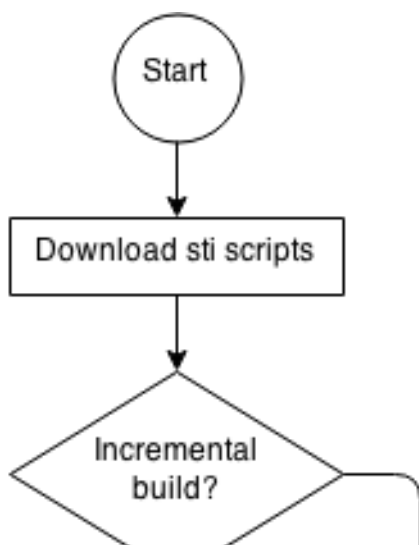
- ソース
- S2I スクリプト
- ビルダーイメージ

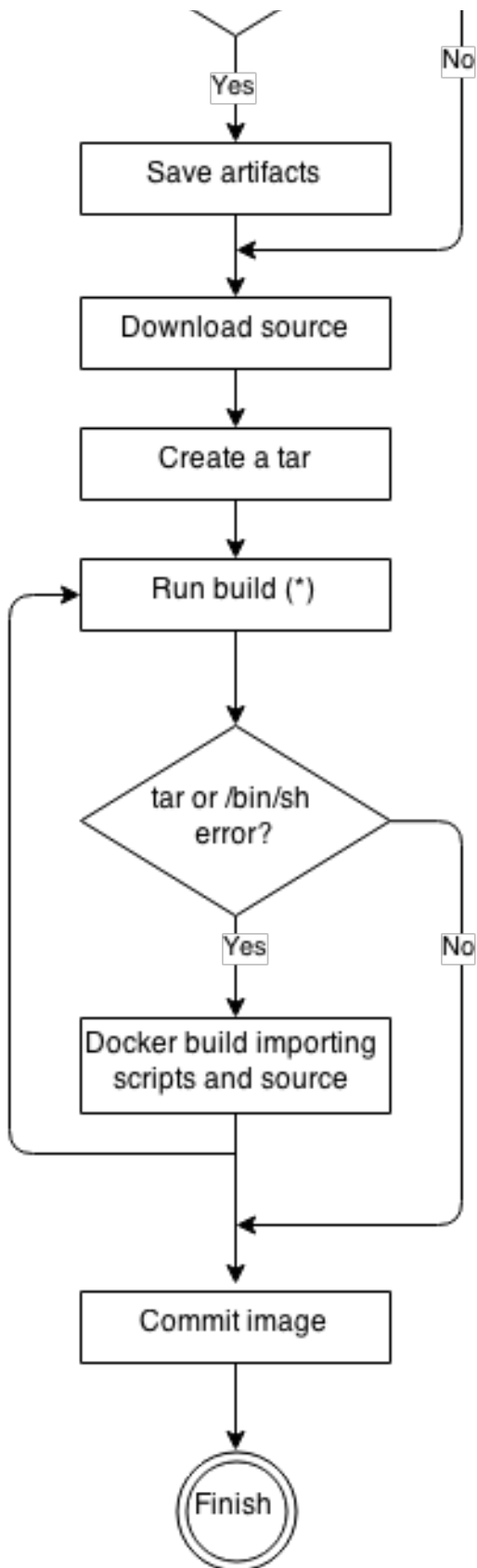
ビルドプロセス中に、S2I はソースとスクリプトをビルダーイメージ内に配置する必要があります。ビルダーイメージ内にソースとスクリプトを配置するために、S2I はソースとスクリプトを含む tar ファイルを作成してから、このファイルをビルダーイメージにストリーミングします。assemble スクリプトを実行する前に、S2I はファイルを展開して、ビルドイメージからコンテンツを **io.openshift.s2i.destination** ラベルで指定される場所に配置します。デフォルトの場所は /tmp ディレクトリーです。

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 に指定される](#) 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"> <li>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)。</li> <li>2. 任意の場所に、アプリケーションソースを配置します。</li> <li>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。</li> <li>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。</li> </ol>
run (必須)	run スクリプトはアプリケーションを実行します。
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すべての依存関係を収集します。以下は例になります。</p> <ul style="list-style-type: none"> <li>● Ruby の場合は、Bundler でインストールされる gems</li> <li>● Java の場合は、.m2 のコンテンツ</li> </ul> <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)	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。

スクリプト	説明
<p><code>test/run</code> (オプション)</p>	<p><code>test/run</code> 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"> <li>1. イメージをビルドします。</li> <li>2. イメージを実行して <code>usage</code> スクリプトを検証します。</li> <li>3. <b>s2i build</b> を実行して <code>assemble</code> スクリプトを検証します。</li> <li>4. 再度 <b>s2i build</b> を実行して、<code>save-artifacts</code> と <code>assemble</code> 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)。</li> <li>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。</li> </ol> <p>詳細は、<a href="#">S2I イメージのテスト</a> のトピックを参照してください。</p> <div data-bbox="517 801 625 999" style="float: left; margin-right: 10px;">  </div> <p><b>注記</b></p> <p><code>test/run</code> 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 <code>test/test-app</code> ディレクトリーです。詳しい情報は、<a href="#">S2I ドキュメント</a> を参照してください。</p>

## 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、公式のコンテナイメージの多くに含まれています。以下に例を示します。

- Ruby
- Node.js
- Python

**ONBUILD** に関する詳細は、[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 ビルダーイメージに **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、[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 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、使用について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` で利用可能にしておく必要があります。必要に応じて、[S2I README](#) のインストール手順に従います。

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る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単純なものである必要がありますが、`assemble` および `run` スクリプトの重要な手順を実行できる必要があります。

### 5.3. スクリプトおよびツールの生成

S2I ツールは、新しい S2I イメージの作成プロセスを加速化する強力な生成ツールと共に提供されます。`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、より使いやすくするために調整する必要がありますが、このスクリプトを開発の開始段階で使用することが推奨されます。



#### 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 イメージテストをローカルに実行する最も簡単な方法として、生成した **Makefile** を使用することができます。

**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ることができます。

### Makefile の例

```
IMAGE_NAME = openshift/ruby-20-centos7
CONTAINER_ENGINE := $(shell command -v podman 2> /dev/null | echo docker)

build:
  ${CONTAINER_ENGINE} build -t $(IMAGE_NAME) .

.PHONY: test
test:
  ${CONTAINER_ENGINE} 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。以下のいずれかのコマンドを実行してください。

- Podman を使用する場合は、以下のコマンドを実行します。

```
$ podman build -t <BUILDER_IMAGE_NAME>_
```

- Docker を使用する場合は、以下のコマンドを実行します。

```
$ 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説明しています。

- usage** スクリプトが機能していることを確認します。

- Podman を使用する場合は、以下のコマンドを実行します。

```
$ podman run <BUILDER_IMAGE_NAME>_ .
```

- Docker を使用する場合は、以下のコマンドを実行します。

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

- 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>_
```

- 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。

- Podman を使用する場合は、以下のコマンドを実行します。

```
$ podman run _<OUTPUT_APPLICATION_IMAGE_NAME>_
```

- Docker を使用する場合は、以下のコマンドを実行します。

```
$ 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 ビルダーイメージを設定する **Dockerfile** と他のアーティファクトが準備できたら、それらを 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。その後は、お使いのリポジトリを参照する [Docker ビルド](#) を定義することのみが必要になります。

OpenShift Container Platform インスタンスが公開 IP アドレスでホストされる場合、ビルドは、S2I ビルダーイメージ 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 ビルダーイメージに基づくアプリケーションの再ビルドをトリガーすることもできます。詳細は、[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](#) の使用により、この要件を満たすことができます。

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、必要な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
<b>BUILD</b>	<b>Build</b> オブジェクト定義のシリアル化された JSON すべて。シリアル化において固有の 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 <a href="#">カスタムストラテジーの仕様</a> で <b>buildAPIVersion</b> パラメーターを設定できます。
<b>SOURCE_REPOSITORY</b>	ビルドするソースが含まれる Git リポジトリの URL
<b>SOURCE_URI</b>	<b>SOURCE_REPOSITORY</b> と同じ値を使用します。どちらでも使用できます。
<b>SOURCE_CONTEXT_DIR</b>	ビルド時に使用する Git リポジトリのサブディレクトリを指定します。定義された場合にのみ表示されます。
<b>SOURCE_REF</b>	ビルドする Git 参照
<b>ORIGIN_VERSION</b>	この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
<b>OUTPUT_REGISTRY</b>	イメージをプッシュするコンテナイメージレジストリー
<b>OUTPUT_IMAGE</b>	ビルドするイメージのコンテナイメージタグ名

変数名	説明
<b>PUSH_DOCKERCFG_PATH</b>	<b>podman push</b> または <b>docker push</b> 操作を実行するためのコンテナレジストリー認証情報へのパス
<b>DOCKER_SOCKET</b>	Docker ソケットの公開がビルド設定で有効にされている場合に Docker ソケットへのパスを指定します ( <b>exposeDockerSocket</b> が <b>true</b> 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作成者は、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ために必要な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、ビルドの入力パラメーターの必要情報をすべて含める。
2. ビルドプロセスを実行する。
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する。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。