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

Last Updated: 2021-04-15

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

法律上の通知

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

第1章 概要

本書では、OpenShift Container Platform で使用可能なコンテナー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 で実行するためにコンテナーイメージを作成する場合には、ユーザーにとってイメージを使用しやすくするために、イメージの作成者は数多くのベストプラクティスを考慮することができます。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は、イメージを OpenShift Container Platform で簡単に使用でき、使用しやすくするのに役立ちます。

2.2. コンテナー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、通常のコンテナー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。アップストリームイメージの **latest** バージョンに互換性を壊す変更が組み込まれている可能性があるため、**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、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、このタグのダウンストリームの利用者が互換性に影響を与え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、ダウンストリームの利用者はいつでも新しいバージョンに移行できますが、これが互換性のない新規イメージによって影響を受けることはありません。**foo:latest** を使用するダウンストリームの利用者の場合、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 つのコンテナー内で起動しないようにしてください。コンテナーは軽量で、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単一 Pod にグループ化して、これらを簡単に共存させ、共同管理できるようになります。

この共存により、コンテナーがネットワークの 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、コンテナー内で実行すると、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、コンテナー全体が停止され、PID 1 プロセスから起動した可能性のある子プロセスが強制的に終了します。

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。

現在のコンテナー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、ファイルを作成したコマンドなどと同じコマンド上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、イメージの階層数が減り、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
コンテナービルダーは **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、上層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、同じイメージの次回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ベストです。

```
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、**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ことができます。

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ようにすることをお勧めします。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、セキュリティー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、コンテナー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。

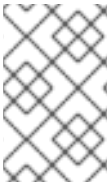
SSHD の回避
イメージでの **SSHD** の実行を回避するようにしてください。ローカルホストで実行中のコンテナーにアクセスするには、**podman exec** または **docker exec** コマンドを使用できます。または、**oc exec** コマンドまたは **oc rsh** コマンドを使用して、OpenShift Container Platform クラスターで実行中のコンテナーにアクセスできます。イメージを使用して **SSHD** をインストール、実行すると、攻撃の経路が増え、セキュリティー修正が必要になります。

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、コンテナー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、コンテナー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ためにボリュームを使用することで、コンテナーが再起動されたり、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ー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。

コンテナー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。コンテナーエンジンは **readonly** フラグをサポートしており、このフラグを使用して、コンテナー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強制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. OPENSHIFT 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ー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ーを実行します。こうすることで、コンテナーエンジンの脆弱性が原因でコンテナー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+rwX /some/directory
```

コンテナーユーザーは常に、root グループのメンバーとして所属しているので、コンテナー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

コンテナー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 などの機密の領域に適用される場合、意図しないユーザーがこのようなファイルを変更することを許可し、コンテナーやホストが公開されてしまう可能性があります。CRI-O は、ランダムなユーザー ID のコンテナーの **/etc/passwd** への挿入をサポートするため、このパーミッションの変更は一切必要ありません。

さらに、コンテナー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。

コンテナーのユーザー 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。この問題に対応するには、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ーのユーザー ID を指定して **passwd** のファイルエントリーを動的に作成できます。

```
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
```

この場合、**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、これが機能するには、**nss_wrapper** 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。後者は **envsubst** コマンドを提供します。たとえば、この行を yum ベースのイメージの **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ぎます。特権のあるユーザーがコンテナーを実行すると、セキュリティー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、コンテナーが停止、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ありません。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、コンテナー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、コンテナー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ー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提供する環境変数としてデータソースなどの設定を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。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、コンテナーの **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。

コンテナーの **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 によるコンテナーイメージの消費が改善され、開発者にとって 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。

クラスタリング
イメージのインスタンスを複数実行するとはどういうことかを完全に理解する必要があります。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推奨されます。OpenShift Container Platform はコンテナー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ーに入り、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ー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ーが再起動されるので、ユーザーは安全にイメージをデプロイできます。

Templates (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 によるコンテナーイメージの消費が改善され、開発者にとって 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ー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ー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。

ラベルの値の中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いでください。**LABEL** 命令についての詳細は 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、コンテナー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ーイメージを提案しやすくなります。 `LABEL io.openshift.tags mongodb,mongodb24,nosql`
io.openshift.wants	コンテナー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、コンテナーイメージが **mysql** と **redis** が必要で、コンテナーイメージに **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する可能性があります。 `LABEL io.openshift.wants mongodb,redis`

変数	説明
io.k8s.description	このラベルは、コンテナー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ー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。 LABEL io.k8s.description "The MySQL 5.5 Server with master-slave replication support"
io.openshift.non-scalable	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 でこのイメージのコンシューマーに通知します。スケーリング不可に指定した場合には、**replicas** の最初の値を 1 よりも大きい値に設定することはできません。 LABEL io.openshift.non-scalable true
io.openshift.min-memory および **io.openshift.min-cpu**	このラベルは、コンテナー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、このコンテナー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。 LABEL io.openshift.min-memory 8Gi LABEL io.openshift.min-cpu 4

第4章 S2I の要件

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インプットとして受け入れるイメージを簡単に作成でき、アウトプットとして、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。

再生成可能なコンテナー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 つを組み合わせて最終的なコンテナーイメージが作成されます。

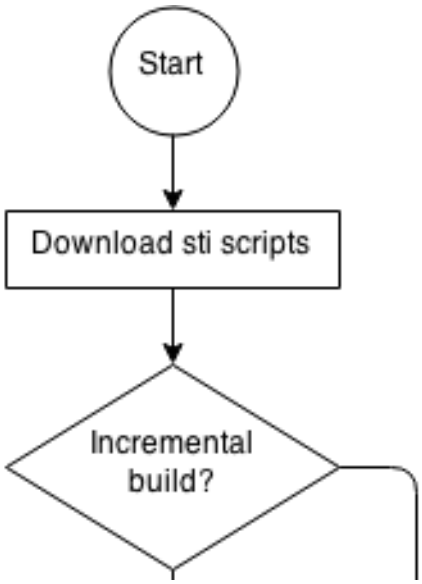
- ソース
- S2I スクリプト
- ビルダーイメージ

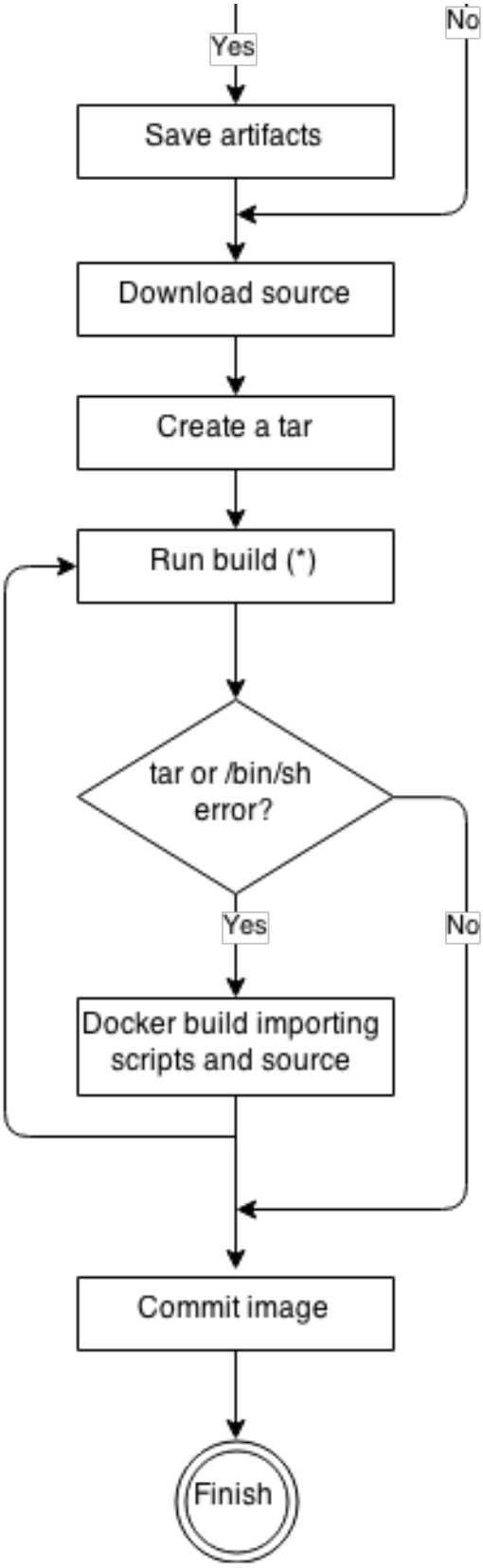
ビルドプロセス中に、S2I はソースとスクリプトをビルダーイメージ内に配置する必要があります。ビルダーイメージ内にソースとスクリプトを配置するために、S2I はソースとスクリプトを含む **tar** ファイルを作成してから、このファイルをビルダーイメージにストリーミングします。**assemble** スクリプトを実行する前に、S2I はファイルを展開して、ビルダーイメージからコンテンツを、デフォルトの場所は **/tmp** ディレクトリーである一方で、**io.openshift.s2i.destination** ラベルが指定する場所に配置します。

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ー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。これらの場所はすべて、ビルドごとに以下の順番にチェックされます。

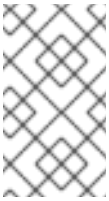
1. BuildConfig で指定するスクリプト
2. アプリケーションソースの **.s2i**/**bin** ディレクトリーにあるスクリプト
3. デフォルトの URL (**io.openshift.s2i.scripts-url** ラベル) にあるスクリプト

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)	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。 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)	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)	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。以下に例を示します。 - 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 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。
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
test/run (オプション)	**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。 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。 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。  **注記** **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ーの **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。

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、以下をはじめとする正式なコンテナー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ー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ーが低い一般的なコンテナー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ー

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 外のコンテナー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、ベースのコンテナー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ー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単純であるはずですが、**assemble** および **run** スクリプトの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 イメージのテストをローカルでテスト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。

1. **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>_ .
```

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、保存して復元する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. コンテナーを実行します。
   - 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. コンテナーが実行され、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が分かります。

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

新規の S2I ビルダー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 リポジトリーに配置して、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ーを参照する 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ている場合には、ビルドは、S2I ビルダーイメージの GitHub リポジトリー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 イメージを基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に対応する特定ビルダーイメージの定義を可能にすることにより、OpenShift Container Platform のカスタムビルドストラテジーは、コンテナーイメージの作成がよく使用されることによって生まれる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ー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ーイメージです。**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
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 リポジトリーの URL
SOURCE_URI	**SOURCE_REPOSITORY** と同じ値を使用します。どちらの値も使用できます。
SOURCE_CONTEXT_DIR	ビルド時に使用する Git リポジトリーのサブディレクトリーを指定します。定義された場合にのみ表示されます。
SOURCE_REF	ビルドする git 参照
ORIGIN_VERSION	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
OUTPUT_REGISTRY	イメージをプッシュするコンテナーイメージレジストリー
OUTPUT_IMAGE	ビルドするイメージのコンテナーイメージタグ名

変数名	説明
PUSH_DOCKERCFG_PATH	**podman push** または **docker push** 操作を実行するためのコンテナーレジストリー認証情報へのパス
DOCKER_SOCKET	Docker ソケットの公開がビルド設定で有効にされている場合に 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。