# OpenShift Dedicated 4

## ビルド

OpenShift Dedicated 4 でのビルドの実行および操作

Last Updated: 2020-05-15

# OpenShift Dedicated 4 ビルド

OpenShift Dedicated 4 でのビルドの実行および操作

## 法律上の通知



## 概要

本書では、OpenShift Dedicated 4 でのビルドおよびビルド設定の概要、およびビルドを実行し、管理するための各種の方法を説明します。

# 目次

# 第1章 イメージビルドについて

## 1.1. ビルド

**ビルド** とは、入力パラメーターを結果として作成されるオブジェクトに変換するプロセスです。ほとんどの場合、このプロセスは入力パラメーターまたはソースコードを実行可能なイメージに変換するために使用されます。**BuildConfig** オブジェクトはビルドプロセス全体の定義です。

OpenShift Dedicated は、ビルドイメージからコンテナーを作成し、それらをコンテナーイメージレジストリーにプッシュして Kubernetes を使用します。

ビルドオブジェクトは共通の特性を共有します。これらには、ビルドの入力、ビルドプロセスの完了についての要件、ビルドプロセスのロギング、正常なビルドからのリリースのパブリッシュ、およびビルドの最終ステータスのパブリッシュが含まれます。ビルドはリソースの制限を利用し、CPU 使用、メモリー使用およびビルドまたは Pod の実行時間などのリソースの制限を指定します。

OpenShift Dedicated ビルドシステムは、ビルド API で指定される選択可能なタイプに基づく **ビルドストラテジー** を幅広くサポートします。利用可能なビルドストラテジーは主に 3 つあります。

- Docker ビルド
- Source-to-Image (S2I) ビルド
- カスタムビルド

デフォルトで、Docker ビルドおよび S2I ビルドがサポートされます。

ビルドの結果作成されるオブジェクトはこれを作成するために使用されるビルダーによって異なります。Docker および S2I ビルドの場合、作成されるオブジェクトは実行可能なイメージです。カスタムビルドの場合、作成されるオブジェクトはビルダーイメージの作成者が指定するものになります。

さらに、Pipeline ビルドストラテジーを使用して、高度なワークフローを実装することができます。

- 継続的インテグレーション
- 継続的デプロイメント

### 1.1.1. Docker ビルド

Docker ビルドストラテジーは docker build コマンドを起動するため、**Dockerfile** とそれに含まれるすべての必要なアーティファクトのあるリポジトリーが実行可能なイメージを生成することを予想します。

### 1.1.2. Source-to-Image (S2I) ビルド

Source-to-Image (S2I) は再現可能な Docker 形式のコンテナーイメージをビルドするためのツールです。これはアプリケーションソースをコンテナーイメージに挿入し、新規イメージをアセンブルして実行可能なイメージを生成します。新規イメージはベースイメージ (ビルダー) とビルドされたソースを組み込み、**buildah run** コマンドで使用することができます。S2I は増分ビルドをサポートします。 これは以前にダウンロードされた依存関係や、以前にビルドされたアーティファクトなどを再利用します。

S2I の利点には以下が含まれます。

| | |
|---|---|
| イメージの柔軟性 | S2I スクリプトを作成して、アプリケーションコードをほとんどすべての既存の Docker 形式コンテナーに挿入し、既存のエコシステムを活用することができます。現時点で S2I は **tar** を使用してアプリケーションソースを挿入するため、イメージは tar が実行されたコンテンツを処理できる必要があることに注意してください。 |
| 速度 | S2I の場合、アセンブルプロセスは、各手順で新規の層を作成せずに多数の複雑な操作を実行でき、これによりプロセスが高速になります。さらに、S2I スクリプトを作成すると、ビルドが実行されるたびにダウンロードまたはビルドを実行することなく、アプリケーションイメージの以前のバージョンに保存されたアーティファクトを再利用できます。 |
| パッチ容易性 (Patchability) | S2I により、基礎となるイメージがセキュリティー上の問題でパッチを必要とする場合にアプリケーションを一貫して再ビルドできるようになります。 |
| 運用効率 | **Dockerfile** が許可するように任意のアクションを実行する代わりにビルド操作を制限することで、PaaS オペレーターはビルドシステムの意図しない、または意図した誤用を避けることができます。 |
| 運用上のセキュリティー | 任意の **Dockerfile** をビルドすると、root の権限昇格のためにホストシステムを公開します。これは、Docker ビルドプロセス全体が Docker 権限を持つユーザーとして実行されるため、悪意あるユーザーが悪用する可能性があります。S2I は root ユーザーとして実行される操作を制限し、スクリプトを root 以外のユーザーとして実行できます。 |
| ユーザー効率 | S2I は開発者が任意の **yum install** タイプの操作を実行することを防ぐため、アプリケーションのビルド時の開発の反復スピードを低下させる可能性があります。 |
| エコシステム | S2I により、アプリケーションのベストプラクティスを利用できるイメージの共有されたエコシステムが促進されます。 |
| 再現性 | 生成されるイメージには、特定バージョンのビルドツールおよび依存関係などのすべての入力が含まれる可能性があります。これにより、イメージを正確に再現できます。 |

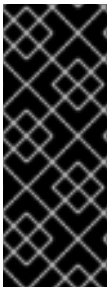
### 1.1.3. カスタムビルド

カスタムビルドストラテジーにより、開発者はビルドプロセス全体を対象とする特定のビルダーイメージを定義できます。独自のビルダーイメージを使用することにより、ビルドプロセスをカスタマイズできます。

カスタムビルダーイメージは、RPM またはベースイメージのビルド用などの、ビルドプロセスロジックで組み込まれた単純な Docker 形式のコンテナーイメージです。

カスタムビルドは非常に高いレベルの権限で実行されるため、デフォルトではユーザーが利用することはできません。クラスター管理者のパーミッションを持つ信頼できるユーザーのみにカスタムビルドを実行するためのアクセスが付与される必要があります。

### 1.1.4. Pipeline ビルド



**重要**

Pipeline ビルドストラテジーは OpenShift Dedicated 4 では非推奨になりました。同等の機能および改善機能は、Tekton をベースとする OpenShift Pipeline にあります。

OpenShift の Jenkins イメージは完全にサポートされており、ユーザーは Jenkins ユーザーのドキュメントに従ってジョブで Jenkinsfile を定義するか、またはこれをソースコントロール管理システムに保存します。

開発者は、Pipeline ビルドストラテジーを利用して Jenkins Pipeline プラグインで実行できるように **Jenkins Pipeline** を定義することができます。このビルドは他のビルドタイプの場合と同様に OpenShift Dedicated での起動、モニタリング、管理が可能です。

Pipeline ワークフローは、ビルド設定に直接組み込むか、または Git リポジトリーに配置してビルド設定で参照して Jenkinsfile で定義します。

# 第2章 ビルド設定について

以下のセクションでは、ビルド、**BuildConfig** の概念を定義し、利用できる主なビルドストラテジーの概要を示します。

## 2.1. BUILDCONFIG

ビルド設定は、単一のビルド定義と新規ビルドを作成するタイミングについてのトリガーを記述します。ビルド設定は **BuildConfig** で定義されます。 BuildConfig は、新規インスタンスを作成するために API サーバーへの POST で使用可能な REST オブジェクトのことです。

**ビルド設定** または **BuildConfig** は、**ビルドストラテジー** と 1 つまたは複数のソースを特徴としています。ストラテジーはプロセスを決定し、ソースは入力内容を提供します。

OpenShift Dedicated を使用したアプリケーションの作成方法の選択に応じて Web コンソールまたは CLI のいずれを使用している場合でも、**BuildConfig** は通常自動的に作成され、いつでも編集できます。**BuildConfig** を構成する部分や利用可能なオプションを理解しておくと、後に設定を手動で変更する場合に役立ちます。

以下の **BuildConfig** の例では、コンテナーイメージのタグやソースコードが変更されるたびに新規ビルドが作成されます。

### BuildConfig のオブジェクト定義

```
kind: BuildConfig
apiVersion: build.openshift.io/v1
metadata:
  name: "ruby-sample-build" (1)
spec:
  runPolicy: "Serial" (2)
  triggers: (3)
    -
      type: "GitHub"
      github:
        secret: "secret101"
    - type: "Generic"
      generic:
        secret: "secret101"
    -
      type: "ImageChange"
  source: (4)
    git:
      uri: "https://github.com/openshift/ruby-hello-world"
  strategy: (5)
    sourceStrategy:
      from:
        kind: "ImageStreamTag"
        name: "ruby-20-centos7:latest"
  output: (6)
    to:
      kind: "ImageStreamTag"
      name: "origin-ruby-sample:latest"
  postCommit: (7)
      script: "bundle exec rake test"
```

1. この仕様は、ruby-sample-build という名前の新規の **BuildConfig** を作成します。

2. **runPolicy** フィールドは、このビルド設定に基づいて作成されたビルドを同時に実行できるかどうかを制御します。デフォルトの値は **Serial** です。 これは新規ビルドが同時にではなく、順番に実行されることを意味します。

3. 新規ビルドを作成するトリガーの一覧を指定できます。

4. **source** セクションでは、ビルドのソースを定義します。ソースの種類は入力の主なソースを決定し、**Git** (コードのリポジトリーの場所を参照)、**Dockerfile** (インラインの Dockerfile からビルド) または **Binary** (バイナリーペイロードを受け入れる) のいずれかとなっています。複数のソースを一度に指定できます。詳細は、各ソースタイプのドキュメントを参照してください。

5. **strategy** セクションでは、ビルドの実行に使用するビルドストラテジーを記述します。ここでは **Source**、**Docker** または **Custom** ストラテジーを指定できます。上記の例では、Source-To-Image がアプリケーションのビルドに使用する **ruby-20-centos7** コンテナーイメージを使用します。

6. コンテナーイメージが正常にビルドされた後に、これは **output** セクションで記述されているリポジトリーにプッシュされます。

7. **postCommit** セクションは、オプションのビルドフック を定義します。

# 第3章 ビルド入力の作成

以下のセクションでは、ビルド入力の概要、ビルドの動作に使用するソースコンテンツを提供するための入力の使用方法、およびビルド環境の使用およびシークレットの作成方法について説明します。

## 3.1. ビルド入力

**ビルド入力** は、ビルドが動作するために必要なソースコンテンツを提供します。以下の **ビルド入力** を使用して OpenShift Dedicated でソースを提供します。以下に優先される順で記載します。

- インラインの Dockerfile 定義
- 既存イメージから抽出したコンテンツ
- Git リポジトリー
- バイナリー (ローカル) 入力
- 入力シークレット
- 外部アーティファクト

複数の異なる入力を単一のビルドにまとめることができます。インラインの Dockerfile が優先されるため、別の入力で指定される Dockerfile という名前の他のファイルは上書きされます。バイナリー (ローカル) 入力および Git リポジトリーは併用できません。

入力シークレットは、ビルド時に使用される特定のリソースや認証情報をビルドで生成される最終アプリケーションイメージで使用不可にする必要がある場合や、Secret リソースで定義される値を使用する必要がある場合に役立ちます。外部アーティファクトは、他のビルド入力タイプのいずれとしても利用できない別のファイルをプルする場合に使用できます。

ビルドを実行すると、以下が行われます。

1. 作業ディレクトリーが作成され、すべての入力内容がその作業ディレクトリーに配置されます。たとえば、入力 Git リポジトリーのクローンはこの作業ディレクトリーに作成され、入力イメージから指定されたファイルはターゲットのパスを使用してこの作業ディレクトリーにコピーされます。
2. ビルドプロセスによりディレクトリーが **contextDir** に変更されます (定義されている場合)。
3. インライン Dockerfile がある場合は、現在のディレクトリーに書き込まれます。
4. 現在の作業ディレクトリーにある内容が Dockerfile、カスタムビルダーのロジック、または **assemble** スクリプトが参照するビルドプロセスに提供されます。つまり、ビルドでは **contextDir** 内にない入力コンテンツは無視されます。

以下のソース定義の例には、複数の入力タイプと、入力タイプの統合方法の説明が含まれています。それぞれの入力タイプの定義方法に関する詳細は、各入力タイプについての個別のセクションを参照してください。

```
source:
  git:
    uri: https://github.com/openshift/ruby-hello-world.git (1)
  images:
  - from:
      kind: ImageStreamTag
```

```
    name: myinputimage:latest
    namespace: mynamespace
  paths:
  - destinationDir: app/dir/injected/dir (2)
    sourcePath: /usr/lib/somefile.jar
 contextDir: "app/dir" (3)
 dockerfile: "FROM centos:7\nRUN yum install -y httpd" (4)
```

1. 作業ディレクトリーにクローンされるビルド用のリポジトリー
2. **myinputimage** の **/usr/lib/somefile.jar** は、 **<workingdir>/app/dir/injected/dir** に保存されます。
3. ビルドの作業ディレクトリーは **<original_workingdir>/app/dir** になります。
4. このコンテンツを含む Dockerfile は **<original_workingdir>/app/dir** に作成され、この名前が指定された既存ファイルは上書きされます。

## 3.2. DOCKERFILE ソース

**dockerfile** の値が指定されると、このフィールドの内容は、**Dockerfile** という名前のファイルとしてディスクに書き込まれます。これは、他の入力ソースが処理された後に実行されるので、入力ソースリポジトリーの root ディレクトリーに **Dockerfile** が含まれる場合は、これはこの内容で上書きされます。

ソースの定義は **BuildConfig** の **spec** セクションに含まれます。

```
source:
  dockerfile: "FROM centos:7\nRUN yum install -y httpd" (1)
```

1. **dockerfile** フィールドには、ビルドされるインライン Dockerfile が含まれます。

**追加リソース**

- このフィールドは、通常は **Dockerfile** を Docker ストラテジービルドに指定するために使用されます。

## 3.3. イメージソース

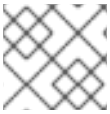
追加のファイルは、イメージを使用してビルドプロセスに渡すことができます。インプットイメージは **From** および **To** イメージターゲットが定義されるのと同じ方法で参照されます。つまり、コンテナーイメージと imagestreamtags の両方を参照できます。イメージとの関連で、1つまたは複数のパスのペアを指定して、ファイルまたはディレクトリーのパスを示し、イメージと宛先をコピーしてビルドコンテキストに配置する必要があります。

ソースパスは、指定したイメージ内の絶対パスで指定してください。宛先は、相対ディレクトリーパスでなければなりません。ビルド時に、イメージは読み込まれ、指定のファイルおよびディレクトリーはビルドプロセスのコンテキストディレクトリーにコピーされます。これは、ソースリポジトリーのコンテンツ (ある場合) のクローンが作成されるディレクトリーと同じです。ソースパスの末尾は **/.** であり、ディレクトリーのコンテンツがコピーされますが、ディレクトリー自体は宛先で作成されません。

イメージの入力は、**BuildConfig** の **source** の定義で指定します。

```
source:
  git:
    uri: https://github.com/openshift/ruby-hello-world.git
  images: (1)
  - from: (2)
      kind: ImageStreamTag
      name: myinputimage:latest
      namespace: mynamespace
    paths: (3)
    - destinationDir: injected/dir (4)
      sourcePath: /usr/lib/somefile.jar (5)
  - from:
      kind: ImageStreamTag
      name: myotherinputimage:latest
      namespace: myothernamespace
    pullSecret: mysecret (6)
    paths:
    - destinationDir: injected/dir
      sourcePath: /usr/lib/somefile.jar
```

1. 1 つ以上のインプットイメージおよびファイルの配列
2. コピーされるファイルが含まれるイメージへの参照
3. ソース/宛先パスの配列
4. ビルドプロセスで対象のファイルにアクセス可能なビルドルートへの相対パス
5. 参照イメージの中からコピーするファイルの場所
6. 認証情報がインプットイメージにアクセスするのに必要な場合に提供されるオプションのシークレット



**注記**

この機能は、カスタムストラテジーを使用するビルドについてサポートされません。

## 3.4. GIT ソース

ソースコードは、指定されている場合は指定先の場所からフェッチされます。

インラインの Dockerfile を指定する場合は、これにより Git リポジトリーの **contextDir** 内にある **Dockerfile** (ある場合) が上書きされます。

ソースの定義は **BuildConfig** の **spec** セクションに含まれます。

```
source:
  git: (1)
    uri: "https://github.com/openshift/ruby-hello-world"
    ref: "master"
  contextDir: "app/dir" (2)
  dockerfile: "FROM openshift/ruby-22-centos7\nUSER example" (3)
```

1. **git** フィールドには、ソースコードのリモート git リポジトリーへの URI が含まれます。オプションで、**ref** フィールドを指定して特定の git 参照をチェックアウトします。SHA1 タグまたはブラン

2. **contextDir** フィールドでは、ビルドがアプリケーションのソースコードを検索する、ソースコードのリポジトリー内のデフォルトの場所を上書きできます。アプリケーションがサブディレクトリーに存在する場合には、このフィールドを使用してデフォルトの場所 (root フォルダー) を上書きすることができます。

3. オプションの **dockerfile** フィールドがある場合は、Dockerfile を含む文字列を指定してください。 この文字列は、ソースリポジトリーに存在する可能性のある Dockerfile を上書きします。

**ref** フィールドにプル要求が記載されている場合には、システムは **git fetch** 操作を使用して **FETCH_HEAD** をチェックアウトします。

**ref** の値が指定されていない場合は、OpenShift Dedicated はシャロークローン (**--depth=1**) を実行します。この場合、デフォルトのブランチ (通常は **master**) での最新のコミットに関連するファイルのみがダウンロードされます。これにより、リポジトリーのダウンロード時間が短縮されます (詳細のコミット履歴はありません)。指定リポジトリーのデフォルトのブランチで完全な **git clone** を実行するには、**ref** をデフォルトのブランチ名に設定します (例: **master**)。

**警告**

中間者 (MITM) TLS ハイジャックまたはプロキシーされた接続の再暗号化を実行するプロキシーを通過する Git クローンの操作は機能しません。

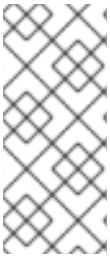
### 3.4.1. プロキシーの使用

プロキシーの使用によってのみ Git リポジトリーにアクセスできる場合は、使用するプロキシーを **BuildConfig** の **source** セクションで定義できます。HTTP および HTTPS プロキシーの両方を設定できます。いずれのフィールドもオプションです。**NoProxy** フィールドで、プロキシーを実行しないドメインを指定することもできます。

**注記**

実際に機能させるには、ソース URI で HTTP または HTTPS プロトコルを使用する必要があります。

```
source:
  git:
    uri: "https://github.com/openshift/ruby-hello-world"
    httpProxy: http://proxy.example.com
    httpsProxy: https://proxy.example.com
    noProxy: somedomain.com, otherdomain.com
```



**注記**

Pipeline ストラテジーのビルドの場合には、現在 Jenkins の Git プラグインに制約があるので、Git プラグインを使用する Git の操作では **BuildConfig** に定義された HTTP または HTTPS プロキシーは使用されません。Git プラグインは、Jenkins UI の Plugin Manager パネルで設定されたプロキシーのみを使用します。どのジョブであっても、Jenkins 内の git のすべての対話にはこのプロキシーが使用されます。

**追加リソース**

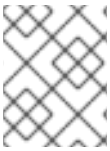
- Jenkins UI でのプロキシーの設定方法については、JenkinsBehindProxyを参照してください。

### 3.4.2. ソースクローンシークレットの追加

ビルダー Pod には、ビルドのソースとして定義された git リポジトリーへのアクセスが必要です。ソースクローンのシークレットは、ビルダー Pod に対し、プライベートリポジトリーや自己署名証明書または信頼されていない SSL 証明書が設定されたリポジトリーなどの通常アクセスできないリポジトリーへのアクセスを提供するために使用されます。

**前提条件**

- 以下は、サポートされているソースクローンのシークレット設定です。
  - .gitconfig ファイル
  - Basic 認証
  - SSH キー認証
  - 信頼されている認証局



**注記**

特定のニーズに対応するために、これらの設定の組み合わせを使用することもできます。

**手順**

ビルドは **builder** サービスアカウントで実行されます。 この builder アカウントには、使用するソースクローンのシークレットに対するアクセスが必要です。

- 以下のコマンドを実行してアクセスを付与します。

```
$ oc secrets link builder mysecret
```

**注記**

シークレットを参照しているサービスアカウントにのみにシークレットを制限することはデフォルトで無効にされています。つまり、マスターの設定ファイルで **serviceAccountConfig.limitSecretReferences** がマスター設定の **false** (デフォルトの設定) に設定されている場合は、サービスにシークレットをリンクする必要はありません。

#### 3.4.2.1. ソースクローンシークレットのビルド設定への自動追加

**BuildConfig** が作成されると、OpenShift Dedicated はソースクローンのシークレット参照を自動生成します。この動作により、追加の設定なしに、作成される **Builds** が参照される **Secret** に保存された認証情報を自動的に使用できるようになり、リモート git リポジトリーに対する認証が可能になります。

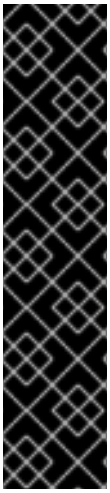
この機能を使用するには、git リポジトリーの認証情報を含む **Secret** が **BuildConfig** が後に作成される namespace になければなりません。この **Secret** には、プレフィックス**build.openshift.io/source-secret-match-uri-** で開始するアノテーション 1 つ以上含まれている必要もあります。これらの各アノテーションの値には、以下で定義される URI パターンを指定します。ソースクローンのシークレット参照なしに **BuildConfig** が作成され、git ソースの URI が **Secret** アノテーションの URI パターンと一致する場合に、OpenShift Dedicated はその **Secret** への参照を **BuildConfig** に自動的に挿入します。

### 前提条件

URI パターンには以下を含める必要があります。

- 有効なスキーム (***://**、**git://**、**http://**、**https://** または **ssh://**)
- ホスト (*、有効なホスト名、またはオプションで ***.** が先頭に指定された IP アドレス)
- パス (/* または、/ の後に * などの文字が後に続く文字列)

上記のいずれの場合でも、* 文字はワイルドカードと見なされます。



**重要**

URI パターンは、RFC3986に準拠する Git ソースの URI と一致する必要があります。URI パターンにユーザー名 (またはパスワード) のコンポーネントを含めないようにしてください。

たとえば、git リポジトリーの URL に **ssh://git@bitbucket.atlassian.com:7999/ATLASSIAN/jira.git** を使用する場合に、ソースのシークレットは **ssh://bitbucket.atlassian.com:7999/*** として指定する必要があります (**ssh://git@bitbucket.atlassian.com:7999/*** ではありません)。

```
$ oc annotate secret mysecret \
    'build.openshift.io/source-secret-match-uri-1=ssh://bitbucket.atlassian.com:7999/*'
```

### 手順

複数の **Secrets** が特定の **BuildConfig** の Git URI と一致する場合は、OpenShift Dedicated は一致する文字列が一番長いシークレットを選択します。これは、以下の例のように基本的な上書きを許可します。

以下の部分的な例では、ソースクローンのシークレットの一部が 2 つ表示されています。1 つ目は、HTTPS がアクセスする **mycorp.com** ドメイン内のサーバーに一致しており、2 つ目は **mydev1.mycorp.com** および **mydev2.mycorp.com** のサーバーへのアクセスを上書きします。

```
kind: Secret
apiVersion: v1
metadata:
  name: matches-all-corporate-servers-https-only
  annotations:
    build.openshift.io/source-secret-match-uri-1: https://*.mycorp.com/*
data:
  ...
```

```
kind: Secret
apiVersion: v1
metadata:
  name: override-for-my-dev-servers-https-only
  annotations:
    build.openshift.io/source-secret-match-uri-1: https://mydev1.mycorp.com/*
    build.openshift.io/source-secret-match-uri-2: https://mydev2.mycorp.com/*
data:
  ...
```

- 以下のコマンドを使用して、**build.openshift.io/source-secret-match-uri-** アノテーションを既存のシークレットに追加します。

```
$ oc annotate secret mysecret \
    'build.openshift.io/source-secret-match-uri-1=https://*.mycorp.com/*'
```

#### 3.4.2.2. ソースクローンシークレットの手動による追加

ソースクローンのシークレットは、ビルド設定に手動で追加できます。 **sourceSecret** フィールドを **BuildConfig** 内の **source** セクションに追加してから、作成した **secret** の名前に設定して実行できます (この例では **basicsecret**)。

```
apiVersion: "v1"
kind: "BuildConfig"
metadata:
  name: "sample-build"
spec:
  output:
    to:
      kind: "ImageStreamTag"
      name: "sample-image:latest"
  source:
    git:
      uri: "https://github.com/user/app.git"
    sourceSecret:
      name: "basicsecret"
  strategy:
    sourceStrategy:
      from:
        kind: "ImageStreamTag"
        name: "python-33-centos7:latest"
```

**手順**

**oc set build-secret** コマンドを使用して、既存のビルド設定にソースクローンのシークレットを設定することも可能です。

- 既存のビルド設定にソースクローンシークレットを設定するには、以下を実行します。

```
$ oc set build-secret --source bc/sample-build basicsecret
```

**追加リソース**

- **BuildConfig** にシークレットを定義すると、このトピックの詳細情報を表示できます。

### 3.4.2.3. .gitconfig ファイルからのシークレットの作成

アプリケーションのクローンが **.gitconfig** ファイルに依存する場合、そのファイルが含まれるシークレットを作成できます。これをビルダーサービスアカウントおよび **BuildConfig**に追加します。

#### 手順

- **.gitconfig** ファイルからシークレットを作成するには、以下を実行します。

```
$ oc create secret generic <secret_name> --from-file=<path/to/.gitconfig>
```

#### 注記

**.gitconfig** ファイルの **http** セクションが **sslVerify=false** に設定されている場合は、SSL 検証をオフにすることができます。

```
[http]
        sslVerify=false
```

### 3.4.2.4. セキュリティー保護された Git の .gitconfig ファイルからのシークレットの作成

Git サーバーが 2 方向の SSL、ユーザー名とパスワードでセキュリティー保護されている場合には、ソースビルドに証明書ファイルを追加して、**.gitconfig** ファイルに証明書ファイルへの参照を追加する必要があります。

#### 前提条件

- Git 認証情報

#### 手順

ソースビルドに証明書ファイルを追加して、**.gitconfig** ファイルに証明書ファイルへの参照を追加します。

1. **client.crt**、**cacert.crt** および **client.key** ファイルをアプリケーションのソースコードの **/var/run/secrets/openshift.io/source/** フォルダーに追加します。

2. サーバーの **.gitconfig** ファイルに、以下の例のように **[http]** セクションを追加します。

```
# cat .gitconfig
[user]
        name = <name>
        email = <email>
[http]
        sslVerify = false
        sslCert = /var/run/secrets/openshift.io/source/client.crt
        sslKey = /var/run/secrets/openshift.io/source/client.key
        sslCaInfo = /var/run/secrets/openshift.io/source/cacert.crt
```

3. シークレットを作成します。

```
$ oc create secret generic <secret_name> \
--from-literal=username=<user_name> \ 1
--from-literal=password=<password> \ 2
```

```
--from-file=.gitconfig=.gitconfig \
--from-file=client.crt=/var/run/secrets/openshift.io/source/client.crt \
--from-file=cacert.crt=/var/run/secrets/openshift.io/source/cacert.crt \
--from-file=client.key=/var/run/secrets/openshift.io/source/client.key
```

1. ユーザーの git ユーザー名
2. このユーザーのパスワード

**重要**

パスワードを再度入力しなくてもよいように、ビルドに S2I イメージを指定するようにしてください。ただし、リポジトリーをクローンできない場合には、ビルドをプロモートするためにユーザー名とパスワードを指定する必要があります。

**追加リソース**

- アプリケーションソースコードの **/var/run/secrets/openshift.io/source/** フォルダー。

### 3.4.2.5. ソースコードの基本的な認証からのシークレットの作成

Basic 認証では、SCM サーバーに対して認証する場合に **--username** と **--password** の組み合わせ、または **token** が必要です。

**前提条件**

- プライベートリポジトリーにアクセスするためのユーザー名およびパスワード。

**手順**

1. **secret** を先に作成してから、プライベートリポジトリーにアクセスするためにユーザー名とパスワードを使用してください。

```
$ oc create secret generic <secret_name> \
    --from-literal=username=<user_name> \
    --from-literal=password=<password> \
    --type=kubernetes.io/basic-auth
```

2. トークンで Basic 認証のシークレットを作成します。

```
$ oc create secret generic <secret_name> \
    --from-literal=password=<token> \
    --type=kubernetes.io/basic-auth
```

### 3.4.2.6. ソースコードの SSH キー認証からのシークレットの作成

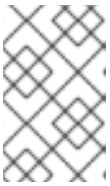
SSH キーベースの認証では、プライベート SSH キーが必要です。

リポジトリーのキーは通常 **$HOME/.ssh/** ディレクトリーにあり、デフォルトで **id_dsa.pub**、**id_ecdsa.pub**、**id_ed25519.pub** または **id_rsa.pub** という名前が付けられています。

**手順**

1. SSH キーの認証情報を生成します。

```
$ ssh-keygen -t rsa -C "your_email@example.com"
```



**注記**

SSH キーのパスフレーズを作成すると、OpenShift Dedicated でビルドができなくなります。パスフレーズを求めるプロンプトが出されても、ブランクのままにします。

パブリックキーと、それに対応するプライベートキーのファイルが 2 つ作成されます (**id_dsa**、**id_ecdsa**、**id_ed25519** または **id_rsa** のいずれか)。これらが両方設定されたら、パブリックキーのアップロード方法についてソースコントロール管理 (SCM) システムのマニュアルを参照してください。プライベートキーは、プライベートリポジトリーにアクセスするために使用されます。

2. SSH キーを使用してプライベートリポジトリーにアクセスする前に、シークレットを作成します。

```
$ oc create secret generic <secret_name> \
    --from-file=ssh-privatekey=<path/to/ssh/private/key> \
    --type=kubernetes.io/ssh-auth
```

#### 3.4.2.7. ソースコードの信頼されている認証局からのシークレットの作成

**git clone** の操作時に信頼される TLS 認証局 (CA) のセットは OpenShift Dedicated インフラストラクチャーイメージにビルドされます。Git サーバーが自己署名の証明書を使用するか、イメージで信頼されていない認証局によって署名された証明書を使用する場合には、その証明書が含まれるシークレットを作成するか、TLS 検証を無効にしてください。

**CA 証明書** のシークレットを作成した場合に、OpenShift Dedicated はその証明書を使用して、**git clone** 操作時に Git サーバーにアクセスします。存在する TLS 証明書をどれでも受け入れてしまう Git の SSL 検証の無効化に比べ、この方法を使用するとセキュリティーレベルが高くなります。

**手順**

- CA 証明書ファイルでシークレットを作成します。

  a. CA が中間認証局を使用する場合には、**ca.crt** ファイルにすべての CA の証明書を統合します。次のコマンドを実行します。

```
$ cat intermediateCA.crt intermediateCA.crt rootCA.crt > ca.crt
```

  b. シークレットを作成します。

```
$ oc create secret generic mycert --from-file=ca.crt=</path/to/file> 1
```

  1 **ca.crt** というキーの名前を使用する必要があります。

#### 3.4.2.8. ソースシークレットの組み合わせ

特定のニーズに対応するために上記の方法を組み合わせてソースクローンのシークレットを作成することができます。

#### 3.4.2.8.1. .gitconfig ファイルでの SSH ベースの認証シークレットの作成

SSH ベースの認証シークレットと **.gitconfig** ファイルなど、特定のニーズに応じてソースクローンシークレットを作成するための複数の異なる方法を組み合わせることができます。

**前提条件**

- SSH 認証
- .gitconfig ファイル

**手順**

- **.gitconfig** ファイルを使って SSH ベースの認証シークレットを作成します。

```
$ oc create secret generic <secret_name> \
    --from-file=ssh-privatekey=<path/to/ssh/private/key> \
    --from-file=<path/to/.gitconfig> \
    --type=kubernetes.io/ssh-auth
```

#### 3.4.2.8.2. .gitconfig ファイルと CA 証明書を組み合わせるシークレットの作成

**.gitconfig** ファイルおよび CA 証明書を組み合わせるシークレットなど、特定のニーズに応じてソースクローンシークレットを作成するための複数の異なる方法を組み合わせることができます。

**前提条件**

- .gitconfig ファイル
- CA 証明書

**手順**

- **.gitconfig** ファイルと CA 証明書を組み合わせるシークレットを作成します。

```
$ oc create secret generic <secret_name> \
    --from-file=ca.crt=<path/to/certificate> \
    --from-file=<path/to/.gitconfig>
```

#### 3.4.2.8.3. CA 証明書ファイルを使用した Basic 認証のシークレットの作成

Basic 認証および CA 証明書を組み合わせるシークレットなど、特定のニーズに応じてソースクローンシークレットを作成するための複数の異なる方法を組み合わせることができます。

**前提条件**

- Basic 認証の認証情報
- CA 証明書

**手順**

- CA 証明書ファイルを使って Basic 認証のシークレットを作成します。

```
$ oc create secret generic <secret_name> \
    --from-literal=username=<user_name> \
    --from-literal=password=<password> \
    --from-file=ca-cert=</path/to/file> \
    --type=kubernetes.io/basic-auth
```

#### 3.4.2.8.4. .gitconfig ファイルを使用した Basic 認証シークレットの作成

Basic 認証および **.gitconfig** ファイルを組み合わせるシークレットなど、特定のニーズに応じてソースクローンシークレットを作成するための複数の異なる方法を組み合わせることができます。

**前提条件**

- Basic 認証の認証情報
- **.gitconfig** ファイル

**手順**

- **.gitconfig** ファイルを使って Basic 認証のシークレットを作成します。

```
$ oc create secret generic <secret_name> \
    --from-literal=username=<user_name> \
    --from-literal=password=<password> \
    --from-file=</path/to/.gitconfig> \
    --type=kubernetes.io/basic-auth
```

#### 3.4.2.8.5. .gitconfig ファイルと CA 証明書を使用した Basic 認証シークレットの作成

Basic 認証、**.gitconfig** ファイルおよび CA 証明書を組み合わせるシークレットなど、特定のニーズに応じてソースクローンシークレットを作成するための複数の異なる方法を組み合わせることができます。

**前提条件**

- Basic 認証の認証情報
- **.gitconfig** ファイル
- CA 証明書

**手順**

- **.gitconfig** ファイルと CA 証明書を使用して Basic 認証シークレットを作成します。

```
$ oc create secret generic <secret_name> \
    --from-literal=username=<user_name> \
    --from-literal=password=<password> \
    --from-file=</path/to/.gitconfig> \
    --from-file=ca-cert=</path/to/file> \
    --type=kubernetes.io/basic-auth
```

-

## 3.5. バイナリー (ローカル) ソース

ローカルのファイルシステムからビルダーにコンテンツをストリーミングすることは、**Binary** タイプのビルドと呼ばれています。このビルドについての **BuildConfig.spec.source.type** の対応する値は **Binary** です。

このソースタイプは、**oc start-build** のみをベースとして使用される点で独特なタイプです。

**注記**

バイナリータイプのビルドでは、ローカルファイルシステムからコンテンツをストリーミングする必要があります。そのため、バイナリーファイルが提供されないので、バイナリータイプのビルドを自動的にトリガーすること (例: イメージの変更トリガーなど) はできません。同様に、Web コンソールからバイナリータイプのビルドを起動することはできません。

バイナリービルドを使用するには、以下のオプションのいずれかを指定して **oc start-build** を呼び出します。

- **--from-file**: 指定したファイルのコンテンツはバイナリーストリームとしてビルダーに送信されます。ファイルに URL を指定することもできます。次に、ビルダーはそのデータをビルドコンテキストの上に、同じ名前のファイルに保存します。

- **--from-dir** および **--from-repo**: コンテンツはアーカイブされて、バイナリーストリームとしてバイナリーに送信されます。次に、ビルダーはビルドコンテキストディレクトリー内にアーカイブのコンテンツを展開します。**--from-dir** を使用して、展開されるアーカイブに URL を指定することもできます。

- **--from-archive**: 指定したアーカイブはビルダーに送信され、ビルドコンテキストディレクトリーに展開されます。このオプションは **--from-dir** と同様に動作しますが、このオプションの引数がディレクトリーの場合には常にアーカイブがホストに最初に作成されます。

上記のそれぞれの例では、以下のようになります。

- **BuildConfig** に **Binary** のソースタイプが定義されている場合には、これは事実上無視され、クライアントが送信する内容に置き換えられます。

- **BuildConfig** に **Git** のソースタイプが定義されている場合には、**Binary** と **Git** は併用できないので、動的に無効にされます。 この場合、ビルダーに渡されるバイナリーストリームのデータが優先されます。

ファイル名ではなく、HTTP または HTTPS スキーマを使用する URL を **--from-file** や **--from-archive** に渡すことができます。**--from-file** で URL を指定すると、ビルダーイメージのファイル名は Web サーバーが送信する **Content-Disposition** ヘッダーか、ヘッダーがない場合には URL パスの最後のコンポーネントによって決定されます。認証形式はどれもサポートされておらず、カスタムのTLS 証明書を使用したり、証明書の検証を無効にしたりできません。

**oc new-build --binary=true** を使用すると、バイナリービルドに関連する制約が実施されるようになります。作成される **BuildConfig** のソースタイプは **Binary** になります。 つまり、この **BuildConfig** のビルドを実行するための唯一の有効な方法は、**--from** オプションのいずれかを指定して **oc start-build** を使用し、必須のバイナリーデータを提供する方法になります。

**dockerfile** および **contextDir** のソースオプションは、バイナリービルドに関して特別な意味を持ちます。

**dockerfile** はバイナリービルドソースと合わせて使用できます。**dockerfile** を使用し、バイナリーストリームがアーカイブの場合には、そのコンテンツはアーカイブにある Dockerfile の代わりとして機能します。**dockerfile** が **--from-file** の引数と合わせて使用されている場合には、ファイルの引数は **dockerfile** となり、**dockerfile** の値はバイナリーストリームの値に置き換わります。

バイナリーストリームが展開されたアーカイブのコンテンツをカプセル化する場合には、**contextDir** フィールドの値はアーカイブ内のサブディレクトリーと見なされます。 有効な場合には、ビルド前にビルダーがサブディレクトリーに切り替わります。

## 3.6. 入力シークレットおよび CONFIGMAP

シナリオによっては、ビルド操作で、依存するリソースにアクセスするための認証情報や他の設定データが必要になる場合がありますが、この情報をソースコントロールに配置するのは適切ではありません。この場合は、**入力シークレット** および **input ConfigMap** を定義することができます。

たとえば、Maven を使用して Java アプリケーションをビルドする場合、プライベートキーを使ってアクセスされる Maven Central または JCenter のプライベートミラーをセットアップできます。そのプライベートミラーからライブラリーをダウンロードするには、以下を指定する必要があります。

1. ミラーの URL および接続の設定が含まれる **settings.xml** ファイル。
2. **~/.ssh/id_rsa** などの、設定ファイルで参照されるプライベートキー。

セキュリティー上の理由により、認証情報はアプリケーションイメージで公開しないでください。

以下の例は Java アプリケーションについて説明していますが、**/etc/ssl/certs** ディレクトリー、API キーまたはトークン、ラインセンスファイルなどに SSL 証明書を追加する場合に同じ方法を使用できます。

### 3.6.1. 入力シークレットおよび ConfigMap の追加

シナリオによっては、ビルド操作で、依存するリソースにアクセスするための認証情報や他の設定データが必要になる場合がありますが、この情報をソースコントロールに配置するのは適切ではありません。この場合は、**入力シークレット** および **input ConfigMap** を定義することができます。

**手順**

既存の **BuildConfig** に入力シークレットおよび/または ConfigMap を追加するには、以下を行います。

1. ConfigMap がない場合は作成します。

```
$ oc create configmap settings-mvn \
    --from-file=settings.xml=<path/to/settings.xml>
```

これにより、**settings-mvn** という名前の新しい ConfigMap が作成されます。 これには、**settings.xml** ファイルのプレーンテキストのコンテンツが含まれます。

2. シークレットがない場合は作成します。

```
$ oc create secret generic secret-mvn \
    --from-file=id_rsa=<path/to/.ssh/id_rsa>
```

これにより、**secret-mvn** という名前の新規シークレットが作成されます。 これには、**id_rsa** プライベートキーの base64 でエンコードされたコンテンツが含まれます。

3. ConfigMap およびシークレットを既存の **BuildConfig** の **source** セクションに追加します。

```
source:
  git:
    uri: https://github.com/wildfly/quickstart.git
  contextDir: helloworld
  configMaps:
    - configMap:
        name: settings-mvn
  secrets:
    - secret:
        name: secret-mvn
```

シークレットおよび ConfigMap を新規の **BuildConfig** に追加するには、以下のコマンドを実行します。

```
$ oc new-build \
    openshift/wildfly-101-centos7~https://github.com/wildfly/quickstart.git \
    --context-dir helloworld --build-secret “secret-mvn” \
    --build-config-map "settings-mvn"
```

ビルド時に、**settings.xml** および **id_rsa** ファイルはソースコードが配置されているディレクトリーにコピーされます。OpenShift Dedicated S2I ビルダーイメージでは、これはイメージの作業ディレクトリーで、**Dockerfile** の **WORKDIR** の指示を使用して設定されます。別のディレクトリーを指定するには、 **destinationDir** を定義に追加します。

```
source:
  git:
    uri: https://github.com/wildfly/quickstart.git
  contextDir: helloworld
  configMaps:
    - configMap:
        name: settings-mvn
      destinationDir: ".m2"
  secrets:
    - secret:
        name: secret-mvn
      destinationDir: ".ssh"
```

新規の **BuildConfig** を作成時に、宛先のディレクトリーを指定することも可能です。

```
$ oc new-build \
    openshift/wildfly-101-centos7~https://github.com/wildfly/quickstart.git \
    --context-dir helloworld --build-secret “secret-mvn:.ssh” \
    --build-config-map "settings-mvn:.m2"
```

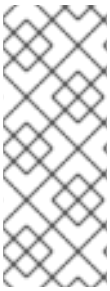
いずれの場合も、**settings.xml** ファイルがビルド環境の **./.m2** ディレクトリーに追加され、**id_rsa** キーは **./.ssh** ディレクトリーに追加されます。

### 3.6.2. Source-to-Image ストラテジー

**Source** ストラテジーを使用すると、定義された入力シークレットはすべて、適切な **destinationDir** にコピーされます。**destinationDir** を空にすると、シークレットはビルダーイメージの作業ディレクトリーに配置されます。

**destinationDir** が相対パスの場合に同じルールが使用されます。シークレットは、イメージの作業ディ

レクトリーに対する相対的なパスに配置されます。**destinationDir** パスの最終ディレクトリーは、ビルダーイメージにない場合に作成されます。**destinationDir** の先行するすべてのディレクトリーは存在している必要があり、そうでない場合にはエラーが生じます。



**注記**

入力シークレットは全ユーザーに書き込み権限が割り当てられた状態で追加され (**0666** のパーミッション)、**assemble** スクリプトの実行後には、サイズが 0 になるように切り捨てられます。つまり、シークレットファイルは作成されたイメージ内に存在はしますが、セキュリティーの関係上、空になります。

入力 ConfigMap は、**assemble** スクリプトの実行後に切り捨てられません。

### 3.6.3. Docker ストラテジー

**Docker** ストラテジーを使用すると、Dockerfileで **ADD** および **COPY** の命令 を使用してコンテナーイメージに定義されたすべての入力シークレットを追加できます。

シークレットの **destinationDir** を指定しない場合は、ファイルは、**Dockerfile** が配置されているのと同じディレクトリーにコピーされます。相対パスを **destinationDir** として指定する場合は、シークレットは、**Dockerfile** と相対的なディレクトリーにコピーされます。これにより、ビルド時に使用するコンテキストディレクトリーの一部として、Docker ビルド操作でシークレットファイルが利用できるようになります。

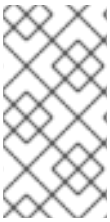
#### シークレットおよび ConfigMap データを参照する Dockerfile の例

```
FROM centos/ruby-22-centos7

USER root
COPY ./secret-dir /secrets
COPY ./config /

# Create a shell script that will output secrets and ConfigMaps when the image is run
RUN echo '#!/bin/sh' > /input_report.sh
RUN echo '(test -f /secrets/secret1 && echo -n "secret1=" && cat /secrets/secret1)' >>
/input_report.sh
RUN echo '(test -f /config && echo -n "relative-configMap=" && cat /config)' >> /input_report.sh
RUN chmod 755 /input_report.sh

CMD ["/bin/sh", "-c", "/input_report.sh"]
```



**注記**

通常はシークレットがイメージから実行するコンテナーに置かれないように、入力シークレットを最終的なアプリケーションイメージから削除する必要があります。ただし、シークレットは追加される階層のイメージ自体に存在します。この削除は、**Dockerfile** の一部として組み込まれる必要があります。

### 3.6.4. カスタムストラテジー

**Custom** ストラテジーを使用する場合、定義された入力シークレットおよび ConfigMap はすべて、**/var/run/secrets/openshift.io/build** ディレクトリー内のビルダーコンテナーで入手できます。カスタムのビルドイメージは、これらのシークレットおよび ConfigMap を適切に使用する必要がありま

す。また、**Custom** ストラテジーを使用すると、カスタムストラテジーのオプションで記載されているようにシークレットを定義できます。

既存のストラテジーのシークレットと入力シークレットには違いはありません。ただし、ビルダーイメージはこれらを区別し、、ビルドのユースケースに基づいてこれらを異なる方法で使用する場合があります。

入力シークレットは常に **/var/run/secrets/openshift.io/build** ディレクトリーにマウントされます。そうでない場合には、ビルダーが完全なビルドオブジェクトを含む **$BUILD** 環境変数を解析できます。

## 3.7. 外部アーティファクト

ソースリポジトリーにバイナリーファイルを保存することは推奨していません。そのため、ビルドプロセス中に追加のファイル (Java **.jar** の依存関係など) をプルするビルドを定義する必要がある場合があります。この方法は、使用するビルドストラテジーにより異なります。

**Source** ビルドストラテジーの場合は、**assemble** スクリプトに適切なシェルコマンドを設定する必要があります。

**.s2i/bin/assemble ファイル**

```
#!/bin/sh
APP_VERSION=1.0
wget http://repository.example.com/app/app-$APP_VERSION.jar -O app.jar
```

**.s2i/bin/run ファイル**

```
#!/bin/sh
exec java -jar app.jar
```

**Docker** ビルドストラテジーの場合は、**Dockerfile** を変更して、**RUN** 命令を指定してシェルコマンドを呼び出す必要があります。

**Dockerfile の抜粋**

```
FROM jboss/base-jdk:8

ENV APP_VERSION 1.0
RUN wget http://repository.example.com/app/app-$APP_VERSION.jar -O app.jar

EXPOSE 8080
CMD [ "java", "-jar", "app.jar" ]
```

I実際には、ファイルの場所の環境変数を使用し、**Dockerfile** または **assemble** スクリプトを更新するのではなく、**BuildConfig** で定義した環境変数で、ダウンロードする特定のファイルをカスタマイズすることができます。

環境変数の定義には複数の方法があり、いずれかの方法を選択できます。

- **.s2i/environment**ファイルの使用 (ソースビルドストラテジーのみ)
- **BuildConfig** での設定
- **oc start-build --env** を使用した明示的な指定 (手動でトリガーされるビルドのみが対象)

## 3.8. プライベートレジストリーでの DOCKER 認証情報の使用

プライベートコンテナーレジストリーの有効な認証情報を指定して、**.docker/config.json** ファイルでビルドを提供できます。これにより、プライベートコンテナーレジストリーにアウトプットイメージをプッシュしたり、認証を必要とするプライベートコンテナーイメージレジストリーからビルダーイメージをプルすることができます。

**注記**

OpenShift Dedicated コンテナーイメージレジストリーでは、OpenShift Dedicated が自動的にシークレットを生成するので、この作業は必要ありません。

デフォルトでは、**.docker/config.json** ファイルはホームディレクトリーにあり、以下の形式となっています。

```
auths:
  https://index.docker.io/v1/: 1
    auth: "YWRfbGzhcGU6R2labnRib21ifTE=" 2
    email: "user@example.com" 3
```

1. レジストリーの URL
2. 暗号化されたパスワード
3. ログイン用のメールアドレス

このファイルに複数のコンテナーイメージレジストリーを定義できます。または **docker login** コマンドを実行して、このファイルに認証エントリーを追加することも可能です。ファイルが存在しない場合には作成されます。

Kubernetes では **Secret** オブジェクトが提供され、これを使用して設定とパスワードを保存することができます。

**前提条件**

- **.docker/config.json** ファイル

**手順**

1. ローカルの **.docker/config.json** ファイルからシークレットを作成します。

```
$ oc create secret generic dockerhub \
    --from-file=.dockerconfigjson=<path/to/.docker/config.json> \
    --type=kubernetes.io/dockerconfigjson
```

   このコマンドにより、**dockerhub** という名前のシークレットの JSON 仕様が生成され、オブジェクトが作成されます。

2. シークレットが作成されたら、これをビルダーサービスアカウントに追加します。ビルドは **builder** ロールで実行されるので、以下のコマンドでシークレットへのアクセスを設定する必要があります。

```
$ oc secrets link builder dockerhub
```

3. **pushSecret** フィールドを **BuildConfig** の **output** セクションに追加し、作成した **secret** の名前 (上記の例では、**dockerhub**) に設定します。

```
spec:
  output:
    to:
      kind: "DockerImage"
      name: "private.registry.com/org/private-image:latest"
    pushSecret:
      name: "dockerhub"
```

**oc set build-secret** コマンドを使用して、ビルド設定にプッシュするシークレットを設定します。

```
$ oc set build-secret --push bc/sample-build dockerhub
```

4. ビルドストラテジー定義に含まれる **pullSecret** を指定して、プライベートコンテナーイメージレジストリーからビルダーコンテナーイメージをプルします。

```
strategy:
  sourceStrategy:
    from:
      kind: "DockerImage"
      name: "docker.io/user/private_repository"
    pullSecret:
      name: "dockerhub"
```

**oc set build-secret** コマンドを使用して、ビルド設定でプルシークレットを設定します。

```
$ oc set build-secret --pull bc/sample-build dockerhub
```

**注記**

以下の例では、ソールビルドに **pullSecret** を使用しますが、Docker とカスタムビルドにも該当します。

## 3.9. ビルド環境

Pod 環境変数と同様に、ビルドの環境変数は Downward API を使用して他のリソースや変数の参照として定義できます。ただし、いくつかは例外があります。

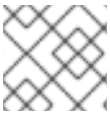
**oc set env** コマンドで、**BuildConfig** に定義した環境変数を管理することも可能です。

**注記**

参照はコンテナーの作成前に解決されるため、ビルド環境変数の **valueFrom** を使用したコンテナーリソースの参照はサポートされません。

### 3.9.1. 環境変数としてのビルドフィールドの使用

ビルドオブジェクトの情報は、値を取得するフィールドの **JsonPath** に、**fieldPath** 環境変数のソースを設定することで挿入できます。



**注記**

Jenkins Pipeline ストラテジーは、環境変数の **valueFrom** 構文をサポートしません。

**手順**

- 値を取得するフィールドの **JsonPath** に、**fieldPath** 環境変数のソースを設定します。

```
env:
  - name: FIELDREF_ENV
    valueFrom:
      fieldRef:
        fieldPath: metadata.name
```

### 3.9.2. 環境変数としてのシークレットの使用

**valueFrom** 構文を使用して、シークレットからのキーの値を環境変数として利用できます。

**手順**

- シークレットを環境変数として使用するには、**valueFrom** 構文を設定します。

```
apiVersion: v1
kind: BuildConfig
metadata:
  name: secret-example-bc
spec:
  strategy:
    sourceStrategy:
      env:
      - name: MYVAL
        valueFrom:
          secretKeyRef:
            key: myval
            name: mysecret
```

## 3.10. ビルドの概要

**Secret** オブジェクトタイプはパスワード、OpenShift Dedicated クライアント設定ファイル、**dockercfg** ファイル、プライベートソースリポジトリーの認証情報などの機密情報を保持するメカニズムを提供します。シークレットは機密内容を Pod から切り離します。シークレットはボリュームプラグインを使用してコンテナーにマウントすることも、システムが Pod の代わりにシークレットを使用して各種アクションを実行することもできます。

**YAML シークレットオブジェクト定義**

```
apiVersion: v1
kind: Secret
metadata:
  name: test-secret
  namespace: my-namespace
type: Opaque (1)
data: (2)
```

```
  username: dmFsdWUtMQ0K (3)
  password: dmFsdWUtMg0KDQo=
stringData: (4)
  hostname: myapp.mydomain.com (5)
```

1. シークレットにキー名および値の構造を示しています。
2. **data** フィールドでキーに使用できる形式は、Kubernetes identifiers glossary の **DNS_SUBDOMAIN** 値のガイドラインに従う必要があります。
3. **data** マップのキーに関連付けられる値は base64 でエンコーディングされている必要があります。
4. **stringData** マップのエントリーが base64 に変換され、このエントリーは自動的に **data** マップに移動します。このフィールドは書き込み専用です。 この値は **data** フィールドでのみ返されます。
5. **stringData** マップのキーに関連付けられた値は単純なテキスト文字列で構成されます。

### 3.10.1. シークレットのプロパティー

キーのプロパティーには以下が含まれます。

- シークレットデータはその定義とは別に参照できます。
- シークレットデータのボリュームは一時ファイルストレージ機能 (tmpfs) でサポートされ、ノードで保存されることはありません。
- シークレットデータは namespace 内で共有できます。

### 3.10.2. シークレットの種類

**type** フィールドの値で、シークレットのキー名と値の構造を指定します。このタイプを使用して、シークレットオブジェクトにユーザー名とキーの配置を実行できます。検証の必要がない場合には、デフォルト設定の **opaque** タイプを使用してください。

以下のタイプから 1 つ指定して、サーバー側で最小限の検証をトリガーし、シークレットデータに固有のキー名が存在することを確認します。

- **kubernetes.io/service-account-token**。サービスアカウントトークンを使用します。
- **kubernetes.io/dockercfg**。必須の Docker 認証には **.dockercfg** ファイルを使用します。
- **kubernetes.io/dockerconfigjson**。必須の Docker 認証には **.docker/config.json** ファイルを使用します。
- **kubernetes.io/basic-auth**。Basic 認証で使用します。
- **kubernetes.io/ssh-auth**。SSH キー認証で使用します。
- **kubernetes.io/tls**。TLS 認証局で使用します。

検証の必要がない場合には **type= Opaque** と指定します。これは、シークレットがキー名または値の規則に準拠しないという意味です。**opaque** シークレットでは、任意の値を含む、体系化されていない **key:value** ペアも利用できます。

**注記**

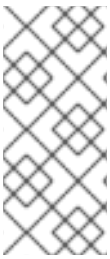
**example.com/my-secret-type** などの他の任意のタイプを指定できます。これらのタイプはサーバー側では実行されませんが、シークレットの作成者がその種類のキー/値の要件に従う意図があることを示します。

### 3.10.3. シークレットの更新

シークレットの値を変更する場合、値 (すでに実行されている Pod で使用される値) は動的に変更されません。シークレットを変更するには、元の Pod を削除してから新規の Pod を作成する必要があります (同じ PodSpec を使用する場合があります)。

シークレットの更新は、新規コンテナーイメージのデプロイと同じワークフローで実行されます。**kubectl rolling-update** コマンドを使用できます。

シークレットの **resourceVersion** 値は参照時に指定されません。したがって、シークレットが Pod の起動と同じタイミングで更新される場合、Pod に使用されるシークレットのバージョンは定義されません。



**注記**

現時点で、Pod の作成時に使用されるシークレットオブジェクトのリソースバージョンを確認することはできません。今後はコントローラーが古い **resourceVersion** を使用して再起動できるよう Pod がこの情報を報告できるようにすることが予定されています。それまでは既存シークレットのデータを更新せずに別の名前で新規のシークレットを作成します。

### 3.10.4. シークレットの作成

シークレットに依存する Pod を作成する前に、シークレットを作成する必要があります。

シークレットの作成時に以下を実行します。

- シークレットデータでシークレットオブジェクトを作成します。
- Pod のサービスアカウントをシークレットの参照を許可するように更新します。
- シークレットを環境変数またはファイルとして使用する Pod を作成します (**secret** ボリュームを使用)。

**手順**

- 作成コマンドを使用して JSON または YAML ファイルのシークレットオブジェクトを作成できます。

```
$ oc create -f <filename>
```

たとえば、ローカルの **.docker/config.json** ファイルからシークレットを作成できます。

```
$ oc create secret generic dockerhub \
    --from-file=.dockerconfigjson=<path/to/.docker/config.json> \
    --type=kubernetes.io/dockerconfigjson
```

このコマンドにより、**dockerhub** という名前のシークレットの JSON 仕様が生成され、オブジェクトが作成されます。

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque (1)
data:
  username: dXNlci1uYW1l
  password: cGFzc3dvcmQ=
```

1. **opaque** シークレットを指定します。

```
apiVersion: v1
kind: Secret
metadata:
  name: aregistrykey
  namespace: myapps
type: kubernetes.io/dockerconfigjson (1)
data:

.dockerconfigjson:bm5ubm5ubm5ubm5ubm5ubm5ubm5ubmdnZ2dnZ2dnZ2dnZ2dnZ2dnZ2cg
YXV0aCBrZXIzCg== (2)
```

1. シークレットが Docker 設定の JSON ファイルを使用することを指定します。
2. Docker 設定 JSON ファイルを base64 でエンコードした出力

#### 3.10.4.1. シークレットの使用

シークレットの作成後に、Pod を作成してシークレットを参照し、ログを取得し、Pod を削除することができます。

**手順**

1. シークレットを参照する Pod を作成します。

```
$ oc create -f <your_yaml_file>.yaml
```

2. ログを取得します。

```
$ oc logs secret-example-pod
```

3. Pod を削除します。

```
$ oc delete pod secret-example-pod
```

**追加リソース**

- シークレットデータを含む YAML ファイルのサンプル

**4 つのファイルを作成する YAML シークレット**

```
apiVersion: v1
kind: Secret
metadata:
  name: test-secret
data:
  username: dmFsdWUtMQ0K      (1)
  password: dmFsdWUtMQ0KDQo=  (2)
stringData:
  hostname: myapp.mydomain.com  (3)
  secret.properties: |-         (4)
    property1=valueA
    property2=valueB
```

1. デコードされる値が含まれるファイル
2. デコードされる値が含まれるファイル
3. 提供される文字列が含まれるファイル
4. 提供されるデータが含まれるファイル

**シークレットデータと共にボリュームのファイルが設定された Pod の YAML**

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-example-pod
spec:
  containers:
    - name: secret-test-container
      image: busybox
      command: [ "/bin/sh", "-c", "cat /etc/secret-volume/*" ]
      volumeMounts:
          # name must match the volume name below
          - name: secret-volume
            mountPath: /etc/secret-volume
            readOnly: true
  volumes:
    - name: secret-volume
      secret:
        secretName: test-secret
  restartPolicy: Never
```

**シークレットデータと共に環境変数が設定された Pod の YAML**

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-example-pod
spec:
  containers:
    - name: secret-test-container
      image: busybox
```

```
      command: [ "/bin/sh", "-c", "export" ]
      env:
        - name: TEST_SECRET_USERNAME_ENV_VAR
          valueFrom:
            secretKeyRef:
              name: test-secret
              key: username
  restartPolicy: Never
```

**シークレットデータと環境変数が投入されたビルド設定の YAML**

```
apiVersion: v1
kind: BuildConfig
metadata:
  name: secret-example-bc
spec:
  strategy:
    sourceStrategy:
      env:
      - name: TEST_SECRET_USERNAME_ENV_VAR
        valueFrom:
          secretKeyRef:
            name: test-secret
            key: username
```

## 3.11. サービス提供証明書のシークレット

サービスが提供する証明書のシークレットは、追加設定なしの証明書を必要とする複雑なミドルウェアアプリケーションをサポートするように設計されています。これにはノードおよびマスターの管理者ツールで生成されるサーバー証明書と同じ設定が含まれます。

### 手順

サービスとの通信のセキュリティーを保護するには、クラスターが署名された提供証明書/キーペアを namespace のシークレットに生成できるようにします。

- 値をシークレットに使用する名前に設定し、 **service.alpha.openshift.io/serving-cert-secret-name** アノテーションをサービスに設定します。
  次に、**PodSpec** はそのシークレットをマウントできます。これが利用可能な場合、Pod が実行されます。この証明書は内部サービス DNS 名、 **<service.name>.<service.namespace>.svc** に適しています。

  証明書およびキーは PEM 形式であり、それぞれ **tls.crt** および **tls.key** に保存されます。証明書/キーのペアは有効期限に近づくと自動的に置換されます。シークレットの **service.alpha.openshift.io/expiry** アノテーションで RFC3339 形式の有効期限の日付を確認します。

**注記**

ほとんどの場合、サービス DNS 名 **<service.name>.<service.namespace>.svc** は外部にルート可能ではありません。**<service.name>.<service.namespace>.svc** の主な使用方法として、クラスターまたはサービス間の通信用として、 re-encrypt ルートで使用されます。

他の Pod は Pod に自動的にマウントされる **/var/run/secrets/kubernetes.io/serviceaccount/service-ca.crt** ファイルの CA バンドルを使用して、クラスターで作成される証明書 (内部 DNS 名の場合にのみ署名される) を信頼できます。

この機能の署名アルゴリズムは **x509.SHA256WithRSA** です。ローテーションを手動で実行するには、生成されたシークレットを削除します。新規の証明書が作成されます。

## 3.12. シークレットの制限

シークレットを使用するには、Pod がシークレットを参照できる必要があります。シークレットは、以下の 3 つの方法で Pod で使用されます。

- コンテナーの環境変数を事前に設定するために使用される。
- 1 つ以上のコンテナーにマウントされるボリュームのファイルとして使用される。
- Pod のイメージをプルする際に kubelet によって使用される。

ボリュームタイプのシークレットは、ボリュームメカニズムを使用してデータをファイルとしてコンテナーに書き込みます。 **imagePullSecrets** は、シークレットを namespace のすべての Pod に自動的に挿入するためにサービスアカウントを使用します。

テンプレートにシークレット定義が含まれる場合、テンプレートで指定のシークレットを使用できるようにするには、シークレットのボリュームソースを検証し、指定されるオブジェクト参照が **Secret** タイプのオブジェクトを実際に参照していることを確認できる必要があります。そのため、シークレットはこれに依存する Pod の作成前に作成されている必要があります。最も効果的な方法として、サービスアカウントを使用してシークレットを自動的に挿入することができます。

シークレット API オブジェクトは namespace にあります。それらは同じ namespace の Pod によってのみ参照されます。

個々のシークレットは 1MB のサイズに制限されます。これにより、apiserver および kubelet メモリーを使い切るような大規模なシークレットの作成を防ぐことができます。ただし、小規模なシークレットであってもそれらを数多く作成するとメモリーの消費につながります。

# 第4章 ビルド出力の管理

ビルド出力の概要およびビルド出力の管理方法についての説明については、以下のセクションを使用します。

## 4.1. ビルド出力

**Docker** または **Source-to-Image (S2I)** ストラテジーを使用するビルドにより、新しいコンテナーイメージが作成されます。このイメージは、**Build** 仕様の **output** セクションで指定されているコンテナーイメージのレジストリーにプッシュされます。

出力の種類が **ImageStreamTag** の場合は、イメージが統合された OpenShift Dedicated レジストリーにプッシュされ、指定のイメージストリームにタグ付けされます。出力が **DockerImage** タイプの場合は、出力参照の名前が Docker のプッシュ仕様として使用されます。この仕様にレジストリーが含まれる場合もありますが、レジストリーが指定されていない場合は、DockerHub にデフォルト設定されます。ビルド仕様の出力セクションが空の場合には、ビルドの最後にイメージはプッシュされません。

### ImageStreamTag への出力

```
spec:
  output:
    to:
      kind: "ImageStreamTag"
      name: "sample-image:latest"
```

### Docker のプッシュ仕様への出力

```
spec:
  output:
    to:
      kind: "DockerImage"
      name: "my-registry.mycompany.com:5000/myimages/myimage:tag"
```

## 4.2. アウトプットイメージの環境変数

**Docker** および **Source-to-Image (S2I)** ストラテジービルドは、以下の環境変数をアウトプットイメージに設定します。

| 変数 | 説明 |
|---|---|
| **OPENSHIFT_BUILD_NAME** | ビルドの名前 |
| **OPENSHIFT_BUILD_NAMESPACE** | ビルドの namespace |
| **OPENSHIFT_BUILD_SOURCE** | ビルドのソース URL |
| **OPENSHIFT_BUILD_REFERENCE** | ビルドで使用する git 参照 |
| **OPENSHIFT_BUILD_COMMIT** | ビルドで使用するソースコミット |

また、**S2I**] または **Docker** ストラテジーオプションなどで設定されたユーザー定義の環境変数も、アウトプットイメージの環境変数一覧の一部になります。

## 4.3. アウトプットイメージのラベル

**Docker** および **Source-to-Image (S2I)** ビルドは、以下のラベルをアウトプットイメージに設定します。

| ラベル | 説明 |
| --- | --- |
| **io.openshift.build.commit.author** | ビルドで使用するソースコミットの作成者 |
| **io.openshift.build.commit.date** | ビルドで使用するソースコミットの日付 |
| **io.openshift.build.commit.id** | ビルドで使用するソースコミットのハッシュ |
| **io.openshift.build.commit.message** | ビルドで使用するソースコミットのメッセージ |
| **io.openshift.build.commit.ref** | ソースに指定するブランチまたは参照 |
| **io.openshift.build.source-location** | ビルドのソース URL |

**BuildConfig.spec.output.imageLabels** フィールドを使用して、カスタムラベルの一覧を指定することも可能です。 このラベルは、**BuildConfig** の各イメージビルドに適用されます。

**ビルドイメージに適用されるカスタムラベル**

```
spec:
  output:
    to:
      kind: "ImageStreamTag"
      name: "my-image:latest"
    imageLabels:
    - name: "vendor"
      value: "MyCompany"
    - name: "authoritative-source-url"
      value: "registry.mycompany.com"
```

# 第5章 ビルドストラテジーの使用

以下のセクションでは、主なサポートされているビルドストラテジー、およびそれらの使用方法を定義します。

## 5.1. DOCKER ビルド

Docker ビルドストラテジーは docker build コマンドを起動するため、**Dockerfile** とそれに含まれるすべての必要なアーティファクトのあるリポジトリーが実行可能なイメージを生成することを予想します。

### 5.1.1. Dockerfile FROM イメージの置き換え

**Dockerfile** の **FROM** 命令は、**BuildConfig** の **from** に置き換えられます。**Dockerfile** がマルチステージビルドを使用する場合、最後の **FROM** 命令のイメージを置き換えます。

**手順**

**Dockerfile** の **FROM** 命令は、**BuildConfig** の **from** に置き換えられます。

```
strategy:
  dockerStrategy:
    from:
      kind: "ImageStreamTag"
      name: "debian:latest"
```

### 5.1.2. Dockerfile パスの使用

デフォルトで、Docker ビルドは **BuildConfig.spec.source.contextDir** フィールドで指定されたコンテキストのルートに配置されている Dockerfile (名前付きの **Dockerfile**) を使用します。

**dockerfilePath** フィールドでは、ビルドが異なるパスを使用して Dockerfile ファイルの場所 (**BuildConfig.spec.source.contextDir** フィールドへの相対パス) を特定できます。デフォルトの **Dockerfile** (例: **MyDockerfile**) とは異なるファイル名や、サブディレクトリーにある Dockerfile へのパス (例: **dockerfiles/app1/Dockerfile**) などを設定できます。

**手順**

ビルドが Dockerfile を見つけるために異なるパスを使用できるように **dockerfilePath** フィールドを使用するには、以下を設定します。

```
strategy:
  dockerStrategy:
    dockerfilePath: dockerfiles/app1/Dockerfile
```

### 5.1.3. Docker 環境変数の使用

環境変数を Docker ビルドプロセスおよび結果として生成されるイメージで利用可能にするには、環境変数を **BuildConfig** の **dockerStrategy** 定義に追加できます。

ここに定義した環境変数は、Dockerfile 内で後に参照できるよう単一の **ENV** Dockerfile 命令として **FROM** 命令の直後に挿入されます。

**手順**

変数はビルド時に定義され、アウトプットイメージに残るため、そのイメージを実行するコンテナーにも存在します。

たとえば、ビルドやランタイム時にカスタムの HTTP プロキシーを定義するには以下を設定します。

```
dockerStrategy:
...
  env:
    - name: "HTTP_PROXY"
      value: "http://myproxy.net:5187/"
```

**oc set env** コマンドで、**BuildConfig** に定義した環境変数を管理することも可能です。

### 5.1.4. Docker ビルド引数の追加

**BuildArgs** 配列を使用して Docker ビルド引数 を設定できます。ビルド引数は、ビルドの開始時に Docker に渡されます。

#### 手順

Docker ビルドの引数を設定するには、以下のように **BuildArgs** 配列にエントリーを追加します。これは、**BuildConfig** の **dockerStrategy** 定義の中にあります。以下は例になります。

```
dockerStrategy:
...
  buildArgs:
    - name: "foo"
      value: "bar"
```

## 5.2. SOURCE-TO-IMAGE (S2I) ビルド

Source-to-Image (S2I) は再現可能な Docker 形式のコンテナーイメージをビルドするためのツールです。これはアプリケーションソースをコンテナーイメージに挿入し、新規イメージをアセンブルして実行可能なイメージを生成します。新規イメージはベースイメージ (ビルダー) とビルドされたソースを組み込み、**buildah run** コマンドで使用することができます。S2I は増分ビルドをサポートします。 これは以前にダウンロードされた依存関係や、以前にビルドされたアーティファクトなどを再利用します。

S2I の利点には以下が含まれます。

| | |
|---|---|
| イメージの柔軟性 | S2I スクリプトを作成して、アプリケーションコードをほとんどすべての既存の Docker 形式コンテナーに挿入し、既存のエコシステムを活用することができます。現時点で S2I は **tar** を使用してアプリケーションソースを挿入するため、イメージは tar が実行されたコンテンツを処理できる必要があることに注意してください。 |
| 速度 | S2I の場合、アセンブルプロセスは、各手順で新規の層を作成せずに多数の複雑な操作を実行でき、これによりプロセスが高速になります。さらに、S2I スクリプトを作成すると、ビルドが実行されるたびにダウンロードまたはビルドを実行することなく、アプリケーションイメージの以前のバージョンに保存されたアーティファクトを再利用できます。 |
| パッチ容易性 (Patchability) | S2I により、基礎となるイメージがセキュリティー上の問題でパッチを必要とする場合にアプリケーションを一貫して再ビルドできるようになります。 |

| | |
|---|---|
| 運用効率 | **Dockerfile** が許可するように任意のアクションを実行する代わりにビルド操作を制限することで、PaaS オペレーターはビルドシステムの意図しない、または意図した誤用を避けることができます。 |
| 運用上のセキュリティー | 任意の **Dockerfile** をビルドすると、root の権限昇格のためにホストシステムを公開します。これは、Docker ビルドプロセス全体が Docker 権限を持つユーザーとして実行されるため、悪意あるユーザーが悪用する可能性があります。S2I は root ユーザーとして実行される操作を制限し、スクリプトを root 以外のユーザーとして実行できます。 |
| ユーザー効率 | S2I は開発者が任意の **yum install** タイプの操作を実行することを防ぐため、アプリケーションのビルド時の開発の反復スピードを低下させる可能性があります。 |
| エコシステム | S2I により、アプリケーションのベストプラクティスを利用できるイメージの共有されたエコシステムが促進されます。 |
| 再現性 | 生成されるイメージには、特定バージョンのビルドツールおよび依存関係などのすべての入力が含まれる可能性があります。これにより、イメージを正確に再現できます。 |

### 5.2.1. Source-to-Image (S2I) 増分ビルドの実行

S2I は増分ビルドを実行できます。 つまり、以前にビルドされたイメージからアーティファクトが再利用されます。

#### 手順

増分ビルドを作成するには、ストラテジー定義に以下の変更を加えて **BuildConfig** を作成します。

```
strategy:
  sourceStrategy:
    from:
      kind: "ImageStreamTag"
      name: "incremental-image:latest" (1)
    incremental: true (2)
```

1. 増分ビルドをサポートするイメージを指定します。この動作がサポートされているか判断するには、ビルダーイメージのドキュメントを参照してください。
2. このフラグでは、増分ビルドを試行するかどうかを制御します。ビルダーイメージで増分ビルドがサポートされていない場合は、ビルドは成功しますが、**save-artifacts** スクリプトがないため、増分ビルドに失敗したというログメッセージが表示されます。

#### 追加リソース

- 増分ビルドをサポートするビルダーイメージを作成する方法の詳細については、S2I 要件について参照してください。

### 5.2.2. Source-to-Image (S2I) ビルダーイメージスクリプトの上書き

ビルダーイメージによって提供される **assemble**、 **run**、および **save-artifacts** S2I スクリプトを上書きできます。

### 手順

ビルダーイメージによって提供される **assemble**、 **run**、および **save-artifacts** S2I スクリプトを上書きするには、以下のいずれかを実行します。

1. アプリケーションのソースリポジトリーの **.s2i/bin** ディレクトリーに **assemble**, **run** または **save-artifacts** スクリプトを指定します。
2. ストラテジー定義の一部として、スクリプトを含むディレクトリーの URL を指定します。以下は例になります。

```
strategy:
  sourceStrategy:
    from:
      kind: "ImageStreamTag"
      name: "builder-image:latest"
    scripts: "http://somehost.com/scripts_directory" 1
```

1 このパスに、**run**, **assemble** および **save-artifacts** が追加されます。一部または全スクリプトがある場合、そのスクリプトが、イメージに指定された同じ名前のスクリプトの代わりに使用されます。

**注記**

**scripts** URL にあるファイルは、ソースリポジトリーの **.s2i/bin** にあるファイルよりも優先されます。

## 5.2.3. Source-to-Image (S2I) 環境変数

ソースビルドのプロセスと生成されるイメージで環境変数を利用できるようにする方法として、2 種類 (環境ファイルおよび **BuildConfig** 環境の値) あります。 指定される変数は、ビルドプロセスでアウトプットイメージに表示されます。

### 5.2.3.1. Source-to-Image (S2I) 環境ファイルの使用

ソースビルドでは、ソースリポジトリーの **.s2i/environment** ファイルに指定することで、アプリケーション内に環境の値 (1 行に 1 つ) を設定できます。このファイルに指定される環境変数は、ビルドプロセス時にアウトプットイメージに表示されます。

ソースリポジトリーに **.s2i/environment** ファイルを渡すと、S2I はビルド時にこのファイルを読み取ります。これにより **assemble** スクリプトがこれらの変数を使用できるので、ビルドの動作をカスタマイズできます。

### 手順

たとえば、Rails アプリケーションのアセットコンパイルを無効にするには、以下を実行します。

- **DISABLE_ASSET_COMPILATION=true** を **.s2i/environment** ファイルに追加します。

ビルド以外に、指定の環境変数も実行中のアプリケーション自体で利用できます。たとえば、Rails アプリケーションが **production** ではなく **development** モードで起動できるようにするには、以下を実行します。

- **RAILS_ENV=development** を **.s2i/environment** ファイルに追加します。

**追加リソース**

- サポートされる環境変数の完全なリストについては、各イメージのイメージの使用についてのセクションを参照してください。

#### 5.2.3.2. Source-to-Image (S2I) BuildConfig 環境の使用

環境変数を **BuildConfig** の **sourceStrategy** 定義に追加できます。ここに定義されている環境変数は、**assemble** スクリプトの実行時に表示され、アウトプットイメージで定義されるので、**run** スクリプトやアプリケーションコードでも利用できるようになります。

**手順**

- たとえば、Rails アプリケーションのアセットコンパイルを無効にするには、以下を実行します。

```
sourceStrategy:
...
  env:
    - name: "DISABLE_ASSET_COMPILATION"
      value: "true"
```

**追加リソース**

- ビルド環境のセクションでは、より詳細な説明を提供します。
- **oc set env** コマンドで、**BuildConfig** に定義した環境変数を管理することも可能です。

### 5.2.4. Source-to-Image (S2I) ソースファイルを無視する

Source to image は **.s2iignore** ファイルをサポートします。このファイルには、無視すべきファイルパターンの一覧が含まれます。 **.s2iignore** ファイルにあるパターンと一致する、さまざまな入力ソースで提供されるビルドの作業ディレクトリーにあるファイルは **assemble** スクリプトでは利用できません。

**.s2iignore** ファイルの形式についての詳細は、 source-to-image ドキュメントを参照してください。

### 5.2.5. S2I によるソースコードからのイメージの作成

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。

#### 5.2.5.1. S2I ビルドプロセスについて

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

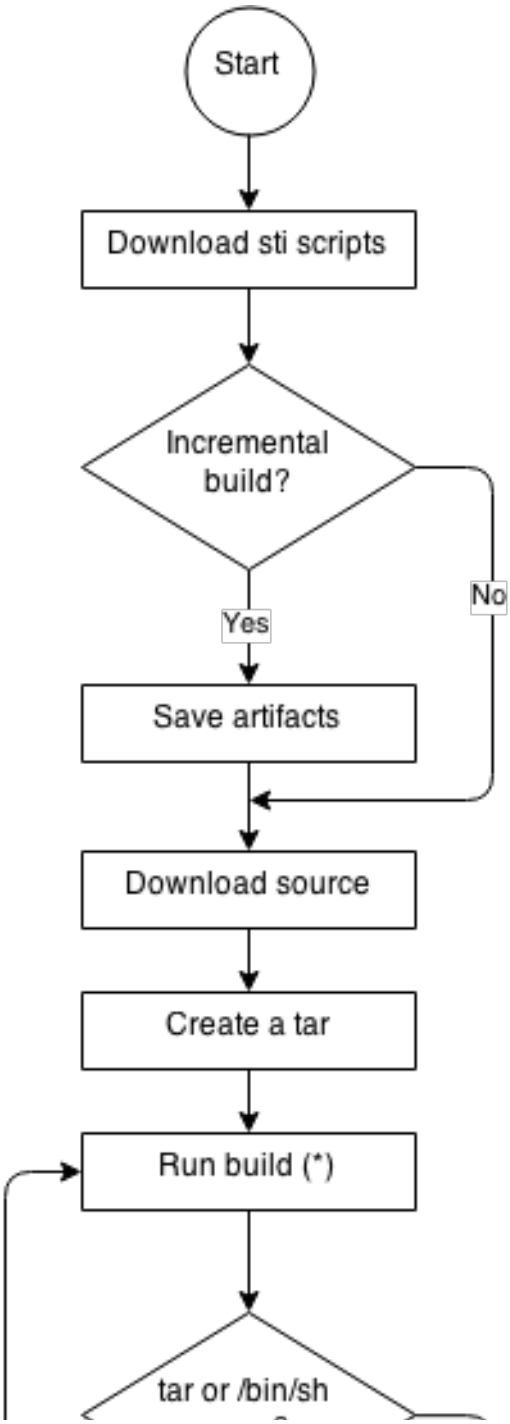
- ビルダーイメージ

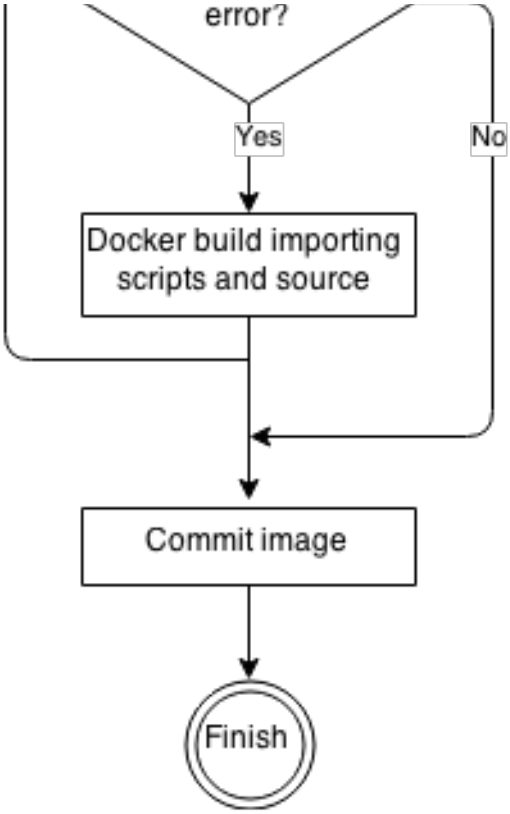
ビルドプロセス中に、S2I はソースとスクリプトをビルダーイメージ内に配置する必要があります。ビルダーイメージ内にソースとスクリプトを配置するために、S2I はソースとスクリプトを含む **tar** ファイルを作成してから、このファイルをビルダーイメージにストリーミングします。**assemble** スクリプトを実行する前に、S2I はファイルを展開して、ビルダーイメージからコンテンツを、デフォルトの **/tmp** ディレクトリーではなく、**io.openshift.s2i.destination** ラベルが指定する場所に配置します。

このプロセスでは、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5.1 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。

#### 5.2.5.2. S2I スクリプトの作成

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5.1 S2I スクリプト**

| スクリプト | 説明 |
| --- | --- |

| 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。<br>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)。<br>2. 任意の場所に、アプリケーションソースを配置します。<br>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。<br>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。以下は例になります。<br>● Ruby の場合は、Bundler でインストールされる **gems**<br>● Java の場合は、**.m2** のコンテンツ<br>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。 |
| **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。<br>1. イメージをビルドします。<br>2. イメージを実行して **usage** スクリプトを検証します。<br>3. **s2i build** を実行して **assemble** スクリプトを検証します。<br>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)。<br>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。<br>  <br>**注記**<br>**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。 |

#### 5.2.5.2.1. S2I スクリプトの例

以下の S2I スクリプトの例は Bash で記述されています。それぞれの例では、tar の内容は**/tmp/s2i** ディレクトリーに展開されることが前提とされています。

**例5.1 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5.2 run スクリプト:**

```
#!/bin/bash

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

**例5.3 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5.4 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
```

## 5.3. カスタムビルド

カスタムビルドストラテジーにより、開発者はビルドプロセス全体を対象とする特定のビルダーイメージを定義できます。独自のビルダーイメージを使用することにより、ビルドプロセスをカスタマイズできます。

カスタムビルダーイメージは、RPM またはベースイメージのビルド用などの、ビルドプロセスロジックで組み込まれた単純な Docker 形式のコンテナーイメージです。

カスタムビルドは非常に高いレベルの権限で実行されるため、デフォルトではユーザーが利用することはできません。クラスター管理者のパーミッションを持つ信頼できるユーザーのみにカスタムビルドを実行するためのアクセスが付与される必要があります。

### 5.3.1. カスタムビルドの FROM イメージの使用

**customStrategy.from** セクションを使用して、カスタムビルドに使用するイメージを指定できます。

**手順**

**customStrategy.from** セクションを設定するには、以下を実行します。

```
strategy:
  customStrategy:
    from:
      kind: "DockerImage"
      name: "openshift/sti-image-builder"
```

### 5.3.2. カスタムビルドでのシークレットの使用

すべてのビルドタイプに追加できるソースおよびイメージのシークレットのほかに、カスタムストラテジーを使用することにより、シークレットの任意の一覧をビルダー Pod に追加できます。

**手順**

各シークレットを特定の場所にマウントするには、以下を実行します。

```
strategy:
  customStrategy:
    secrets:
      - secretSource: 1
          name: "secret1"
        mountPath: "/tmp/secret1" 2
      - secretSource:
          name: "secret2"
        mountPath: "/tmp/secret2"
```

1. **secretSource** は、ビルドと同じ namespace にあるシークレットへの参照です。
2. **mountPath** は、シークレットがマウントされる必要のあるカスタムビルダー内のパスです。

### 5.3.3. カスタムビルドの環境変数の使用

環境変数をカスタムビルドプロセスで利用可能にするには、環境変数を **BuildConfig** の **dockerStrategy** 定義に追加できます。

ここに定義された環境変数は、カスタムビルドを実行する Pod に渡されます。

**手順**

ビルド時に使用されるカスタムの HTTP プロキシーを定義するには、以下を実行します。

```
customStrategy:
...
  env:
    - name: "HTTP_PROXY"
      value: "http://myproxy.net:5187/"
```

**oc set env** コマンドで、**BuildConfig** に定義した環境変数を管理することも可能です。

### 5.3.4. カスタムビルダーイメージの使用

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 Dedicated 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。

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。

#### 5.3.4.1. 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5.2 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 リポジトリーの 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。 |

| 変数名 | 説明 |
| --- | --- |
| **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 Dedicated 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 |
| **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 |

#### 5.3.4.2. 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 Dedicated 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、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。

## 5.4. PIPELINE ビルド

**重要**

Pipeline ビルドストラテジーは OpenShift Dedicated 4 では非推奨になりました。同等の機能および改善機能は、Tekton をベースとする OpenShift Pipeline にあります。

OpenShift の Jenkins イメージは完全にサポートされており、ユーザーは Jenkins ユーザーのドキュメントに従ってジョブで Jenkinsfile を定義するか、またはこれをソースコントロール管理システムに保存します。

開発者は、Pipeline ビルドストラテジーを利用して Jenkins Pipeline プラグインで実行できるように **Jenkins Pipeline** を定義することができます。このビルドは他のビルドタイプの場合と同様に OpenShift Dedicated での起動、モニタリング、管理が可能です。

Pipeline ワークフローは、ビルド設定に直接組み込むか、または Git リポジトリーに配置してビルド設定で参照して Jenkinsfile で定義します。

### 5.4.1. OpenShift Dedicated Pipeline について

Pipeline により、OpenShift Dedicated でのアプリケーションのビルド、デプロイ、およびプロモートに対する制御が可能になります。Jenkins Pipeline ビルドストラテジー、Jenkinsfiles、および OpenShift Dedicated のドメイン固有言語 (DSL) (Jenkins クライアントプラグインで提供される) の組み合わせを使用することにより、すべてのシナリオにおける高度なビルド、テスト、デプロイおよびプロモート用のパイプラインを作成できます。

**OpenShift Dedicated Jenkins 同期プラグイン**

OpenShift Dedicated Jenkins 同期プラグインは、**BuildConfig** および Build オブジェクトを Jenkins ジョブおよびビルドと同期し、以下を提供します。

- Jenkins での動的なジョブ/実行の作成。
- ImageStreams、ImageStreamTag、または ConfigMap からのスレーブ Pod テンプレートの動的作成。
- 環境変数の挿入。
- OpenShift Web コンソールでの Pipeline の可視化。
- Jenkins Git プラグインとの統合。 ここからコミット情報が渡されます。
- OpenShift が Jenkins git プラグインに対して作成する Jenkins 認証情報エントリーに対するシークレットの同期。

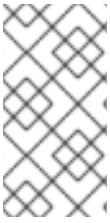
**OpenShift Dedicated Jenkins Client プラグイン**

OpenShift Dedicated Jenkins Client プラグインは、OpenShift Dedicated API Server との高度な対話を実現するために、読み取り可能かつ簡潔で、包括的で Fluent (流れるような) な Jenkins パイプライン構文を提供することを目的とした Jenkins プラグインです。このプラグインは、スクリプトを実行するノードで使用できる必要がある OpenShift コマンドラインツール (**oc**) を活用します。

Jenkins クライアントプラグインは Jenkins マスターにインストールされ、OpenShift Dedicated DSL がアプリケーションの JenkinsFile 内で利用可能である必要があります。このプラグインは、OpenShift Dedicated Jenkins イメージの使用時にデフォルトでインストールされ、有効にされます。

プロジェクト内で OpenShift Dedicated Pipeline を使用するには、Jenkins Pipeline ビルドストラテジーを使用する必要があります。このストラテジーはソースリポジトリーのルートで **jenkinsfile** を使用するようにデフォルト設定されますが、以下の設定オプションも提供します。

- **BuildConfig** 内のインラインの **jenkinsfile** フィールド。
- ソース **contextDir** との関連で使用する **jenkinsfile** の場所を参照する **BuildConfig** 内の **jenkinsfilePath**。



**注記**

オプションの **jenkinsfilePath** フィールドは、ソース **contextDir** との関連で使用するファイルの名前を指定します。**contextDir** が省略される場合、デフォルトはリポジトリーのルートに設定されます。**jenkinsfilePath** が省略される場合、デフォルトは **jenkinsfile** に設定されます。

### 5.4.2. Pipeline ビルド用の Jenkinsfile の提供

**jenkinsfile** は標準的な groovy 言語構文を使用して、アプリケーションの設定、ビルド、およびデプロイメントに対する詳細な制御を可能にします。

**jenkinsfile** は以下のいずれかの方法で指定できます。

- ソースコードリポジトリー内にあるファイルの使用。
- **jenkinsfile** フィールドを使用してビルド設定の一部として組み込む。

最初のオプションを使用する場合、**jenkinsfile** を以下の場所のいずれかでアプリケーションソースコードリポジトリーに組み込む必要があります。

- リポジトリーのルートにある **jenkinsfile** という名前のファイル。
- リポジトリーのソース **contextDir** のルートにある **jenkinsfile** という名前のファイル。
- ソース **contextDir** に関連して BuildConfig の **JenkinsPipelineStrategy** セクションの **jenkinsfilePath** フィールドで指定される名前のファイル (指定される場合)。 指定されない場合は、リポジトリーのルートに設定されます。

**jenkinsfile** は Jenkins スレーブ Pod で実行されます。 ここでは OpenShift DSL を使用する場合に OpenShift クライアントのバイナリーを利用可能にしておく必要があります。

### 手順

Jenkinsfile を指定するには、以下のいずれかを実行できます。

1. ビルド設定に Jenkinsfile を埋め込む
2. Jenkinsfile を含む git リポジトリーへの参照をビルド設定に追加する

### 埋め込み定義

```
kind: "BuildConfig"
apiVersion: "v1"
metadata:
  name: "sample-pipeline"
spec:
  strategy:
    jenkinsPipelineStrategy:
      jenkinsfile: |-
        node('agent') {
          stage 'build'
          openshiftBuild(buildConfig: 'ruby-sample-build', showBuildLogs: 'true')
          stage 'deploy'
          openshiftDeploy(deploymentConfig: 'frontend')
        }
```

### git リポジトリーへの参照

```
kind: "BuildConfig"
apiVersion: "v1"
metadata:
  name: "sample-pipeline"
spec:
  source:
    git:
      uri: "https://github.com/openshift/ruby-hello-world"
```

```
strategy:
  jenkinsPipelineStrategy:
    jenkinsfilePath: some/repo/dir/filename 1
```

1 オプションの **jenkinsfilePath** フィールドは、ソース **contextDir** との関連で使用するファイルの名前を指定します。**contextDir** が省略される場合、デフォルトはリポジトリーのルートに設定されます。**jenkinsfilePath** が省略される場合、デフォルトは **Jenkinsfile** に設定されます。

### 5.4.3. Pipeline ビルドの環境変数の使用

環境変数を Pipeline ビルドプロセスで利用可能にするには、環境変数を **BuildConfig** の **jenkinsPipelineStrategy** 定義に追加できます。

定義した後に、環境変数は **BuildConfig** に関連する Jenkins ジョブのパラメーターとして設定されます。

#### 手順

ビルド時に使用される環境変数を定義するには、以下を実行します。

```
jenkinsPipelineStrategy:
...
  env:
    - name: "FOO"
      value: "BAR"
```

**oc set env** コマンドで、**BuildConfig** に定義した環境変数を管理することも可能です。

#### 5.4.3.1. BuildConfig 環境変数と Jenkins ジョブパラメーター間のマッピング

Pipeline ストラテジーの **BuildConfig** への変更に従い、Jenkins ジョブが作成/更新されると、**BuildConfig** の環境変数は Jenkins ジョブパラメーターの定義にマッピングされます。 Jenkins ジョブパラメーター定義のデフォルト値は、関連する環境変数の現在の値になります。

Jenkins ジョブの初回作成後に、パラメーターを Jenkins コンソールからジョブに追加できます。パラメーター名は、**BuildConfig** の環境変数名とは異なります。上記の Jenkins ジョブ用にビルドを開始すると、これらのパラメーターが使用されます。

Jenkins ジョブのビルドを開始する方法により、パラメーターの設定方法が決まります。

- **oc start-build** で開始された場合には、**BuildConfig** の環境変数が対応するジョブインスタンスに設定するパラメーターになります。Jenkins コンソールからパラメーターのデフォルト値に変更を加えても無視されます。**BuildConfig** の値が優先されます。
- **oc start-build -e** で開始する場合、**-e** オプションで指定される環境変数の値が優先されます。
  - **BuildConfig** に一覧表示されていない環境変数を指定する場合、それらは Jenkins ジョブパラメーター定義として追加されます。
  - Jenkins コンソールから環境変数に対応するパラメーターに加える変更は無視されます。**BuildConfig** および **oc start-build -e** て指定する内容が優先されます。
- Jenkins コンソールで Jenkins ジョブを開始した場合には、ジョブのビルドを開始する操作の一環として、Jenkins コンソールを使用してパラメーターの設定を制御できます。

**注記**

ジョブパラメーターに関連付けられる可能性のあるすべての環境変数を、**BuildConfig** に指定することが推奨されます。これにより、ディスク I/O が減り、Jenkins 処理時のパフォーマンスが向上します。

### 5.4.4. Pipeline ビルドのチュートリアル

以下の例では、**nodejs-mongodb.json** テンプレートを使用して **Node.js/MongoDB** アプリケーションをビルドし、デプロイし、検証する OpenShift Pipeline を作成する方法を紹介します。

**手順**

1. Jenkins マスターを作成するには、以下を実行します。

```
$ oc project <project_name> 1
$ oc new-app jenkins-ephemeral 2
```

1 **oc new-project <project_name>** で新規プロジェクトを使用するか、または作成するプロジェクトを選択します。

2 永続ストレージを使用する場合は、**jenkins-persistent** を代わりに使用します。

2. 以下の内容で **nodejs-sample-pipeline.yaml** という名前のファイルを作成します。

**注記**

Jenkins Pipeline ストラテジーを使用して **Node.js/MongoDB** のサンプルアプリケーションをビルドし、デプロイし、スケーリングする **BuildConfig** を作成します。

```
kind: "BuildConfig"
apiVersion: "v1"
metadata:
  name: "nodejs-sample-pipeline"
spec:
  strategy:
    jenkinsPipelineStrategy:
      jenkinsfile: <pipeline content from below>
    type: JenkinsPipeline
```

3. **jenkinsPipelineStrategy** で **BuildConfig** を作成したら、インラインの **jenkinsfile** を使用して、Pipeline に指示を出します。

**注記**

この例では、アプリケーションに Git リポジトリーを設定しません。

以下の **jenkinsfile** の内容は、OpenShift DSL を使用して Groovy で記述されています。ソースリポジトリーに **jenkinsfile** を追加することが推奨される方法ですが、この例では YAML Literal Style を使用して **BuildConfig** にインラインコンテンツを追加しています。

```
def templatePath = 'https://raw.githubusercontent.com/openshift/nodejs-
ex/master/openshift/templates/nodejs-mongodb.json' 1
def templateName = 'nodejs-mongodb-example' 2
pipeline {
  agent {
    node {
      label 'nodejs' 3
    }
  }
  options {
    timeout(time: 20, unit: 'MINUTES') 4
  }
  stages {
    stage('preamble') {
        steps {
            script {
                openshift.withCluster() {
                    openshift.withProject() {
                        echo "Using project: ${openshift.project()}"
                    }
                }
            }
        }
    }
    stage('cleanup') {
      steps {
        script {
            openshift.withCluster() {
                openshift.withProject() {
                  openshift.selector("all", [ template : templateName ]).delete() 5
                  if (openshift.selector("secrets", templateName).exists()) { 6
                    openshift.selector("secrets", templateName).delete()
                  }
                }
            }
        }
      }
    }
    stage('create') {
      steps {
        script {
            openshift.withCluster() {
                openshift.withProject() {
                  openshift.newApp(templatePath) 7
                }
            }
```

```
      }
    }
  }
  stage('build') {
    steps {
      script {
          openshift.withCluster() {
              openshift.withProject() {
                def builds = openshift.selector("bc", templateName).related('builds')
                timeout(5) { 8
                  builds.untilEach(1) {
                    return (it.object().status.phase == "Complete")
                  }
                }
              }
          }
      }
    }
  }
  stage('deploy') {
    steps {
      script {
          openshift.withCluster() {
              openshift.withProject() {
                def rm = openshift.selector("dc", templateName).rollout()
                timeout(5) { 9
                  openshift.selector("dc", templateName).related('pods').untilEach(1) {
                    return (it.object().status.phase == "Running")
                  }
                }
              }
          }
      }
    }
  }
  stage('tag') {
    steps {
      script {
          openshift.withCluster() {
              openshift.withProject() {
                openshift.tag("${templateName}:latest", "${templateName}-staging:latest") 10
              }
          }
      }
    }
  }
}
```

1. 使用するテンプレートへのパス
2. 作成するテンプレート名
3. このビルドを実行する **node.js** のスレーブ Pod をスピンアップします。
4. この Pipeline に 20 分間のタイムアウトを設定します。

5 このテンプレートラベルが指定されたものすべてを削除します。

6 このテンプレートラベルが付いたシークレットをすべて削除します。

7 **templatePath** から新規アプリケーションを作成します。

8 ビルドが完了するまで最大 5 分待機します。

9 デプロイメントが完了するまで最大 5 分待機します。

10 すべてが正常に完了した場合は、**$ {templateName}:latest** イメージに **$ {templateName}-staging:latest** のタグを付けます。ステージング環境向けの Pipeline の **BuildConfig** は、変更する **$ {templateName}-staging:latest** イメージがないかを確認し、このイメージをステージング環境にデプロイします。

**注記**

以前の例は、**declarative pipeline** スタイルを使用して記述されていますが、以前の **scripted pipeline** スタイルもサポートされます。

4. OpenShift クラスターに Pipeline **BuildConfig** を作成します。

```
$ oc create -f nodejs-sample-pipeline.yaml
```

   a. 独自のファイルを作成しない場合には、以下を実行して Origin リポジトリーからサンプルを使用できます。

```
$ oc create -f
https://raw.githubusercontent.com/openshift/origin/master/examples/jenkins/pipeline/nodejs-sample-pipeline.yaml
```

5. Pipeline を起動します。

```
$ oc start-build nodejs-sample-pipeline
```

**注記**

または、OpenShift Web コンソールで Builds → Pipeline セクションに移動して、**Start Pipeline** をクリックするか、Jenkins コンソールから作成した Pipeline に移動して、**Build Now** をクリックして Pipeline を起動できます。

パイプラインが起動したら、以下のアクションがプロジェクト内で実行されるはずです。

- ジョブインスタンスが Jenkins サーバー上で作成される
- パイプラインで必要な場合には、スレーブ Pod が起動される
- Pipeline がスレーブ Pod で実行されるか、またはスレーブが必要でない場合には master で実行される
  - **template=nodejs-mongodb-example** ラベルの付いた以前に作成されたリソースは削除されます。

- 新規アプリケーションおよびそれに関連するすべてのリソースは、**nodejs-mongodb-example** テンプレートで作成されます。
- ビルドは **nodejs-mongodb-example BuildConfig** を使用して起動されます。
  - Pipeline は、ビルドが完了して次のステージをトリガーするまで待機します。
- デプロイメントは、**nodejs-mongodb-example** のデプロイメント設定を使用して開始されます。
  - パイプラインは、デプロイメントが完了して次のステージをトリガーするまで待機します。
- ビルドとデプロイに成功すると、**nodejs-mongodb-example:latest** イメージが **nodejs-mongodb-example:stage** としてトリガーされます。

- Pipeline で以前に要求されていた場合には、スレーブ Pod が削除される

**注記**

OpenShift Web コンソールで確認すると、最適な方法で Pipeline の実行を視覚的に把握することができます。Web コンソールにログインして、Builds → Pipelines に移動し、Pipeline を確認します。

## 5.5. WEB コンソールを使用したシークレットの追加

プライベートリポジトリーにアクセスできるように、ビルド設定にシークレットを追加することができます。

**手順**

プライベートリポジトリーにアクセスできるようにビルド設定にシークレットを追加するには、以下を実行します。

1. 新規の OpenShift Dedicated プロジェクトを作成します。
2. プライベートのソースコードリポジトリーにアクセスするための認証情報が含まれるシークレットを作成します。
3. ビルド設定を作成します。
4. ビルド設定エディターページまたは Web コンソールの **create app from builder image** ページで、**Source Secret** を設定します。
5. **Save** ボタンをクリックします。

## 5.6. プルおよびプッシュの有効化

プライベートレジストリーへのプルを実行できるようにするには、ビルド設定に **Pull Secret** を設定し、プッシュを有効にするには **Push Secret** を設定します。

**手順**

プライベートレジストリーへのプルを有効にするには、以下を実行します。

- ビルド設定に **Pull Secret** を設定します。

プッシュを有効にするには、以下を実行します。

- ビルド設定に **Push Secret** を設定します。

# 第6章 基本的なビルドの実行

以下のセクションでは、ビルドの開始および中止、BuildConfig の削除、ビルドの詳細の表示、およびビルドログへのアクセスを含む基本的なビルド操作についての方法を説明します。

## 6.1. ビルドの開始

現在のプロジェクトに既存のビルド設定から新規ビルドを手動で起動できます。

**手順**

手動でビルドを開始するには、以下を実行します。

```
$ oc start-build <buildconfig_name>
```

### 6.1.1. ビルドの再実行

**--from-build** フラグを使用してビルドを手動で再度実行します。

**手順**

手動でビルドを再実行するには、以下を実行します。

```
$ oc start-build --from-build=<build_name>
```

### 6.1.2. ビルドログのストリーミング

**--follow** フラグを指定して、標準出力 (stdout) のビルドのログをストリーミングします。

**手順**

標準出力 (stdout) でビルドのログを手動でストリーミングするには、以下を実行します。

```
$ oc start-build <buildconfig_name> --follow
```

### 6.1.3. ビルド開始時の環境変数の設定

**--env** フラグを指定して、ビルドの任意の環境変数を設定します。

**手順**

必要な環境変数を指定するには、以下を実行します。

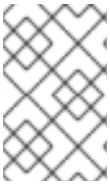
```
$ oc start-build <buildconfig_name> --env=<key>=<value>
```

### 6.1.4. ソースを使用したビルドの開始

Git ソースプルまたは Dockerfile に依存してビルドするのではなく、ソースを直接プッシュしてビルドを開始することも可能です。ソースには、Git または SVN の作業ディレクトリーの内容、デプロイする事前にビルド済みのバイナリーアーティファクトのセットまたは単一ファイルのいずれかを選択できます。これは、**start-build** コマンドに以下のオプションのいずれかを指定して実行できます。

| オプション | 説明 |
| --- | --- |
| **--from-dir=<directory>** | アーカイブし、ビルドのバイナリー入力として使用するディレクトリーを指定します。 |
| **--from-file=<file>** | 単一ファイルを指定します。これはビルドソースで唯一のファイルでなければなりません。 このファイルは、元のファイルと同じファイル名で空のディレクトリーのルートに置いてください。 |
| **--from-repo=<local_source_repo>** | ビルドのバイナリー入力として使用するローカルリポジトリーへのパスを指定します。**--commit** オプションを追加して、ビルドに使用するブランチ、タグ、またはコミットを制御します。 |

以下のオプションをビルドに直接指定した場合には、コンテンツはビルドにストリーミングされ、現在のビルドソースの設定が上書きされます。



**注記**

バイナリー入力からトリガーされたビルドは、サーバー上にソースを保存しないため、ベースイメージの変更でビルドが再度トリガーされた場合には、ビルド設定で指定されたソースが使用されます。

**手順**

たとえば、以下のコマンドは、タグ **v2** からのアーカイブとしてローカルの Git リポジトリーのコンテンツを送信し、ビルドを開始します。

```
$ oc start-build hello-world --from-repo=../hello-world --commit=v2
```

## 6.2. ビルドの中止

Web コンソールまたは以下の CLI コマンドを使用して、ビルドを中止できます。

**手順**

手動でビルドを中止するには、以下を実行します。

```
$ oc cancel-build <build_name>
```

### 6.2.1. 複数ビルドの中止

以下の CLI コマンドを使用して複数ビルドを中止できます。

**手順**

手動で複数ビルドを中止するには、以下を実行します。

```
$ oc cancel-build <build1_name> <build2_name> <build3_name>
```

### 6.2.2. すべてのビルドの中止

以下の CLI コマンドを使用し、ビルド設定からすべてのビルドを中止できます。

**手順**

すべてのビルドを中止するには、以下を実行します。

```
$ oc cancel-build bc/<buildconfig_name>
```

### 6.2.3. 指定された状態のすべてのビルドの中止

特定の状態にあるビルドをすべて中止できます (例: **new** または **pending**)。 この際、他の状態のビルドは無視されます。

**手順**

指定された状態のすべてのビルドを中止するには、以下を実行します。

```
$ oc cancel-build bc/<buildconfig_name>
```

## 6.3. BUILDCONFIG の削除

以下のコマンドで **BuildConfig** を削除します。

**手順**

**BuildConfig** を削除するには、以下を実行します。

```
$ oc delete bc <BuildConfigName>
```

これにより、この **BuildConfig** でインスタンス化されたビルドがすべて削除されます。ビルドを削除しない場合には、**--cascade=false** フラグを指定します。

```
$ oc delete --cascade=false bc <BuildConfigName>
```

## 6.4. ビルドの詳細表示

Web コンソールまたは **oc describe** CLI コマンドを使用して、ビルドの詳細を表示できます。

これにより、以下のような情報が表示されます。

- ビルドソース
- ビルドストラテジー
- 出力先
- 宛先レジストリーのイメージのダイジェスト
- ビルドの作成方法

ビルドが **Docker** または **Source** ストラテジーを使用する場合、**oc describe** 出力には、コミット ID、作成者、コミットしたユーザー、メッセージなどのビルドに使用するソースのリビジョンの情報が含まれます。

**手順**

ビルドの詳細を表示するには、以下を実行します。

```
$ oc describe build <build_name>
```

## 6.5. ビルドログへのアクセス

Web コンソールまたは CLI を使用してビルドログにアクセスできます。

**手順**

ビルドを直接使用してログをストリーミングするには、以下を実行します。

```
$ oc describe build <build_name>
```

### 6.5.1. BuildConfig ログへのアクセス

Web コンソールまたは CLI を使用して **BuildConfig** ログにアクセスできます。

**手順**

BuildConfig の最新ビルドのログをストリーミングするには、以下を実行します。

```
$ oc logs -f bc/<buildconfig_name>
```

### 6.5.2. 特定バージョンのビルドについての BuildConfig ログへのアクセス

Web コンソールまたは CLI を使用して、**BuildConfig** についての特定バージョンのビルドのログにアクセスすることができます。

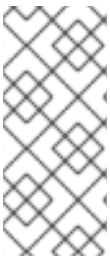
**手順**

BuildConfig の特定バージョンのビルドのログをストリームするには、以下を実行します。

```
$ oc logs --version=<number> bc/<buildconfig_name>
```

### 6.5.3. ログの冗長性の有効化

詳細の出力を有効にするには、**BuildConfig** 内の **sourceStrategy** または **dockerStrategy** の一部として **BUILD_LOGLEVEL** 環境変数を指定します。



**注記**

管理者は、**env/BUILD_LOGLEVEL** を設定して、OpenShift Dedicated インスタンス全体のデフォルトのビルドの詳細レベルを設定できます。このデフォルトは、指定の **BuildConfig** で **BUILD_LOGLEVEL** を指定することで上書きできます。コマンドラインで **--build-loglevel** を **oc start-build** に渡すことで、バイナリー以外のビルドについて優先順位の高い上書きを指定することができます。

ソースビルドで利用できるログレベルは以下のとおりです。

| | |
|---|---|
| レベル 0 | **assemble** スクリプトを実行してコンテナーからの出力とすべてのエラーを生成します。これはデフォルトの設定です。 |
| レベル 1 | 実行したプロセスに関する基本情報を生成します。 |
| レベル 2 | 実行したプロセスに関する詳細情報を生成します。 |
| レベル 3 | 実行したプロセスに関する詳細情報と、アーカイブコンテンツの一覧を生成します。 |
| レベル 4 | 現時点ではレベル 3 と同じ情報を生成します。 |
| レベル 5 | これまでのレベルで記載したすべての内容と docker のプッシュメッセージを提供します。 |

### 手順

詳細の出力を有効にするには、**BuildConfig** 内の **sourceStrategy** または **dockerStrategy** の一部として **BUILD_LOGLEVEL** 環境変数を渡します。

```
sourceStrategy:
...
  env:
    - name: "BUILD_LOGLEVEL"
      value: "2" 1
```

1 この値を任意のログレベルに調整します。

# 第7章 ビルドのトリガーおよび変更

以下のセクションでは、ビルドフックを使用してビルドをトリガーし、ビルドを変更する方法についての概要を説明します。

## 7.1. ビルドトリガー

**BuildConfig** の定義時に、**BuildConfig** を実行する必要のある状況を制御するトリガーを定義できます。以下のビルドトリガーを利用できます。

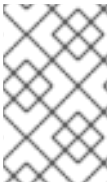
- Webhook
- イメージの変更
- 設定の変更

### 7.1.1. Webhook のトリガー

Webhook のトリガーにより、要求を OpenShift Dedicated API エンドポイントに送信して新規ビルドをトリガーできます。GitHub、GitLab、Bitbucketまたは Generic webhook を使用して、Webhook トリガーを定義できます。

OpenShift Dedicated の Webhook は現在、Git ベースのソースコード管理システム (SCM) のそれぞれのプッシュイベントに似たイベントバージョンのみをサポートしています。その他のイベントタイプはすべて無視されます。

プッシュイベントを処理する場合に、OpenShift Dedicated マスターホストは、イベント内のブランチ参照が、対応の **BuildConfig** のブランチ参照と一致しているかどうを確認します。一致する場合には、OpenShift Dedicated ビルドの Webhook イベントに記載されているのと全く同じコミット参照がチェックアウトされます。一致しない場合には、ビルドはトリガーされません。



**注記**

**oc new-app** および **oc new-build** は GitHub および Generic Webhook トリガーを自動的に作成しますが、それ以外の Webhook トリガーが必要な場合には手動で追加する必要があります (「トリガーの設定」を参照)。

Webhook すべてに対して、**WebHookSecretKey** という名前のキーで、**Secret** と、Webook の呼び出し時に提供される値を定義する必要があります。webhook の定義で、このシークレットを参照する必要があります。このシークレットを使用することで URL が一意となり、他の URL でビルドがトリガーされないようにします。キーの値は、webhook の呼び出し時に渡されるシークレットと比較されます。

たとえば、**mysecret** という名前のシークレットを参照する GitHub webhook は以下のとおりです。

```
type: "GitHub"
github:
  secretReference:
    name: "mysecret"
```

次に、シークレットは以下のように定義します。シークレットの値は base64 エンコードされており、この値は **Secret** オブジェクトの **data** フィールドに必要である点に注意してください。

```
- kind: Secret
  apiVersion: v1
```

```
metadata:
  name: mysecret
  creationTimestamp:
data:
  WebHookSecretKey: c2VjcmV0dmFsdWUx
```

**追加リソース**

- GitHub
- GitLab
- Bitbucket

#### 7.1.1.1. GitHub Webhook の使用

GitHub webhook は 、リポジトリーの更新時に GitHub からの呼び出しを処理します。トリガーを定義するときに、**secret** を定義してください。 このシークレットは、Webhook の設定時に GitHub に渡される URL に追加されます。

GitHub Webhook の定義例:

```
type: "GitHub"
github:
  secretReference:
    name: "mysecret"
```

**注記**

Webhook トリガーの設定で使用されるシークレットは、GitHub UI で Webhook 設定時に表示される **secret** フィールドとは異なります。Webhook トリガー設定で使用するシークレットは、Webhook URL を一意にして推測ができないようにし、GitHub UI のシークレットは、任意の文字列フィールドで、このフィールドを使用して本体の HMAC hex ダイジェストを作成して、**X-Hub-Signature** ヘッダーとして送信します。

**oc describe** コマンドは、ペイロード URL を GitHub Webhook URL として返します (「Webhook URL の表示」を参照)。 ペイロード URL は以下のような構成です。

```
http://<openshift_api_host:port>/oapi/v1/namespaces/<namespace>/buildconfigs/<name>/webhooks/<se
cret>/github
```

**前提条件**

- GitHub リポジトリーから **BuildConfig** を作成します。

**手順**

1. GitHub Webhook を設定するには以下を実行します。
   a. GitHub リポジトリーから **BuildConfig** を作成した後に、以下を実行します。

```
$ oc describe bc/<name-of-your-BuildConfig>
```

以下のように、上記のコマンドは Webhook GitHub URL を生成します。

```
<https://api.starter-us-east-
1.openshift.com:443/oapi/v1/namespaces/nsname/buildconfigs/bcname/webhooks/<secret>
/github>.
```

b. GitHub の Web コンソールから、この URL を GitHub にカットアンドペーストします。

c. GitHub リポジトリーで、**Settings → Webhooks & Services** から **Add Webhook** を選択します。

d. **Payload URL** フィールドに、(上記と同様の) URL の出力を貼り付けます。

e. **Content Type** を GitHub のデフォルト **application/x-www-form-urlencoded** から **application/json** に変更します。

f. **Add webhook** をクリックします。
webhook の設定が正常に完了したことを示す GitHub のメッセージが表示されます。

これで変更を GitHub リポジトリーにプッシュするたびに新しいビルドが自動的に起動し、ビルドに成功すると新しいデプロイメントが起動します。

**注記**

Gogs は、GitHub と同じ webhook のペイロード形式をサポートします。そのため、Gogs サーバーを使用する場合は、GitHub webhook トリガーを **BuildConfig** に定義すると、Gogs サーバー経由でもトリガーされます。

2. **payload.json** などの有効な JSON ペイロードがファイルに含まれる場合には、**curl** を使用して webhook を手動でトリガーできます。

```
$ curl -H "X-GitHub-Event: push" -H "Content-Type: application/json" -k -X POST --data-
binary @payload.json
https://<openshift_api_host:port>/oapi/v1/namespaces/<namespace>/buildconfigs/<name>/web
hooks/<secret>/github
```

**-k** の引数は、API サーバーに正しく署名された証明書がない場合にのみ必要です。

**追加リソース**

- GitHub
- Gogs

#### 7.1.1.2. GitLab Webhook の使用

GitLab Webhook は、リポジトリーの更新時の GitLab による呼び出しを処理します。GitHub トリガーでは、**secret** を指定する必要があります。以下の例は、**BuildConfig** 内のトリガー定義の YAML です。

```
type: "GitLab"
gitlab:
  secretReference:
    name: "mysecret"
```

**oc describe** コマンドは、ペイロード URL を GitLab Webhook URL として返します (「Webhook URL の表示」を参照)。 ペイロード URL は以下のような構成です。

```
http://<openshift_api_host:port>/oapi/v1/namespaces/<namespace>/buildconfigs/<name>/webhooks/<secret>/gitlab
```

**手順**

1. GitLab Webhook を設定するには以下を実行します。
   a. **BuildConfig** を Webhook URL を取得するように記述します。

   ```
   $ oc describe bc <name>
   ```

   b. Webhook URL をコピーします。 **<secret>** はシークレットの値に置き換えます。
   c. GitLab の設定手順 に従い、GitLab リポジトリーの設定に Webhook URL を貼り付けます。
2. **payload.json** などの有効な JSON ペイロードがファイルに含まれる場合には、 **curl** を使用して webhook を手動でトリガーできます。

   ```
   $ curl -H "X-GitLab-Event: Push Hook" -H "Content-Type: application/json" -k -X POST --data-binary @payload.json https://<openshift_api_host:port>/oapi/v1/namespaces/<namespace>/buildconfigs/<name>/webhooks/<secret>/gitlab
   ```

   **-k** の引数は、API サーバーに正しく署名された証明書がない場合にのみ必要です。

**追加リソース**

- GitLab

### 7.1.1.3. Bitbucket Webhook の使用

Bitbucket Webhook リポジトリーの更新時の Bitbucket による呼び出しを処理します。これまでのトリガーと同様に、**secret** を指定する必要があります。以下の例は、 **BuildConfig** 内のトリガー定義の YAML です。

```
type: "Bitbucket"
bitbucket:
  secretReference:
    name: "mysecret"
```

**oc describe** コマンドは、ペイロード URL を Bitbucket Webhook URL として返します (「Webhook URL の表示」を参照)。 ペイロード URL は以下のような構成です。

```
http://<openshift_api_host:port>/oapi/v1/namespaces/<namespace>/buildconfigs/<name>/webhooks/<secret>/bitbucket
```

**手順**

1. Bitbucket Webhook を設定するには以下を実行します。

a. 'BuildConfig' を記述して Webhook URL を取得します。

```
$ oc describe bc <name>
```

b. Webhook URL をコピーします。 **<secret>** はシークレットの値に置き換えます。

c. Bitbucket の設定手順 に従い、Bitbucket リポジトリーの設定に Webhook URL を貼り付けます。

2. **payload.json** などの有効な JSON ペイロードがファイルに含まれる場合には、 **curl** を使用して webhook を手動でトリガーできます。

```
$ curl -H "X-Event-Key: repo:push" -H "Content-Type: application/json" -k -X POST --data-binary @payload.json https://<openshift_api_host:port>/oapi/v1/namespaces/<namespace>/buildconfigs/<name>/webhooks/<secret>/bitbucket
```

**-k** の引数は、API サーバーに正しく署名された証明書がない場合にのみ必要です。

**追加リソース**

- Bitbucket

### 7.1.1.4. Generic Webhook の使用

Generic Webhook は、Web 要求を実行できるシステムから呼び出されます。他の webhook と同様に、シークレットを指定する必要があります。このシークレットを使用することで URL が一意となり、他の URL でビルドがトリガーされないようにします。このシークレットを使用することで URL が一意となり、他の URL でビルドがトリガーされないようにします。 以下の例は、**BuildConfig** 内のトリガー定義の YAML です。

```
type: "Generic"
generic:
  secretReference:
    name: "mysecret"
  allowEnv: true (1)
```

(1) **true** に設定して、Generic Webhook が環境変数で渡させるようにします。

**手順**

1. 呼び出し元を設定するには、呼び出しシステムに、ビルドの Generic Webhook エンドポイントの URL を指定します。

```
http://<openshift_api_host:port>/oapi/v1/namespaces/<namespace>/buildconfigs/<name>/webhooks/<secret>/generic
```

呼び出し元は、**POST** 操作として Webhook を呼び出す必要があります。

2. 手動で Webhook を呼び出すには、**curl** を使用します。

```
$ curl -X POST -k
https://<openshift_api_host:port>/oapi/v1/namespaces/<namespace>/buildconfigs/<name>/webhooks/<secret>/generic
```

HTTP 動詞は **POST** に設定する必要があります。セキュアでない **-k** フラグを指定して、証明書の検証を無視します。クラスターに正しく署名された証明書がある場合には、2 つ目のフラグは必要ありません。

エンドポイントは、以下の形式で任意のペイロードを受け入れることができます。

```
git:
  uri: "<url to git repository>"
  ref: "<optional git reference>"
  commit: "<commit hash identifying a specific git commit>"
  author:
    name: "<author name>"
    email: "<author e-mail>"
  committer:
    name: "<committer name>"
    email: "<committer e-mail>"
  message: "<commit message>"
env: 1
  - name: "<variable name>"
    value: "<variable value>"
```

1 **BuildConfig** 環境変数{0} と同様に、ここで定義されている環境変数は、ビルドで利用できます。これらの変数が **BuildConfig** の環境変数と競合する場合には、これらの変数が優先されます。デフォルトでは、webhook 経由で渡された環境変数は無視されます。Webhook 定義の **allowEnv** フィールドを **true** に設定して、この動作を有効にします。

3. **curl** を使用してこのペイロードを渡すには、**payload_file.yaml** という名前のファイルにペイロードを定義して実行します。

```
$ curl -H "Content-Type: application/yaml" --data-binary @payload_file.yaml -X POST -k
https://<openshift_api_host:port>/oapi/v1/namespaces/<namespace>/buildconfigs/<name>/webhooks/<secret>/generic
```

引数は、ヘッダーとペイロードを追加した以前の例と同じです。**-H** の引数は、ペイロードの形式により **Content-Type** ヘッダーを **application/yaml** または **application/json** に設定します。**--data-binary** の引数を使用すると、**POST** 要求では、改行を削除せずにバイナリーペイロードを送信します。

**注記**

OpenShift Dedicated は、要求のペイロードが無効な場合でも (例: 無効なコンテンツタイプ、解析不可能または無効なコンテンツなど)、Generic Webhook 経由でビルドをトリガーできます。この動作は、後方互換性を確保するために継続されています。無効な要求ペイロードがある場合には、OpenShift Dedicated は、**HTTP 200 OK** 応答の一部として JSON 形式で警告を返します。

#### 7.1.1.5. Webhook URL の表示

以下のコマンドを使用して、**BuildConfig** に関連付けられた Webhook URL を表示します。コマンドが Webhook URL を表示しない場合、そのビルド設定に定義される Webhook トリガーはありません。トリガーを手動で追加するには、「トリガーの設定」を参照してください。

**手順**

- **BuildConfig** に関連付けられた Webhook URL を表示するには、以下を実行します。

```
$ oc describe bc <name>
```

### 7.1.2. イメージ変更トリガーの使用

イメージ変更のトリガーを使用すると、アップストリームで新規バージョンが利用できるようになるとビルドが自動的に呼び出されます。たとえば、RHEL イメージ上にビルドが設定されている場合には、RHEL のイメージが変更された時点でビルドの実行をトリガーできます。その結果、アプリケーションイメージは常に最新の RHEL ベースイメージ上で実行されるようになります。

**注記**

v1 コンテナーレジストリー のコンテナーイメージを参照するイメージストリームは、イメージストリームタグが利用できるようになった時点でビルドが度だけトリガーされ、後続のイメージ更新ではトリガーされません。これは、v1 コンテナーレジストリーに一意で識別可能なイメージがないためです。

**手順**

イメージ変更のトリガーを設定するには、以下のアクションを実行する必要があります。

1. トリガーするアップストリームイメージを参照するように、**ImageStream** を定義します。

```
kind: "ImageStream"
apiVersion: "v1"
metadata:
  name: "ruby-20-centos7"
```

この定義では、イメージストリームが **<system-registry>/<namespace>/ruby-20-centos7** に配置されているコンテナーイメージリポジトリーに紐付けられます。**<system-registry>** は、OpenShift Dedicated で実行する **docker-registry** の名前で、サービスとして定義されます。

2. イメージストリームがビルドのベースイメージの場合には、ビルドストラテジーの From フィールドを設定して、イメージストリームを参照します。

```
strategy:
  sourceStrategy:
    from:
      kind: "ImageStreamTag"
      name: "ruby-20-centos7:latest"
```

上記の例では、**sourceStrategy** の定義は、この namespace 内に配置されている **ruby-20-centos7** という名前のイメージストリームの **latest** タグを使用します。

3. イメージストリームを参照する 1 つまたは複数のトリガーでビルドを定義します。

```
type: "imageChange" 1
```

```
imageChange: {}
type: "imageChange" 2
imageChange:
  from:
    kind: "ImageStreamTag"
    name: "custom-image:latest"
```

1 ビルドストラテジーの **from** フィールドに定義されたように **ImageStream** および **Tag** を監視するイメージ変更トリガー。ここの **imageChange** オブジェクトは空でなければなりません。

2 任意のイメージストリームを監視するイメージ変更トリガー。この例に含まれる **imageChange** の部分には **from** フィールドを追加して、監視する **ImageStreamTag** を参照させる必要があります。

ストラテジーイメージストリームにイメージ変更トリガーを使用する場合は、生成されたビルドに不変な Docker タグが付けられ、そのタグに対応する最新のイメージを参照させます。この新規イメージ参照は、ビルド用に実行するときに、ストラテジーにより使用されます。

ストラテジーイメージストリームを参照しない、他のイメージ変更トリガーの場合は、新規ビルドが開始されますが、一意のイメージ参照で、ビルドストラテジーは更新されません。

この例には、ストラテジーについてのイメージ変更トリガーがあるので、結果として生成されるビルドは以下のようになります。

```
strategy:
  sourceStrategy:
    from:
      kind: "DockerImage"
      name: "172.30.17.3:5001/mynamespace/ruby-20-centos7:<immutableid>"
```

これにより、トリガーされたビルドは、リポジトリーにプッシュされたばかりの新しいイメージを使用して、ビルドが同じ入力内容でいつでも再実行できるようにします。

ビルドの開始前に、参照されるイメージストリームに対する複数の変更を可能にするためにイメージ変更トリガーを一時停止することができます。また、ビルドがすぐにトリガーされるのを防ぐために、最初に **ImageChangeTrigger** を **BuildConfig** に追加する際に、 **paused** 属性を true に設定することもできます。

```
type: "ImageChange"
imageChange:
  from:
    kind: "ImageStreamTag"
    name: "custom-image:latest"
  paused: true
```

カスタムビルドの場合、すべての **Strategy** タイプにイメージフィールドを設定するだけでなく、**OPENSHIFT_CUSTOM_BUILD_BASE_IMAGE** の環境変数もチェックされます。この環境変数が存在しない場合は、不変のイメージ参照で作成されます。存在する場合には、この不変のイメージ参照で更新されます。

ビルドが Webhook トリガーまたは手動の要求でトリガーされた場合に、作成されるビルドは、**Strategy** が参照する **ImageStream** から解決する **<immutableid>** を使用します。これにより、簡単に再現できるように、一貫性のあるイメージタグを使用してビルドが実行されるようになります。

**追加リソース**

- v1 コンテナーレジストリー

### 7.1.3. 設定変更のトリガー

設定変更トリガーにより、新規の **BuildConfig** が作成されるとすぐに、ビルドが自動的に起動されます。

このシークレットを使用することで URL が一意となり、他の URL でビルドがトリガーされないようにします。 以下の例は、**BuildConfig** 内のトリガー定義の YAML です。

```
type: "ConfigChange"
```

**注記**

設定変更のトリガーは新しい **BuildConfig** が作成された場合のみ機能します。今後のリリースでは、設定変更トリガーは、**BuildConfig** が更新されるたびにビルドを起動できるようになります。

#### 7.1.3.1. トリガーの手動設定

トリガーは、**oc set triggers** を使用してビルド設定に対して追加/削除できます。

**手順**

- ビルド設定に GitHub Webhook トリガーを設定するには、以下を使用します。

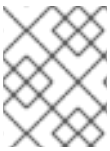
```
$ oc set triggers bc <name> --from-github
```

- イメージ変更トリガーを設定するには以下を使用します。

```
$ oc set triggers bc <name> --from-image='<image>'
```

- トリガーを削除するには **--remove** を追加します。

```
$ oc set triggers bc <name> --from-bitbucket --remove
```



**注記**

Webhook トリガーがすでに存在する場合には、トリガーをもう一度追加すると、Webhook のシークレットが再生成されます。

詳細情報は、**oc set triggers --help** のヘルプドキュメントを参照してください。

## 7.2. ビルドフック

ビルドフックを使用すると、ビルドプロセスに動作を挿入できます。

**BuildConfig** オブジェクトの **postCommit** フィールドにより、ビルドアウトプットイメージを実行する一時的なコンテナー内でコマンドが実行されます。イメージの最後の層がコミットされた直後、かつイメージがレジストリーにプッシュされる前に、フックが実行されます。

現在の作業ディレクトリーは、イメージの **WORKDIR** に設定され、コンテナーイメージのデフォルトの作業ディレクトリーになります。多くのイメージでは、ここにソースコードが配置されます。

ゼロ以外の終了コードが返された場合、一時コンテナーの起動に失敗した場合には、フックが失敗します。フックが失敗すると、ビルドに失敗とマークされ、このイメージはレジストリーにプッシュされません。失敗の理由は、ビルドログを参照して検証できます。

ビルドフックは、ビルドが完了とマークされ、イメージがレジストリーに公開される前に、単体テストを実行してイメージを検証するために使用できます。すべてのテストに合格し、テストランナーにより終了コード 0 が返されると、ビルドは成功とマークされます。テストに失敗すると、ビルドは失敗とマークされます。すべての場合で、ビルドログには、テストランナーの出力が含まれるので、失敗したテストを特定するのに使用できます。

**postCommit** フックは、テストの実行だけでなく、他のコマンドにも使用できます。一時的なコンテナーで実行されるので、フックによる変更は永続されず、フックの実行は最終的なイメージには影響がありません。この動作はさまざまな用途がありますが、これにより、テストの依存関係がインストール、使用されて、自動的に破棄され、最終イメージには残らないようにすることができます。

### 7.2.1. コミット後のビルドフックの設定

ビルド後のフックを設定する方法は複数あります。以下の例に出てくるすべての形式は同等で、**bundle exec rake test --verbose** を実行します。

**手順**

- シェルスクリプト:

```
postCommit:
  script: "bundle exec rake test --verbose"
```

**script** の値は、**/bin/sh -ic** で実行するシェルスクリプトです。上記のように単体テストを実行する場合など、シェルスクリプトがビルドフックの実行に適している場合に、これを使用します。たとえば、上記のユニットテストを実行する場合などです。イメージのエントリーポイントを制御するか、イメージに **/bin/sh** がない場合は、**command** および/または **args**を使用します。

**注記**

CentOS や RHEL イメージでの作業を改善するために、追加で **-i** フラグが導入されましたが、今後のリリースで削除される可能性があります。

- イメージエントリーポイントとしてのコマンド:

```
postCommit:
  command: ["/bin/bash", "-c", "bundle exec rake test --verbose"]
```

この形式では **command** は実行するコマンドで、Dockerfile 参照に記載されている、実行形式のイメージエントリーポイントを上書きします。Command は、イメージに **/bin/sh** がない、またはシェルを使用しない場合に必要です。他の場合は、**script** を使用することが便利な方法になります。

- 引数のあるコマンド:

```
postCommit:
  command: ["bundle", "exec", "rake", "test"]
  args: ["--verbose"]
```

この形式は **command** に引数を追加するのと同じです。

**注記**

**script** と **command** を同時に指定すると、無効なビルドフックが作成されてしまいます。

### 7.2.2. CLI を使用したコミット後のビルドフックの設定

**oc set build-hook** コマンドを使用して、ビルド設定のビルドフックを設定することができます。

**手順**

1. コミット後のビルドフックとしてコマンドを設定します。

```
$ oc set build-hook bc/mybc \
    --post-commit \
    --command \
    -- bundle exec rake test --verbose
```

2. コミット後のビルドフックとしてスクリプトを設定します。

```
$ oc set build-hook bc/mybc --post-commit --script="bundle exec rake test --verbose"
```

# 第8章 ビルドの信頼される認証局の追加設定

以下のセクションを参照して、イメージレジストリーからイメージをプルする際に追加の認証局 (CA) がビルドによって信頼されるように設定します。

この手順を実行するには、Dedicated 管理者が ConfigMap を作成し、追加の CA を ConfigMap のキーとして追加する必要があります。

- ConfigMap は **openshift-config** namespace で作成される必要があります。
- **domain** は ConfigMap のキーであり、 **value** は PEM エンコード証明書です。
  - それぞれの CA はドメインに関連付けられている必要があります。ドメインの形式は **hostname[..port]** です。
- ConfigMap 名は、**image.config.openshift.io/cluster** クラスタースコープ設定リソースの **spec.additionalTrustedCA** フィールドに設定される必要があります。

**注記**

OpenShift Dedicated 管理者は **registry-cas** ConfigMap を使用する必要があります。

## 8.1. クラスターへの認証局の追加

以下の手順でイメージのプッシュおよびプル時に使用する認証局 (CA) をクラスターに追加することができます。

**前提条件**

- Dedicated 管理者の権限があること。
- レジストリーの公開証明書 (通常は、**/etc/docker/certs.d/** ディレクトリーにある **hostname/ca.crt** ファイル)。

**手順**

1. 自己署名証明書を使用するレジストリーの信頼される証明書が含まれる ConfigMap を **openshift-config** namespace に作成します。それぞれの CA ファイルについて、 ConfigMap のキーが **hostname[..port]** 形式のレジストリーのホスト名であることを確認します。

```
$ oc create configmap registry-cas -n openshift-config \
--from-file=myregistry.corp.com..5000=/etc/docker/certs.d/myregistry.corp.com:5000/ca.crt \
--from-file=otherregistry.com=/etc/docker/certs.d/otherregistry.com/ca.crt
```

2. クラスターイメージの設定を更新します。

```
$ oc patch image.config.openshift.io/cluster --patch '{"spec":{"additionalTrustedCA":
{"name":"registry-cas"}}}' --type=merge
```

## 8.2. 追加リソース

- Create a ConfigMap
- Secrets and ConfigMaps